Learn to use CGI in Two Hours

By Steve Humphrey

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Copyright and Legal Information

I'll keep this simple and to the point. You are allowed to install this software on one computer. You may move it to another computer any time you like, but you are only licensed to run one copy. You may make one backup on a floppy disk, Zip Drive, etc., for archival purposes − in case your computer's hard disk fails.

You are not licensed to sell, to give away copies or to distribute this software by any means: digital, mechanical or otherwise. Doing any of these things, except as allowed by Sections 107 and 108, is a violation of federal copyright law (title 17, U.S. code).

ISBN 0−9713986−0−7

Chapter 1: Overview

About the author

I've been using the Internet since 1993. I opened my first Web site in 1998. I've taught myself HTML, CGI, Perl and SQL. I am *not* an expert by any means. However, I have written Perl scripts to solve problems for myself, my clients and my friends. I hope that much of what I've learned will help you to master the art of writing, installing and using CGI scripts more easily (and faster) than I did.

Purpose of this book

This book is intended to teach you how to obtain (or write) and install CGI scripts written in Perl or other programming languages. The focus is mainly on scripting in Perl, but the installation process is very much the same for CGI scripts written in any other language.

By providing you with some complete scripts, some subroutines and the tools for combining them in new ways, I hope to teach you a "mix−and−match" approach to solving scripting tasks. I believe that if you follow the principles taught here, you can be installing CGI scripts on your Web server/site in a very short time.

Friends, this is not rocket science! It's simply a skill you have not yet acquired. There was a time (come on...admit it) when you didn't even know how to "surf the Web". You mastered that easily enough, didn't you? You can pick this up almost as easily. Once you've got it, you'll see that it really wasn't all that hard.

What you will learn

First, you will learn what CGI is, what a script is and how it works. Then, you will learn about subroutines and how they can be re−used to save lots of time and effort. You'll discover where to find free CGI scripts and how to get them. You'll find out how to take a useful subroutine from one script and use it in another script.

By applying what you've learned, you'll be able to create whole new scripts "out of thin air" by putting together pieces you found here and elsewhere.

You'll discover how CGI can make your Web site more interactive and how to make it perform repetitive tasks automatically. Your Web site should be able to work for you like a tireless employee. Your knowledge of CGI scripts will make this possible.

You will learn how to install a script on your Web server. We will cover using ftp to get the script into the right place, setting file permissions so the script can be run and connecting it to your Web pages. You will see how to tell whether the script is working properly and how to fix the most common problems that keep it from behaving as intended.

You will learn how to collect the email addresses of your visitors, add them to your mailing list and follow up with sales letters or other information. By doing this, you'll be able find new prospects and turn them into paying customers. You'll learn how to take payments and give receipts − online and by email. You'll see how to do any math you need to do and how to display properly formatted dollar amounts. Users of other currencies may need to modify these routines somewhat, but the principles will still apply.

You'll learn how to save data you collect in a database and extract usable information from that data. You will learn how to deliver error pages, thank−you pages and totally customized pages that are unique to each visitor during their stay at your Web site.

What this book is NOT

This book is not an exhaustive reference for programming in Perl. If you want to all about Perl, you'll need to get one or more of the excellent books available on the subject. Some of these are listed in Appendix A. For more guidance, visit the Perl home page:

http://www.perl.com

This book is also not a complete guide to installing scripts on every possible type and configuration of Web server and/or hosting company. That would require years of research and would fill many hundreds of pages. Not only that, but it would need to be updated constantly.

There are many "quirks" associated with different servers which can make installing a script more challenging. The basic principles are discussed here. For most users, this will be enough to enable you to install your own scripts.

If you are unable to get a script installed, it is usually best to talk to the system administrator or the technical support personnel at your Web hosting company. Many problems like this can be avoided if you carefully follow the instructions and procedures given here.

In rare cases, it may be necessary to have a professional install your script. A few reputable installers are listed in Appendix B.

How to use this book

I'm pretty sure you're going to skip around in this book, at least at first. So, there are several places where topics are repeated, discussed in more detail or you're referred back to another chapter. This is your book now; use it to your best advantage. Bounce around, read it straight through or whatever works best for you. I've tried to make it as easy as possible for all types of readers.

First, read Chapters 2, 3 and 5. This will give you a good understanding of what CGI is and help you determine whether you can use it on your Web site. If you find that there is problem with your host which will prevent you from using CGI, seriously consider changing to a different host. Many free sites, as well as some very low−priced hosts, will not let you run your own CGI scripts. If this is the case, please refer to Appendix B for some hosting companies that will give you more power and flexibility. Keep in mind that the some scripts are going to be difficult to install if your host does not allow you enough control.

If you did not find any of the limitations mentioned in the last paragraph, that's great! Now, read Chapter 6 to get a grasp of what's involved in installing scripts. As you go through this crucial area, I'll guide you every step of the way. Great care has been taken to present the tools and concepts in a logical order. If you find that you are sufficiently familiar with the contents of a section, feel free to skip ahead. I do suggest that you do everything in the order I've presented it, because you'll encounter far less trouble that way.

Now decide what task you want to accomplish with a script. If what you want can be done by one of the complete scripts provided here, you are in luck. See the next paragraph; it will tell you how to pull out that script and get started.

For any of the complete scripts, simply copy and paste all the code into a new text file. Save it with a name of your choice and a .pl or .cgi extension. I use WordPad for this purpose, but any text editor will do. Don't use a word processor (such as Microsoft Word, for example). Word processors insert invisible (non−printing) characters for formatting which will corrupt your script.

If you didn't see a script that does exactly what you want among the ones included here − don't worry. There are loads of CGI scripts free for the taking on a number of Web sites. Chapter 4 will show you a lot of places to shop for free scripts. Some will also have scripts for sale. In a few cases, you'll come across a script (usually a very complex one) that requires a monthly or yearly payment for using it. Often, such scripts reside on the owner's server, and you merely call them with a few lines of your own code.

My advice is simple: If you can find a free script that does the job...use it. If not, consider putting together a script of your own. I'll show you how to do that. You can use one or more of the subroutines included here to build a new script or add features to an existing one. If you aren't comfortable with the job of composing a new script, search around for a paid one that meets your needs and buy it.

For subroutines, copy and paste them into some existing script or a new one you are starting. Be absolutely sure that you get the closing "}" character of the subroutine, or they won't work!

If you buy a script, you can install it yourself in most cases. Take your time, read all the instructions and do exactly what they say. You may have to edit a line here and there to customize the script. After you've read Chapters 6 and 7, you'll see that this is really pretty easy.

If you are going to write (or at least piece together) some scripts of your own, you'll want to read Chapters 8 through 17. Feel free to skip any sections that you don't need for your current project. You can come back later when you need to learn what they can teach you. If you stop at Chapter 7 and come back later, be sure to at least skim Chapter 7 and review the mechanics of writing a script. As with many skills, you may have forgotten something if you haven't used it for a while.

There will be times when you just want to modify an existing script. Perhaps you want to add a feature or change the way something works. In Chapter 12, you'll discover how to work with subroutines. This comes in handy for adding one or more features to an existing script. You can often find a subroutine that provides just the feature you want to add.

Here's an example: Let's say you found a really useful free script that uses sendmail to compose and send email messages. If you're on a Unix host, it will probably work just fine. Almost every Unix host has the sendmail program installed. However, if your site is on a server that runs Windows NT, the script cannot work. Why? Because sendmail is a Unix program! Most NT hosts have a mail server program called iMail. The current version is 6.05. Here's a case for modifying a script. See Chapter 14 for advice on how to convert the script to use iMail. The reverse case is also covered.

By the way, the documentation of a great many scripts says the script will not run on NT. Quite often the problem is simply that the script expected to find sendmail or qmail. Replace the mail−handling subroutine with one from here that works with iMail. If the server is using iMail, your script may work now! If not, there may be other differences to work out.

Notations used in this book:

Note: These items contain extra information about the current topic. They may help you avoid pitfalls and increase your understanding of the material.

Tip: These items offer shortcuts, time−savers and clever tricks you can use.

Chapter 2: WHAT IS CGI?

Definitions

Server − a computer that runs programs for its clients. A Web server, for example, runs mail programs, delivers Web pages, maintains Domain Name Services, etc.

Client − a user's computer, such as your own PC, Mac, etc., which requests files or services from a server. Client also refers to the programs you run which need services from another computer or a program running on another computer (a server). For example, your Web browser or FTP program is a "client" which requests and (usually) gets services from another computer and/or program. When you visit a Web page, your browser (the client) asks a Web host (which is running Web server software) to send you a file (an HTML document or other file). In any networked environment, such as the Internet, there are a number of servers and a much larger number of clients. Similarly, if you have a telnet (client) program, you can log in to a remote computer's telnet (server) program and operate that computer from your own keyboard and monitor.

Unix − a multi−user, multi−tasking operating system for computers made by Sun, DEC, IBM and others. Multi−user means that many users (people) with separate work−stations (computers connected in a network) can use the servers (computers which hold application programs, shared files, etc.) together at the same time. Multi−tasking means that many programs can be run at the same time.

The majority of Web hosts (servers) are using the Unix operating system. There are many versions or dialects of Unix, including Solaris and the wildly popular Linux (which comes in many varieties including Red Hat, Slackware, Debian and others).

In this book, Unix may be taken to mean the same as Linux for all practical purposes. A typical modern Web server (computer) will have the Linux operating system and the Apache Web server (software to manage Web sites).

Another popular server configuration is Windows NT (operating system from Microsoft) with a Web server (software package) such as WebSite.

CGI (Common Gateway Interface), is simply a way of exchanging data between your browser and a program running on a Web server. It is made up of a set of standardized variables used to pass specific types of data between a client and a server. (Now you see why I wanted you to know the terms "client" and "server".)

FTP (File Transfer Protocol), is a way to send documents or files from one computer to another using the Internet. An FTP program, or client, is the program you use to send (upload) or receive (download) files to or from a computer (server) connected to the Internet. When you send Web pages to your site, you are using ftp, whether you start an FTP program or use a control panel on your Web host.

Telnet − a client program that lets you use a remote computer. Once logged in, you can run programs, list directory contents, set file permissions (Unix systems), etc., as if you were actually at the console of the remote machine.

Program − a set of instructions that tell a computer how to do something. Since the instructions don't usually change from one execution to another, a program generally does exactly the same thing every time it runs. If a program's outward actions are directed by data given to the program when it starts, the visible output will be different for different inputs.

Script − often referred to as a program, a script is actually a little different. Sure, it acts just like a program. It follows the description I just gave you for a program, but the difference is in how it runs. A program is ready to run immediately (its instructions are in machine code) but a script is interpreted line by line. There is a command processor or shell program on the server that figures out what each line means and then runs it. For Perl scripts, this processor is the Perl interpreter. Its name is perl.exe, and it is a program by the first definition.

How CGI works

This is a big topic, and I'm only going to give you a brief description. In general, data of some kind is passed to a CGI program or script. Taking the classic example of filling in a form on a Web site, the form is the first link in the chain of events. Data items are typed into the form by a user. Then the user clicks on the button that "activates" the form.

A script is called, which takes data given to it by the form. It then decides what actions to perform on or with the data. Finally, it produces some output. This output can be in the form of sending an email message, delivering a customized (or stock) Web page or perhaps reading from/writing to a database. The form (client) has requested the services of the script (server), which does its thing and sends the result back to the client (browser, mail program, database, etc.)

CGI programs run on the Web server. Because of this, your visitors can use any browser they like. Other types of Web programming such as JavaScript, actually run on your visitor's computer. If the user has an older browser which can't run JavaScript, your efforts are wasted on them. Likewise, if they have a browser than can run JavaScript but have disabled it in their preferences, they won't see the whiz−bang interactive features you worked so hard to implement.

About the only downside to all this is that running CGI programs make your Web server do the work. But I just said that was good, didn't I? Everyhing has a cost, and CGI is no exception. If your server is under−powered or overloaded, it may slow down as a result of running your CGI programs.

Programming languages

Many different languages can be used for CGI. Perl, Python, ASP, C++ and Java are a few that have become popular. PHP is gaining ground, especially where high−performance programs are needed. The rule is basically that any program that can be compiled and run on your server can be a CGI program. Interpreted languages such as Perl and Python can even be compiled into an executable form. However, they are most often left as scripts to be interpreted by the server's command processor.

Most of the database programs in use today can be controlled by statements in a standardized language called SQL (Structured Query Language). Programs and scripts can send SQL commands to a database server program running on a host computer. Methods of preparing and sending the commands may vary, but the commands themselves are nearly universal. You'll see examples of how a Perl program can send commands (or queries, as they are usually called) to a database server later in this book.

What can I do with CGI?

You can write and run scripts or programs that make your Web site more "interactive". In other words, you can use it to make a Web page actually do something.

Typical CGI programs do things like gather information from a form, send email messages or read from/write to a database. They can be used to automate all sorts of processes like subscribing people to your newsletter, taking payments for products and/or services that you sell and so on. Remember that you'll need some type of payment processor such as a merchant account, ClickBank, etc., to actually get the money and send it to you.

Chapter 3: What Is PERL?

Definitions

PERL − Practical Extraction and Reporting Language; Perl is an interpreted language (like BASIC from the Dark Ages of computing).

It is handled by a command processor which determines the meaning of each line as it reads the script. Despite being processed by this other software before actually running, Perl is very fast.

Perl used to be known as "the language of the Internet". This is not as true as it once was, but Perl is still extremely popular. Reasons for this include its power, speed and ease of use. Aside from the many odd−looking operators, it's easy to read a Perl script if you have any experience with a structured programming language such as Pascal, C or C++.

Why use Perl?

As I just mentioned, Perl is fast, easy and relatively simple. Its string−handling capabilities make it a great choice for processing information from forms, producing customized Web pages and much more. It can do math operations, talk to databases, manage lists of items and many other tasks using very little code.

Since it supports subroutines, you can reuse blocks of code from earlier scripts in a new script. See Chapter 12: Using Subroutines for more about this topic.

A third good reason is availability. Almost every Web server in existence has a copy of the Perl interpreter installed. This is partly because it's free. Perl is supported, maintained and improved by a community of users and programmers.

It is made available under the GNU General Public License. Server administrators don't have to buy Perl and keep paying for every upgrade. That's a great incentive to keep a current version of Perl on their computers.

Another reason to program in Perl is that it's pretty easy to learn. You can pick up the basics in a few hours and learn more advanced elements as the need arises. You can go to the Perl home page and search for a topic that interests you. Chances are good that a script or a module has already been written which you can use right now.

Within minutes, you can be reading the code of a script that does some or all of what you need for your current project. If it can be used "as is", just download it and use it. Or, if some part of it solves a small problem in a larger project, grab the that part of the code, paste it into your script and make whatever adjustments may be needed. Often it's just a matter of matching a few variable names so the new code fits right in with yours.

What version do I need?

For best results in all situations, I suggest using the most recent version you can get. Currently, version 5.004 or higher is available on most systems. This version will support practically anything you can ask a Perl script to do. The most recent version can always be found at the Perl home page:

www.perl.com

DO try this at home

Get your hands on one or more books about Perl. Even if you never write a script, it's good to know how it works. That will help you decide if a free script you're considering will do what you want, the way you want it done. There are several good ones listed in Appendix A.

These and others can be bought at bookstores like Barnes & Noble, BookStar, Books−A−Million and Borders. Feel free to shop the online stores as well; you may find a bargain at Amazon.com or bn.com that beats the store price. Don't overlook used−book stores either! For those on a very tight budget, borrow them from a library or a friend. Technical Community Colleges, state Universities, etc., have vast libraries with lots of this kind of stuff.

Read a book about Perl programming and work through the examples. Get familiar with "reading" and understanding how the various parts of a Perl script work. Next, download some completely unfamiliar scripts and use what you've learned to figure out exactly how the script works.

After you do that a few times, you'll be able to understand most of what you read. If you see something you don't understand, look it up in a Perl manual. Typically, the most confusing parts are the strange−looking "operators". So, start with a cryptic statement like this:

```
/^[\w\d][\w\d\,\.\−]*\@([\w\d\−]+\.)+([a−zA−Z]{3})$/
```

Find a list of operators in the book and see which of the symbols in the line you're reading are operators. Replace them with plain English words that mean the same thing as the operators. Keep replacing symbols with words until you have a readable sentence. Now go back and look at that line again. It has magically become readable to you!

You see, programming languages are just a way to tell the computer what we want, in a way it will understand. Perl is no exception to this rule.

Chapter 4: Getting Free Scripts

This is child's play!

Look in the list of links below! Collectively, these sites have hundreds of scripts. Most are free; there are a few that cost some money. Also on some of these sites are FAQ (Frequently Asked Question) files, debugging tips, HTML help, Java and more. Pick a site, rummage around and download something. Look it over and see if it meets your needs.

Even if it won't solve your problem right this minute, it could be a learning tool. See the section above, titled "DO try this at home". Or, it may have one or two subroutines that are more efficient than ones you're using now.

By now, you've got your foot in the door. So, just kick it wide open and see what's on the other side! Be very careful when you buy scripts, but otherwise just dig in and have fun with this stuff. You'll discover lots of ways to add life and profitability to your Web site.

Sources for free scripts, debugging help, etc.

- BOUTELL.com
- CGI/Perl Tips, Tricks, and Hints
- CGI Resource
- Cliff's Perl Scripts
- Comprehensive Perl Archive Network
- www.dtp−aus.com
- Extropia.com
- Free Code
- Freeware Web
- freewell.com
- go2cgi
- HotScripts.com
- Lake Web
- Matt's Script Archive
- myCGI scripts
- New Millennium Network
- Perl Coders
- Script Crypt
- Script Search
- www.ShavenFerret.com
- Site Point
- Solution Scripts
- Webad Design
- Web Developer's Virtual Library
- Web Thing

Compressed files

You'll very often find files in a compressed form on the Web. File compression makes them smaller so they take less time to upload and download. These compressed files are often archives, which contain several compressed files. This is good, since you only need to download one file.

These files are most commonly end in ".zip". You'll need an unzipping program such as WinZip to restore them to their original size before you can use them. Using WinZip (and other similar programs) is convenient for a number of reasons. It's possible to display a compressed file while the archive is still "zipped". That way, you can check for any text files that may contain instructions you need to read *before* you unzip the archive.

Other compressed file types are ".gz" and ".tar.gz" These files are for Unix users and will not work on a Windows PC or a Mac. Be sure you get the right type of compressed archive for your particular machine. Oddly enough, the unzipping utility that comes with most versions of Unix can uncompress a zip file. You can then read any text files that were included, although the program itself might not work on a Unix machine.

Chapter 5: System Requirements

Here, I'm referring to the requirements for your Web host. These are things you'll need to have in place before running any CGI scripts written in Perl. If you choose to use CGI programs written in a different language, other requirements may apply. Be sure to read any material concerning these requirements before you try to install a CGI script. The more of these items that can be verified and checked off your list before you begin, the smoother and quicker the installation will be.

For those of you who are really serious about learning Perl, I most strongly recommend you get a book that comes with a CD−ROM. This CD−ROM should have the code described in the book, and a Perl interpreter (and other supporting files) that you can install on your own PC. Once you've installed this code, you can write and, more importantly, debug Perl scripts before you try to install them on your Web server.

If the last paragraph applied to you, read these 3 paragraphs carefully! It will save you a lot of grief. I found that after installing a Perl package (that was intended for Win 95) on my computer (running Win 98), that none of my scripts would run. I'm talking about fully debugged and perfectly good scripts that had been working fine before!

Here's why: The installation program had failed to edit the PATH statement in the AUTOEXEC.BAT file. I found a solution and here it is: Use Find (on the Start menu) to locate the program sysedit.exe. It should be in the Windows/System directory. Run the program using the Run function on the Start menu. It will open several small windows. Look for the one labeled AUTOEXEC.BAT; it's usually the one "on top" of the others.

Find the PATH line in this file. You can edit the file in the window, so add the path to Perl. In a PATH statement, several "search paths" can be specified. You separate them by semi−colons (;). Make sure you know the complete path to the file perl.exe, and the path to the other Perl files, too. In my case, this was "C:\PERL" and "C:\PERL\BIN". Add these to the PATH line, like this: C:\PERL;C:\PERL\BIN Click Save from the File menu of this window and quit the sysedit program. The change will take effect the *next* time you restart Windows.

Be aware that a perl script you run on your PC will need the #! line set to the actual path to perl.exe on your system. For example, it was #!C:\perl\bin on my system. This "bang−path" will not work on a Web host. Change it to the right path for the host *before* you upload the script. In case you're curious, it's called a "bang−path" because it begins with a "!", which we geeks pronounce as "bang". We also call a * a "splat". Go figure.

For example, if you are installing a script/program written in PHP, you need to know which version of PHP in already installed on the server. It may or may not be compatible with your script or program!

There are several small scripts that will show you the path and environment information you need for further installations. CGI.ID from: The Matrix Vault is great for getting all the environment information. While you're at their site, check out the other scripts they offer.

The Root path

Some scripts will need to know the "root path". Most will work using a "virtual" or "relative" path, but others won't. Get this information from your host administrator. Be sure they give you the actual path. Sometimes the diagnostic scripts return a virtual path.

The Perl interpreter

For running CGI scripts/programs written in Perl, this is the single most important thing to get right. Your scripts must know where to find the Perl interpreter. If you visit the support page of your Web host, you may be able to find this path. Be warned, though, that sometimes the information is outdated! Someone may have moved this vital file to a new location and not updated the support page. Other times, the information is simply not there at all.

What you are looking for is something like this:

#!/usr/bin/perl

If you suspect that you have been given an incorrect path, immediately send an email to their technical support address and request it. If you cannot get them to tell you what it is, or they tell you that you're not allowed to install your own scripts, you need to find a different Web host. Period.

If you have telnet access, you can start a session and type "which perl", "which mail" and "which date." This will get you the vital info. Another way of finding out what version of Perl is installed is to use the "diag.cgi" script from BigNose Bird's Trouble Shooting Page. This little gem will also reveal the locations of the date and mail programs. Still another is the findprograms.cgi script I've included with this book. (See Appendix D).

Note: This may not work on NT systems.

The mail program

This will be vital to any scripts that need to send an email message. Let's begin with Unix systems. Nearly all of these systems have the program sendmail installed. You'll need to know where this program resides (on the Web server) so that you can use it. The most common location is:

/usr/bin/sendmail

As with the Perl interpreter, you should be able to get this path from the support section of your host's Web site. If not, simply email or call tech support and ask them. Email them first. If they don't respond within a day or two, call them. This is a simple test that will show you how willing tech support is to help you when you have problems.

Once again, this is a vital tool for your operations, so be sure you are allowed to use it. If your host has sendmail installed but won't let you use it, then you need to find a new host. Remember, email is the most important weapon in your sales arsenal. Don't go online without it!

For Windows NT systems, the most common mail program is iMail. The current version is 6.04 (or possibly higher). This program is sold by IPSwitch (for $995.00). Their Web site is located at:

www.ipswitch.com

Visit their site and download the user manual in PDF format. Be sure to right−click on the filename. If you left−click, it will open on your screen instead of downloading.

You'll probably have to ask tech support for the path to imail.exe. At my host, it was:

C:\\IMAIL\\IMAIL1.EXE

Since we're talking Windows here, it's most likely not case−sensitive (imail1.exe refers to the same file as IMAIL1.EXE). Did you notice that there were two backslashes between the parts of the path? This is something that matters a great deal in a script. You see, the backslash (\) character has special meanings in a Perl script. So, to tell your script that you mean to use a backslash without a special meaning, you have to precede it with another backslash. What this actually does is indicate that the character immediately after the back−slash is *not* a special character. This is referred to as an "escape sequence". You'll see this type of thing a lot. For example, an email address contains the "@" character, which must often be "escaped" with a "\" because Perl has a special meaning for "@".

In scripts that you write or modify, you'll have to have a statement that assigns the location of the mail program to a variable. There are ways to avoid doing this, but they are really more trouble than they are worth. Just do it the easy way once, and it will save time and effort later. Here's how that would look in a typical script:

$mailprog = "/usr/bin/sendmail"; (for Unix systems)

The double−quotes around the path tell Perl to save this as a string of characters. They won't be included when the variable is read by a routine that needs to call the mail program.

CGI−BIN access

This is truly the heart of the matter at hand. To be able to run any CGI programs or scripts, you *must* have access to this directory. If you can't get it, you're just plain dead in the water. When you look at the features offered by a hosting company, they should proudly mention that they grant you "full CGI−BIN access", or say that "you get your own cgi−bin directory" or some other such wording. If they fail to mention it, odds are it's not available to you. This is a case where you need to check with tech support *before* you sign up for the hosting service.

There are two main issues here. First, can you put your script (s) in this directory? This is most often done with FTP. You start your FTP client, select the "local" directory (on your machine) that contains the script, select the cgi−bin directory on the "remote" machine (your Web server) and transfer the file from your machine to the Web server. Second, is this a secure operation? What I mean is this: If you must use a UserID and a password to get into your Web host's FTP server, then your ftp access is not "anonymous". That means that only you can upload, rename, delete, etc., the files in your cgi−bin directory. That's how you want it to be.

NOTE: The directory name "cgi−bin" is nearly universal on Unix machines. However, if your host is using the Windows NT operating system, the name (s) may be different. For instance, on my host (it's NT), there are "cgi−shl", "cgi−win" and "cgi−shl−prot" directories. I've been told that they all work basically the same. If there are differences between them, I don't know what they are...and it really doesn't seem to matter. I just put all my scripts in "cgi−shl". In fact, it's often possible to put a perl script in a "normal" directory like /htdocs or /htdocs/special_stuff. Your mileage may vary.

Chapter 6: Installing CGI Scripts

Using FTP

First let me say a little about FTP client programs. First, you *must* have one (or several). I have three favorites. I'm not a reseller for any of these; choose whichever one suits you best, or shop around for a different one altogether. Just take my word for it that I've downloaded and tested literally dozens of these clients and these are the ones I liked best.

Overall, WS_FTP LE is the best free FTP client for CGI installers. It makes it easy to set the file permissions so that your script will execute (Unix systems). It's literally a point−and−click operation; you don't have to know how it works. Also, it comes with login data in a list for several FTP sites. You can type the login info for your site (including UserID and password) and it will add it to the list. Add as many sites as you want. Select the one you wish to access from the list, click Connect and presto! You're there. Get WS_FTP LE v5.08 here:

www.download.com

Use the search box and type in WS_FTP. It's been downloaded over 2,000,000 times, and 96% of users recommend it. So do I! It's free for most users. If you don't qualify to use it free, they offer a Pro version (v6.6) for $39.95 via download and a $9.95 upgrade to get even more features. If you will be doing a lot of CGI installing or just need a more feature−rich FTP client, this one's outstanding.

For Web site maintenance, Cute FTP is a great client. It allows you to transfer multiple files at once. That's good for backing up a Web site to your local machine and Cute FTP makes it easy. Download it from:

www.download.com

Use the search box and type in Cute FTP. Get the 32−bit version. The current revision is 4.2.3. It comes as shareware with a 30−day trial (all features enabled). After 30−days, several features, including multiple file transfer are disabled. Registration is $39.95 and will restore it to full power.

Then there's one rare beast, NFTP. This little gem can "see" and display hidden files! For a project where I was using data files that began with a ".", this was the first one I found that could display them on the remote machine. It offers a side−by−side display screen, with a list of the files in the current local directory in the left pane. The right pane displays the contents of the current "remote" directory (at your Web site, for example). When you log in to a site, it can take you to the desired directory initially, so you don't have to traverse the file system when you get connected. It can set file permission for Unix systems, but it's not as intuitive as WS_FTP for this purpose. Download this program from:

www.download.com

Use the search box and type in NFTP. The current version is 1.62. It's a shareware program with a 60−day free trial. After 60 days, it starts asking you to register it, which costs $25. If you work with hidden files (ones whose names begin with a "."), this is a "must−have" utility.

Cardinal Rule #1:
Whatever FTP program you use, be *absolutely* sure you are uploading the script file(s) in **ASCII** mode, not **BINARY** mode. If you get this wrong, your script will never work. Text editors insert a "^M" wherever there's a carriage return. In **BINARY** mode. these characters will be transferred along with the rest of your code. They will confuse the Perl interpreter.

Getting Started: I'll assume you've just acquired an FTP client program, unpacked the archive and run the Install or Setup procedure. Now, it's set up on your home computer. If you're already using ftp and are comfortable with your level of expertise, you can skip the rest of this section.

If not... Start up the program, read the Help files and play with the controls a bit. Log into an ftp site somewhere and get used to how downloading works. it may not seem as easy as it is on the Web − at first. Practice it a little and it will become routine.

Now, I'll assume you have a Web site, and you want to move a file to it. Let's start with something very safe. Send a plain text file to some directory on your site.

Now change the local directory in your ftp client. Next, download the file from your site to this directory. Load the file into WordPad or NotePad. Now, start a second copy of WordPad (or NotePad) and load the other copy of the text file. Compare the two; they should be exactly the same.

Use the ftp client to delete this test file from your Web server. This harmless little exercise should have convinced you that you can use FTP without any problems at all.

Note: While many people recommend using NotePad, it does have one important shortcoming. It can't handle files over 32K in size. For larger files, use WordPad or some other text editor.

Setting file permissions

OK, now let's assume you're ready to begin installing a CGI script. Perhaps you obtained a script that you want to install; maybe you just wrote (and debugged?) one of your own. Either way, be sure it is ready to work on your Web server. Did you make sure the path to Perl was correct? Did you add the line that localizes the path to the mail program to a variable? (See Chapter 5 for help if you need it.)

If you're sure both of these jobs have been completed successfully, go ahead and upload the script to the cgi−bin directory (or wherever they need to be on your server). If you're on a Unix or Linux host, you must set the file permissions. Depending on which FTP program you're using, there are different methods for doing this. The permissions must be:

- **Owner − read, write and execute (rwx in Unix terms)**
- **Group − read and execute (write is usually optional)**
- **Others − read and execute (write is usually optional)**

If your script will be writing to files, the directory containing those files will usually need to be set to allow everything. Owner −rwx, Group − rwx and Others − rwx. This is permission mode 777.

If you're using WS_FTP, this part is really easy. Right after you uploaded the script file to cgi−bin, there was a list of files in the right−hand pane of the WS_FTP window. You right−click on the script file's name to bring up an option menu. Choose chmod (Unix) from this menu. Now you get a little dialog box with 9 checkboxes and a couple of buttons. They are grouped into Owner, Group and Other from left to right. They are grouped as Read, Write and Execute from top to bottom. Click all three boxes marked Execute so that there's a check mark or dot in the box.

Click the OK button. That's it! Now, whenever you right−click on the file and bring up the chmod (Unix) box, you'll see that the correct boxes are checked. Tip: Verify this *any* time you edit the file and send a new version to the server. Sometimes the permission go back to the default state (which does not include Execute). If this happens, just set them again and all will be right in your world.

Some programs, such as telnet and certain ftp clients, give you access to the Unix command prompt. If this is the case, setting file permissions is also a breeze. Just type:

chmod 0755 [filename] or
chmod 0777 [directoryname]

Telnet requires the 0, but it doesn't affect the file permissions.

Just in case you're wondering about the numbers, here's the story. Read has a value of 4, Write has a value of 2 and Execute has the value 1. Adding the permission values together results in a number between 0 and 7. There's one of these "composite" numbers for each of Owner, Group and Other. So, if you do a chmod 755, you've set Owner to Read+Write+Execute, Group to Read+Execute and Other to Read+Execute.

If you fail to set the permissions properly, here's what can happen: When the server is asked to run the script, it will display an error page. Most often, it will say, "ERROR 403: Forbidden". This means that the file does not have "execute" permission − the server thinks that nobody is allowed to run it. To fix the problem, repeat the procedure in the last paragraph. This doesn't usually happen on Windows NT hosts, because most of them don't have the concept of file permissions.

However, depending on the Web server software package being used, the tech support folks may be able to deny execute permission to scripts on a case−by−case basis. This brings up a point that cannot be over−stressed. Always treat tech support people with courtesy and respect. They have a lot of power that you don't. They're often over−worked and stressed−out from dealing with hackers, viruses, "Trojan horse" programs, mail system hijacking and a lot of other very bad things you don't even want to know about!

Chapter 7: Writing and Editing

Text Editors

WARNING: Text editors are *not* the same as word processors; know the difference. A text editor such as NotePad won't put things in the file you can't see. Exception: sometimes another text editor may display a ^M wherever there was a carriage return in the file created by NotePad. This generally has no real effect, as long as you remember to always upload in ASCII. On the other hand, a word processor such as MS−Word or Word Perfect will put all sorts of strange stuff in the file. It's supposed to do that; they are special characters that don't display but are used to control the way a document looks on screen and on paper. These special characters will wreak havoc on a script. It's just not possible to create a properly functioning script with a word processor. Don't even think of using one.

There are plenty of free or inexpensive text editors available. If you don't like the ones that come with Windows, or whatever OS you're using, feel free to use something else.

My personal favorite is Unix program called emacs. It's found on virtually every Unix system. It's included with all Linux distributions. So is "vi", but I didn't learn that one. One of the coolest things about emacs is that it is capable of some basic HTML editing. Plus, it's great for writing computer code because it will "highlight" reserved words (used for commands, etc.) in different colors from the other text. Comments are highlighted in still another color. It checks for mismatched parentheses and "curly braces".

These functions help keep you from writing code that won't work. The reserved word "sub" that signals the beginning of a subroutine is highlighted, along with the name of the subroutine − if the routine is written legally. Other clues like the presence or absence of bold text tell you when and where there is something seriously wrong with your code.

Bonus Advice: You don't have to take my advice, but you paid for it so here it is. I encourage you to consider installing the Linux operating system on a second hard drive, or even a separate computer. Unlike Windows, it will run very well on an older machine with a less−powerful processor. You'll get a nicely graphical user interface, access to tons of free software from a bunch of sites on the Web and it already comes with emacs and other great tools. You'll get a complete set of Perl files, so you can write, debug and test your scripts without any practical limitations.

Sound good? Hold onto your hats, folks − it gets even better! Unix systems cannot run the executable files written for Windows. Why is that good? I'm glad you asked! Viruses written for Windows PCs generally can't run on a Unix or Linux machine. So, you can get a great deal of immunity to viruses. Not ready to order yet? Here's another bonus! Ever thought of buying Microsoft Office but didn't have the big bucks it costs? Get StarOffice v5.2 for Linux. It'll cost you $39.95 (plus a shipping charge) to get it on a CD−ROM, or you can just download it for free. It has all the applications you expect: Word Processor, Database, Spreadsheet, Scheduler, Mail and Newsgroup Manager, etc. Download it or order the CD−ROM from Sun Microsystems here:

http://www.sun.com/products/staroffice/get.html

How can I learn to write Perl scripts?

The best way is probably learning by example. Read other scripts and see how they work. A good Perl reference book is a big help. Once you begin to see how Perl scripts work, you can write your own by coding them from start to finish or by combining parts of existing scripts to create new ones.

This book will give you several ready−to−run scripts, and a number of subroutines you can use to build more complex scripts. You'll see a complete script in just a moment.

The first line

Be sure that the first line of the script contains the exact path to Perl. If you don't know what it is, you should ask your system administrator or technical support contact. Hint: It's most often /usr/bin/perl which requires the first line to be:

#!/usr/bin/perl

Here's what it means: The "#!" part tells the server to expect a path to the Perl interpreter. Immediately following the "!" is the actual path.

Note that forward slashes indicate subdirectories, just like in a URL. This is how Unix does things − and the Internet began on computers running Unix. DOS and Windows came later, with their backslashes indicating subdirectories.

Initializing variables

First, what are variables? They are special names that stand for items of data your script needs to remember for some reason. Don't begin the name of a variable with a digit; Perl and most other languages have a rule against this ($1bigvar is illegal). There are scalar variables that store a single data item and array variables that store a list of items. Scalar variable names are prefaced with "$". Suppose you wanted your script to count something, starting with 0. You'd have a variable such as $count, and you'd assign it a value of zero at the beginning of the script like this:

```
$count = 0;
```

Since you gave it an "initial" value; you "initialized it". Easy, right? Sure it is. Now what if you want your script to know about a group of items like a list of colors, for example? You'd initialize an array. Its name must begin with "@". It could be done like this:

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

While the program is running, you can refer to "red" as $colors[0], "blue" as $colors[2] and "orange" as $colors[4]. The number in [ ]s is called a subscript; it tells the program where to look in the list for an item. This is how arrays are handled in almost every computer language. Perl is more flexible with arrays than most. A single item from the list starts with "$", so your program can tell which things are items and which things are lists. Since the "items" in the list are scalars and scalars can be numeric or string data, a list can be all mixed up:

```
@mylist = (1, "five", 3.14, "Bob", "email", 42);
```

Perl takes all this a step further than other languages, which lets you do really cool stuff. It has another array type called an "associative" array. Its name must begin with "%". The subscripts in this array can be anything you want them to be! How cool is that? Here's an example:

```
%fruit = ("pears", 2, "lemons", sour, "grapes", 100);
```

These arrays use { } to indicate the subscript. So now, $fruit{pears} means 2, $fruit{lemons} means sour and $fruit{grapes} means 100. Notice that when you defined the associative array, you listed a subscript, then a value, then another subscript, another value, etc. That's important to remember!

Comments: documenting your code

After the first line, any line that begins with a "#" is a comment. the command processor will ignore all these lines. This works in Perl and many shell−scripting languages. Other languages such as C++ use // or /*...*/ to indicate a comment. This can be used to your advantage in two ways.

One, you can add plain language that explains what a certain part of the code is supposed to be doing.

Two, you can temporarily put the comment character in front of lines of real code that you don't want to run. Now, why would you want to do that? It's a way of debugging the code. See the section on Debugging for a more complete explanation.

The main body

OK, now we've got things set up so we can run our script and given it some initial data. The main body is where we put code that only needs to run once (or needs to run in a very specific order). It's also the place where the action starts and (usually) ends.

Peek ahead to the end of this chapter. There's a section called "Anatomy of a simple Perl script". This script only has a main body; there are no subroutines and no calls to anything outside the main body. Only a very small script is ever likely to look this way.

What is more often done is to make the main body little more than a bunch of calls to subroutines. There will usually be some decision (if) statements here and there. They act like traffic cops, directing the flow of the script. They follow this pattern:

```
if (this condition is true) {
  execute this code
}
elsif (another condition is true) {
  execute some different code
}
else (this final condition is true) {
  execute this code
}
```

Depending on how the decision must be made, there can be different numbers of clauses. There might be just an "if" statement, an "if" with an "else" or an "if" with one or more "elseif" clauses and then, finally, an

"else" clause.

At the very end is an "exit" statement. This signals the server that the script has finished. In the main body, you don't always need it but there's no harm in having one. This statement consists of the word "exit" followed by a semicolon. An "error level" is optional but, when used, it's enclosed in parentheses and placed between the word "exit" and the semicolon. A value zero is commonly used.

Subroutines

What the heck is a subroutine? I'm glad you asked that; it shows you're paying attention. A subroutine is a block of code that is treated as if it were a single object. For instance, if you have a subroutine that writes data to a database, it would look something like this:

```
sub write_data {
    # code to actually write to the database goes here
    # if you want the subroutine to send back some value
    # when it finishes, use a return statement like this:
    # return ($success);
}
```

This line would make the subroutine run:

&write_data;

If a subroutine will return some value, that value can be saved in a variable for later use, as in this line:

$result = &write_data;

Subroutines can be reused! Once you're written (or found) a really useful subroutine, you can use it in another program just by copying and pasting it in. Add the code to your script and call it using either one of the methods shown above.

You'll see examples of scripts with subroutines and calls to them later in the book. When you put a script together, the subroutines are nearly always listed after the main body.

Filename conventions

CGI scripts that will be run from the /cgi−bin directory can usually be named "script.cgi", regardless of what language was used to create them. If they will be run in some other directory, they should be named "script.pl" if they are written in Perl. Windows NT systems are notorious for requiring the filename extension to be ".pl". Some of these systems don't even have a /cgi−bin directory, although this does not necessarily prevent you from running CGI scripts. For example, on my host I put CGI scripts in a directory called cgi−shl.

Debugging

Suppose you're having a problem getting the script to run properly. You can "comment out" parts of it that you suspect may be causing the problem. You just put the "#" character in front of every line in that part of the code. Save the file and try to run it again.

If you told the interpreter to ignore a certain part and the script ran fine without it, then your problem is in that part of the code! You can use this method to turn off bits of code until you find the one that kills the script. Now, turn on other parts that you had turned off, one by one. As long as a piece doesn't kill the script, leave it turned on.

Once you've narrowed it down to one section of the code, examine that section closely for some type of mistake. Fix that section, save the file and try again. If more than one place is giving trouble, just fix one at a time. When one section is fixed, move on to another area. When it works with all the code turned on, look at any output to see if the results are what you expect. Some would say that this is a brute force approach, and they would be right. However, I can tell you that it does work.

A better approach is to read the error logs on your server. They should give you more detailed information about what went wrong. A script called LASTLINES.CGI from The Matrix Vault will prove extremely valuable for reading out the errors in the log. If you don't have access to these logs, you should seriously consider finding a better host. See Appendix B for my recommendation.

Perl comes with debugging tools. The simplest are the error messages you get when a script refuses to run. They are often all the clues you need to fix a problem. Other problems, especially those that don't keep the script from running, will require you to use the debugger. Start the debugger by typing:

perl −d [filename]

See your Perl manual for a full description of how to use the debugger.

Common errors and how to avoid them

First and foremost in the error department is mismatched brackets or parentheses. What this means is that these symbols must be used in matched pairs. If a block of code begins with a "{", it must end with a "}". The same rule applies to parentheses and the square brackets used to indicate array indices.

The easiest method I've found for avoiding this kind of error is to type in both symbols when you want to use a block of code in between them. For example, if you're going to use an "if" statement, the rules say that you have to enclose the test condition in parentheses.

So, type if( ) and stop. Then put the cursor between the parentheses and type the conditional statement. The next rule for the "if" statement says that the body (the code that will execute if the condition is true) must be enclosed in "curly braces" ({ }). So, put the left brace right after the conditional statement. This "opens" the body. Then skip a line and put the right brace on the next line. This "closes" the body.

Note: You have to follow these two rules for every "if" statement, even if the body is only one line of code. Now that the body is properly punctuated, put in your code.

ALWAYS be sure you've used the right operator for the operation you want to perform. Perl has a *LOT* of different operators. Not only that, but there are two or more operators for what seems like the same job. If you use the wrong one, the result will be wrong. Here's a classic example: Comparing two numbers to see if they are equal uses the "==" operator, while a string comparison uses the "eq" operator. If you use the "==" to compare two strings, "Bob" will be seen as equivalent to "Debbie"!

Here's another case where it can get confusing: Comparing two test conditions can be done with the "||" operator when you mean "A is true" or "B is true". However, in matching operations A|B means you're looking for a match on either A or B. When in doubt, check the manual! Look up the operator you're using and see if it's really the one you meant to use.

Anatomy of a simple Perl script

Here is a simple example that illustrates how a Perl script can be built to accomplish a routine task. This script will examine a list of email addresses and remove all the duplicates.

Except for the first line, any line beginning with a "#" is a comment, which explains what the code does. This script takes an input file which contains a list of email addresses and removes any duplicates. Question: How long would it take you to do this job by hand if there were several thousand addresses?

```
#!/usr/bin/perl
# script name = rem_dupe.pl

# The variable $filename is declared and assigned the
# contents of $ARGV[0]. $ARGV[0] is the first element
# in a list of all the "arguments" (filenames, options,
# etc.) on the command line where the script is called.

$filename = $ARGV[0];

# the input file is read into an array
@input_list = ;

# all newline characters are removed
chop (@input_list);

# the list is sorted alphabeticvally
@input_list = sort (@input_list);

# empty the original file and open it for writing
open (OUTFILE, ">$filename");

# get the first address and save to the output
$last_addr = $input_list[0];
print OUTFILE ("$last_addr\n");

# set the counter to 1
$j = 1;

# for each remaining address, save it to the
# output file only if it's unique
foreach $i (1 .. $#input_list) {
  $next_addr = $input_list[$i];
  if ($next_addr !~ /$last_addr/) {
    print OUTFILE ("$next_addr\n");
    $j++; # count the saved address
  }
  $last_addr = $next_addr;
}
close(OUTFILE);
print ("Finished processing $filename\n");
print ("Saved $j unique addresses\n");
exit(0);
```

At this point, the original file contains only unique email addresses. This script can be saved as rem_dupe.pl and run (in an MSDOS window) by typing:

perl rem_dupe.pl [filename] [Return]

Note: If you're running Windows 2000/NT, you won't see MSDOS anywhere on the Start menu(s). Instead, click Start −> Run and type "cmd" (without the quotes) and click OK. This window will act just like the MSDOS windows found in Win95 and Win 98.

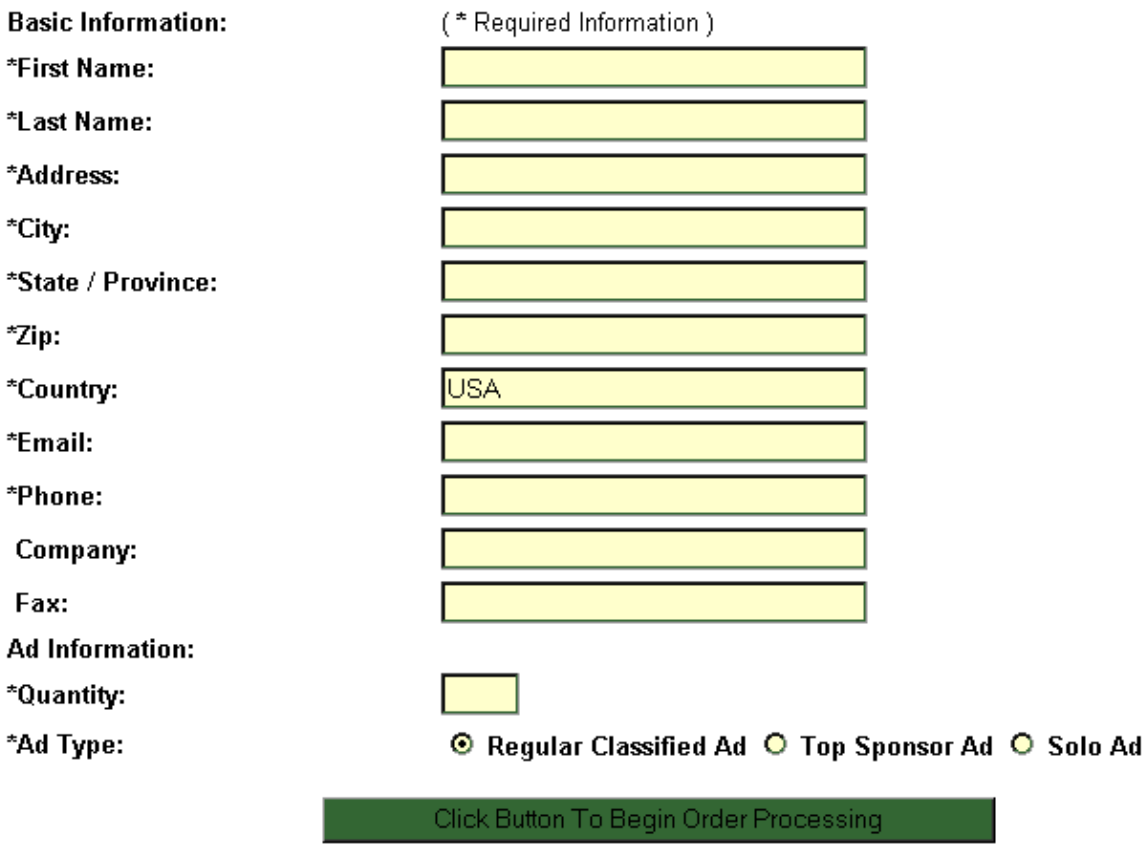
Chapter 8: Working With Forms

Designing forms in HTML

I can't possibly cover all of this here. It's a big subject. What I will do is show you a fairly typical form and how to use a script to extract the data the visitor typed into it, plus any hidden data. Fair enough? I thought so.

Here's the form I used to allow people to purchase advertising in my newsletter. You are free to copy, modify and use this form any way you would like. I've removed all of the decorative elements. Your focus should be on the "active" parts of the form, not on "window dressing". Make no mistake − the appearance of your form page will play a large role in how well it succeeds in its purpose (getting orders, etc.). Doing a good job of making it attractive and successful is up to you.

EXAMPLE ORDER FORM:



After you press the button, the computer will pause as it processes your order. Please do not hit the button twice, as it is normal for the computer to pause for a few seconds as it processes your order.

Here's the HTML code that creates the form:

```
<p align="center"><br>
<H2>EXAMPLE ORDER FORM:</H2><br>
<FORM method="POST"
action="http://www.mysite.com/cgi−bin/ad_proc.pl">
<input type="hidden" name=".required_data"
value="Adtype::Quantity1::Name1::Name2::Email::Phone::Address::City::State::Zipco
<table border="0" width="100%">
<tr>
<td width="250"><b><font face="Arial"> Basic
Information:</font></b></td>
<td width="350"><font face="Arial">( <b>*</b> Required Information
)</font></td>
</tr>
<tr>
<td width="250"><font face="Arial"><b>*First Name: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Name1"
size="30" maxlength="15"></font></td>
</tr>
<tr>
<td width="250">
<font face="Arial"><b>*<Last Name: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Name2"
size="30" maxlength="20"></font></td>
</tr>
<tr>
<td width="250"><font face="Arial"><b>*Address: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Address"
size="30" maxlength="60"></font></td>
</tr>
<tr>
<td width="250"><font face="Arial"><b>*<City: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="City"
size="30" maxlength="40"></font></td>
</tr>
<tr>
<td width="250"><font face="Arial" ><b>*<State / Province: </b></font>
</td><td width="350"><font face="Arial"><input type="text" name="State"
size="30" maxlength="20"></font></td>
</tr>
<tr>
<td width="250"><font face="Arial"><b>*Zip: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Zipcode"
size="30" maxlength="10"></font></td>
```

```
</tr>
<tr>
<td width="250"><font face="Arial"><b>*Country: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Country"
size="30" value="USA" maxlength="60">
</font></td>
</tr>
<tr>
<td width="250"><font face="Arial"><b>*Email: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Email"
size="30" maxlength="48"></font></td>
</tr>
<tr>
<td width="250">
<font face="Arial"><b>*Phone: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Phone"
size="30" maxlength="20"></font></td>
</tr>
<tr>
<td width="250">
<font face="Arial"><b> Company: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Company"
size="30" maxlength="60"></font></td>
</tr>
<tr>
<td width="250">
<font face="Arial"><b> Fax: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Fax"
size="30" maxlength="20"></font></td>
</tr>
<tr>
<td width="250"><b><font face="Arial">Ad Information:</font></b></td>
<td width="350"> </td>
</tr>
<tr>
<td width="250"><font face="Arial"><b>*Quantity: </b></font></td>
<td width="350"><font face="Arial"><input type="text" name="Quantity1"
size="4"></font></td>
</tr>
<tr>
<td width="250"><font face="Arial"><b>*Ad Type:</b></font></td>
<td width="500"><font face="Arial"><b>
<input type="radio" name="Adtype" value="Regular" checked> Regular
Classified Ad
```

<input type="radio" name="Adtype" value="Sponsor"> Top Sponsor Ad
<input type="radio" name="Adtype" value="Solo"> Solo
Ad</td>
</tr>
</table>
<table border="0" width="90%">
<tr><td width="100%" height="50">
<input type="submit" value="Click Button To Begin Order Processing">
</td></tr>
<tr><td>After you press the button, the computer will pause as it processes your order. Please do not hit the button twice, as it is normal for the computer to pause for a few seconds as it processes your order.

</td></tr>
<tr><td align="left">Note: Due to intermittent server problems, you may sometimes get an "Error 500" screen. If this happens, simply press your browser's "BACK" button. You'll return here with all your information intact. Then click the button again to submit your order.

</td>
</tr></table>
</FORM>

Look at the very first line of the FORM. The first thing you should notice is that the form uses the "POST" method. There are two methods available, GET and POST. Don't use GET; it can cause a whole bunch of security problems. The second thing to observe is the action= part. This indicates what will happen when the "Submit" button is clicked.

To run a script when the form is submitted, you'll put the path to the script here. Most times you can use a relative path, such as /cgi−bin/my_script.pl, but sometimes the path must be absolute (complete and unambiguous). I just ignore the issue and use a complete path every time. However, if you move the file to a new server or domain (or to a different directory on the same server) you'll have to change any absolute paths.

Note: for HTML files, use a relative path whenever possible. It's a real time−saver if you move things around on your site.

Asking for user input

Now, look for any lines beginning with the word "input". In each of these, there's a "type" given. Inputs of type text display a one−line text box used for items like name or email address. Inputs of the radio button or checkbox types allow selection of one or more items from a limited set of choices. Another input type, not shown in this example, is "select", which produces a drop−down list. See an HTML reference book or tutorial for instructions on using this common input.

Another common input type is TEXTAREA; it won't have the word "input" inside the tag, but it is an input anyway. Inputs of this type display a box where you can type in several lines of text. You can specify the width and height of the box, too.

These inputs are all things the user can type in or select. There are other inputs, of type hidden, which the user normally doesn't know about. They are not really well hidden. They just don't display as part of the Web page. Click on View Source to see the code of a Web page and they are perfectly visible.

Sending the data somewhere

A script can catch all these inputs and also the CGI variables that pass between client and server as part of handling the form. Notice that each input has a name given in the form code. These names are what your script will look for among all the data passed to it by the CGI process.

What you should have learned from this chapter is that you can collect data items with a form, and send them to a script to be processed in some way. You'll have to give each item a name, so you might as well make the name meaningful. In the form you just looked at, "Email" contains the user's email address, "Name1" contains their first name, etc. These names are referred to as "keys" in Perl. During processing by the script, they will end up as key/value pairs. The key will become a variable, and be assigned the value that the key picked up from the form.

The keys used in this form are Name1, Name2, Address, City, State, Zipcode, Country, Email, Phone, Company, Fax, Quantity1 and Adtype.

Chapter 9: Processing Input

One good way − ReadParse (from cgi−lib.pl)

The ReadParse function is part of Stephen Brenner's incredibly useful Perl library called cgi−lib.pl, which is documented here:

http://cgi−lib.berkeley.edu/

You'll find the cgi−lib.pl library in its entirety as a separate file included with this book. The basic idea is to write your script and include this line near the top of the file:

require "cgi−lib.pl";

This tells the system that all the functions of the library will be available to the current script.

The code that follows is based on an older version of ReadParse. I urge you to use it as merely an illustration. It is easy to read and implement, but the newer version found in cgi−lib.pl is greatly improved. It does, however, give you plenty of things to look up in your Perl manual. That will be a good learning exercise for those of you who are new to Perl. We will refer back to a few of the techniques used by this code later in this book.

```
sub ReadParse {
   # @_ is a system variable if it has any non−zero value,
   # assign that value to a pointer to the string $in
   local (*in) = @_ if @_;
   local ($i, $key, $val);
   if ( $ENV{'REQUEST_METHOD'} eq "GET" ) {
   # don't accept method=GET; just quit processing
      print "Content−type: text/html\n\n";
      print "Sorry, this script only accepts METHOD=POST. ";
      exit;
   }
   # otherwise, read all the form input into $in
   else ($ENV{'REQUEST_METHOD'} eq "POST") {
      read(STDIN,$in,$ENV{'CONTENT_LENGTH'});
   }
   # $in is a string, with the key/value pairs separated by
   # ampersands. Split this sting at each ampersand (&)and put
   # the pieces into the array @in
   @in = split(/&/,$in);
   foreach $i (0 .. $#in) {
      # all the key/value pairs passed to the script are
      # handled, one at a time. Each value is assigned
      # to a local variable ($Quantity1, etc.)
      # Convert plus's to spaces
      $in[$i] =~ s/\+/ /g;
      # Split into key and value.
      ($key, $val) = split(/=/,$in[$i],2);
```

```
# splits on the first =.
# Convert %XX from hex numbers to alphanumeric
$key =~ s/%(..)/pack("c",hex($1))/ge;
$val =~ s/%(..)/pack("c",hex($1))/ge;
# if the "key" is Quantity1, then the value is
# assigned to $Quantity1. Repeat for each "key"
if ($key eq "Quantity1") {
    $Quantity1 = $val;
    $Quantity1 =~ s/;.*//;
    $Quantity1 =~ s/,.*//;
    $Quantity1 =~ tr/0−9/\b/cs;
}
if ($key eq "Name1") {
    $Name1 = $val;
    $Name1 =~ s/;.*//;
    $Name1 =~ s/,.*//;
    $Name1 =~ tr/[A−Z][a−z]/\b/cs;
}
if ($key eq "Name2") {
    $Name2 = $val;
    $Name2 =~ s/;.*//;
    $Name2 =~ s/,.*//;
    $Name2 =~ tr/[A−Z][a−z]/\b/cs;
}
if ($key eq "Email") {
    $Email = $val;
    $Email =~ s/;.*//;
    $Email =~ s/,.*//;
    if ($Email !~
        /^[\w\d][\w\d\,\.\−]*\@([\w\d\−]+\.)+
    ([a−zA−Z]{3}|[a−zA−Z]{2})$/) {
        $Email = "";
    }
}
if ($key eq "Phone") {
    $Phone = $val;
    $Phone =~ tr/\(\)\−[0−9]/\b/cs;
}
if ($key eq "Address") {
    $Address = $val;
    $Address =~ s/;.*//;
    $Address =~ s/,.*//;
    $Address =~ tr/[A−Z][a−z][0−9]\t \./\b/cs;
}
if ($key eq "City") {
    $City = $val;
    $City =~ s/;.*//;
    $City =~ s/,.*//;
    $City =~ tr/[A−Z][a−z]/\b/cs;
}
if ($key eq "State") {
    $State = $val;
    $State =~ s/;.*//;
    $State =~ s/,.*//;
    $State =~ tr/[A−Z][a−z]/\b/cs;
}
if ($key eq "Zipcode") {
    $Zipcode = $val;
    $Zipcode =~ s/;.*//;
    $Zipcode =~ s/,.*//;
    $Zipcode =~ tr/[0−9]\−/\b/cs;
}
if ($key eq "Country") {
    $Country = $val;
```

```
            $Country =~ s/;.*//;
            $Country =~ s/,.*//;
            $Country =~ tr/[A−Z][a−z]/\b/cs;
        }
        if ($key eq "Company") {
            $Company = $val;
        }
        if ($key eq "Fax") {
            $Fax = $val;
            $Fax =~ tr/\(\)\−[0−9]/\b/cs;
        }
        if ($key eq "Adtype") {
            $Adtype = $val;
            # Determine the ad price based on the ad type
            if ($Adtype eq "Regular") {
                $Describe = "Classified Ad(s)";
                $Unit_Price = 5.00;
            }
            if ($Adtype eq "Sponsor") {
                $Describe = "Top Sponsor Ad(s)";
                $Unit_Price = 10.00;
            }
            if ($Adtype eq "Solo") {
                $Describe = "Solo Ad(s)";
                $Unit_Price = 15.00;
            }
        }
        # Voila' − Perl can do math!
        $Subtotal = $Quantity1 * $Unit_Price;
        $Total    = $Subtotal;
        # tax calculations not shown; rates vary widely.
        # So, if you need to figure sales tax, create a
        # subroutine to work it out and call it here.
        $Total    = $Total + $Tax;
        if ($Subtotal !~ /\./) {
            $Subtotal .= ".00";
        }
        if ($Subtotal !~/\.[0−9][0−9]/) {
            $Subtotal .= "0";
        }
        if ($Tax !~ /\./) {
            $Tax .= ".00";
        }
        if ($Tax !~/\.[0−9][0−9]/) {
            $Tax .= "0";
        }
      # These two if−statements ensure that the price
      # will contain two digits to the right of the
      # decimal point, even if they are zeroes. Perl
      # tends to drop the ".00" from a whole number.
        if ($Total !~ /\./) {
            $Total .= ".00";
        }
        if ($Total !~/\.[0−9][0−9]/) {
            $Total .= "0";
        }
        # Associate key and value
        # \0 is the multiple separator
        $in{$key} .= "\0" if (defined($in{$key}));
        $in{$key} .= $val;
    }
    return length($in);
}
```

In each "input" line is a type field and a name field. The name field's value is the name this particular input will have when your script catches it. It's caught as a name/value or key/value pair, where the name (or key) acts like a program variable and the value is whatever the user chose to enter in that field. The same is true for "select" and "textarea".

ReadParse does the work of separating the input data into a set of key/value pairs, then assigns the values associated with the keys to local variables.

A slightly different way − using CGI.pm

While the method described in the last section works well, there is another way that you may find simpler and easier. The CGI.pm module contains a lot of functions that shortcut your programming efforts. To use this module, simply make sure that it has been installed on your server. Contact your host's technical support and ask them if it's already there; if it isn't, get them to install it for you. You probably can't do this yourself, since you normally don't have access to the directory where it must be placed.

Once you have this module available, it's a snap to use it. Include these lines near the top of your script:

```
use CGI;
$query = new CGI;
```

The first line allows you to use anything in the CGI.pm module. For those of you who have any C++ experience, this is like using "#INCLUDE" to make the header file of a class visible to your program. Refer to the documentation for the module at:

http://stein.cshl.org/WWW/software/CGI/cgi_docs.html

for more ways to use this tool.

Tip: CGI.pm can make processing your forms easy, using just a few lines of Perl.

The next line creates a CGI "object". This object lets you grab data items that were passed to your script (from your form), using the names you gave them in the form. It does this bit of magic by looking at the "environment" for a "query string", parses the contents and stores everything it found. To get the results into local variables, you can now use lines like this:

```
$Name1 = $query−>param('Name1');
$Name2 = $query−>param('Name2');
$Address = $query−>param('Address');
$City = $query−>param('City');
...etc.
```

Make as many lines like this as you need, one for each data item you need to capture from the form. Once the items are local variables in your script, you can do whatever you like with them.

Chapter 10: Query Strings

What is a query string?

First, you need to understand a little about CGI itself. It's a set of variables, not a language. These variables make up an "environment". They have names like QUERY_STRING, REQUEST_METHOD, REMOTE_HOST, etc. They are all passed between client and server during processing of a CGI program. Data can be extracted from them that tells you all sorts of things about the connections, the type and length of content being passed, and so on. The $QUERY_STRING variable holds all the inputs that your form deliberately sent from the client (their web browser) to the server (computer where the script is running).

Tracking traffic with a query string

Have you ever noticed a bunch of characters beginning with a "?" following the URL of a Web page? That's a query string, too. This allows tracking "hits" on the page. The server logs will tell you how many times the URL with a particular query string was requested. By using different query strings in your ads, you can find out which ads were more effective at getting click−throughs to the page!

Controlling a process

Query strings can also be used to influence the behavior of your script. For instance, the script that generates my Ezine Directory, can be called from the page it displays. When a user selects a category from the menu, the script is called again, but this time it "knows" to display only the ezines in the selected category. This is done by reading the items in the query string and looking for "data=xxxxxx".

The value of the data parameter tells the script which file to read. Then, the info in that file is used to draw the new page. Here's how it's done:

```
# initialize the database

$database = ".top20";

#define the page the user will see after submitting the form
$return_path="http://www.example.com/this_page.html";

if ($ENV{'QUERY_STRING'} ne "") {
    $qs = $ENV{'QUERY_STRING'};
    @qs = split(/&/,$qs);
    foreach $i (0 .. $#qs) {
        $qs[$i] =~ s/\+/ /g;
        $qs =~ s/%(..)/pacl("c",hex($1))/ge;
        ($key, $val) = split(/=/,$qs[$i],2);
        $qs{$key} = $val;
        if ($key eq "referrer") {
            $return_path = $val;
        }
        if ($key eq "data") {
           $database = $val;
        }
    }
```

First, the main script runs and displays a "default" group of ezines. The page includes a drop−down menu at the top and bottom, where you can select a new category. When you click the Go! button after choosing a category, a script called decode.pl is run. Its only inputs are the URL of the return_path page and the database file to use next. The return_path page address is where the user will be sent after using the Quick−Subscribe form. The database file contains the entries for the ezines in the selected category.

Decode.pl preserves the return_path address because it's needed by the main script. Here's the decode.pl script:

```
#!/usr/bin/perl
# parse the form data
&SetCategory;
select (STDOUT);
$| = 1;
select (STDERR);
$| = 1;
print ("Location:
http://www.example.com/cgi−shl/subunsub2.pl
?referrer=$return_path;
exit(0);


sub SetCategory {
    local (*in) = @_ if @_;
    local ($i, $key, $val);
    if ($ENV{'REQUEST_METHOD'} eq "POST") {
        read(STDIN,$in,$ENV{'CONTENT_LENGTH'});
    }
    else {
        exit;
    }
    @in = split(/&/,$in);
    foreach $i (0 .. $#in) {
        # Convert plus's to spaces

        $in[$i] =~ s/\+/ /g;
        # Split into key and value.
        ($key, $val) = split(/=/,$in[$i],2);
        # splits on the first =.
        # Convert %XX from hex numbers to alphanumeric
        $key =~ s/%(..)/pack("c",hex($1))/ge;
        $val =~ s/%(..)/pack("c",hex($1))/ge;
        if ($key eq "return_path") {
            $return_path = $val;
        }
        if ($key eq "Data") {
            $data = $val;
        }
    }
}
```

The script then calls the main script like this:

```
print ("Location:
http://www.merrymonk.com/cgi−shl/subunsub2.pl
?referrer=$return_path;
```

The query string is "?referrer=$return_path&data=$data". The two linefeed characters (\n\n) are needed to insert a blank line between the header and the data. Otherwise, the server will get confused and give you an "Error 500".

A round trip ticket

Now we're back at the main script. This script can extract the "data=xxxxx" and save its value in a local variable, $database. Based on the value of $database, the main script selects the appropriate page title. The title indicates what category of newsletters are being displayed.

Now the main script runs again, but this time the default value of $database is replaced by the value imported from a query string. The logic here is that if the value of the "data" is empty, the script will use its default value, which was initialized at the top of the script. If it has a non−emtpy value, then $database becomes that value. Now the script uses the information in the file named by $database to generate the list of ezine entries for the page.

There is a second, outer round−trip performed because we have a variable in the query string that "remembers" where the script was originally called. Someone placed a link on his/her Web page that included this key/value pair in the query string:

referrer="http://www.example.com/this_page.html"

Then all the other work of the script is finished. It substitutes the value of referrer (which was captured in the local variable $return_path), into this line:

print "Location: $FORM{'return_path'}\n\n";

Because of this, the script is able to load the specified page into the user's browser and complete its round trip. The real beauty of this is that to the visitor, it appears that the Quick−Subscribe form is on is part of the site where they followed the link. To the site owner, the advantage is that the visitor is returned to the owner's site without realizing that he/she even left.

Finally, you can use the second method from Chapter 9 to make the code much shorter. You should be able to do this easily after reading the previous chapter.

Chapter 11: Security Issues

Keeping it private

If your site/server allows anonymous ftp, anyone with an ftp client can see, download and modify your scripts. This is a dangerous situation! You should make sure that accessing your site via ftp requires a valid username and password. Protecting ftp access is only one of a number of security measures you should take. However, it is the first step in keeping your vital scripts safe from hackers and others who don't have your best interests at heart.

Restricting access

One thing to consider is preventing your scripts from being run on any sites that are not under your control. This can be done by checking the environment variables to determine where the request (to run the script) originated. Then, you can force the script to exit without taking any action if it was being called from some other server. This is not a perfect solution, since some people are capable of making it appear that the request *did* come from your server when in fact it did not. This hacking technique is known as "spoofing" the IP address.

Here's an easy way to implement this security precaution:

```
# List the acceptable URLs
@referrers = ('www.example.com','xxx.xxx.xxx.xxx');
# Check Referring URL

# subroutine to verify the referring URL
sub check_url {
    # Localize the check_referrer flag which determines
    # whether the user is valid.
    local($check_referrer) = 0;
    # If a referring URL was specified, for each valid
    # referrer, make sure that a valid referring URL
    # was passed into your script.
    if ($ENV{'HTTP_REFERRER'}) {
        foreach $referrer (@referrers) {
            if ($ENV{'HTTP_REFERRER'} =~
                m|https?://([^/]*)$referrer|i) {
                $check_referrer = 1;
                last;
            }
        }
    }
    else {
        $check_referrer = 1;
    }
    # If the HTTP_REFERRER was invalid, call an
    # error−handling routine.
    if ($check_referrer != 1) {  }
}
```

The subroutine "error_url" could return an error page (or not); in either case, it should end with an "exit" statement so your script stops running at once.

Why worry about the inputs?

When people are allowed to enter their information in a form, it's possible for them to enter things you don't want. Sometimes it's just a typo; other times it's a deliberate attempt to cause you harm. When this happens, your script may not respond correctly to the input. In severe cases, a system command or a Java applet or script may be unleashed on your server, causing untold damage. But don't panic! It's fairly easy to prevent the vast majority of this activity by taking a few simple precautions.

First, limit the maximum length of an input string to no more than it needs to be. Do this by using the "maxlength=" attribute in a text input. Make the maximum length big enough to handle the input text you expect to get, but nomore.

For instance, 48 characters is *usually* plenty of room for an email address. For a Zip Code, you don't need more than 10 characters (9 digits plus a −).

Next, filter out any characters that should not be in the field. The lines for filtering should execute immediately after the value is assigned to the variable, so there's no time for spurious characters to be acted on by your code.

Tip: If you've put any filtering in place, be sure to test your form thoroughly. Make sure that any characters that could belong in the field are not rejected. This is a tricky area. You'll have to strike a balance between rejecting attempts to harm your system and being too restrictive about input. Be especially careful with names and street addresses!

Filtering string data

Here's where Perl really shines! It makes it easy to replace unwanted characters with "nothing", translate upper−case to lower case, etc. A system command could be passed in along with the text, by simply putting a "**;**" after the text, then the command.

To get rid of some characters, use this form: First the variable containing the string. Then, the "match" operator (**=~**) folowed by the substitution operator (**s**). There are three forward−slashes after the substitution operator. Put the stuff to be removed between the first and second forward slashes. They will be replaced with whatever is between the second and third forward slashes. If there is nothing between the last two slashes, the "to−be−replaced" characters are simply removed.

Caution: Unix commands can be very short. To delete all the files in the current directory, the Unix command is **rm *** . Even a little leftover space could be enough room for something destructive. You don't expect a "**;**" to be part of a name, so reject it and whatever follows it. In the following code, the " **.*** " means "everything after this character".

Anything after (and including) a semicolon or a comma should usually be removed. Here, the variable $Name1 represents the person's first name. So, you need not accept anything but letters − no other symbols, spaces, etc., should be here. The last line will translate anything that is not a letter (A−Z or a−z) to a backspace.

When you use the translation operator (**tr**), the behavior is a little different. The "**\b**" is the backspace character, the **c** option on the end means "everything that doesn't match the pattern" and the **s** option means "replace the pattern with a single character". Without the **s** option, you'd get multiple backspaces as a result of the translation. If you allow entry of the full name, account for the fact that there will be spaces (between first and last name, for example), possibly some periods (examples: Dr., Mr.) and even a comma (ex: Joe Smith, Jr.).

Note: If the form results come to you by email, a backspace will often look like a little square.

```
$Name1 =~ s/;.*//;
$Name1 =~ tr/[A−Z][a−z]/\b/cs;
```

If you allow entry of the full nameCity, Country, State, etc., can be filtered in much the same way, but you can allow the space characters.

Filtering numeric data

As before, we don't want any scripts, commands or other "junk" coming in through a numeric item, so we use the first two filters and add a third to remove any letters in this item (which might be a Zip Code, for example):

```
$Zipcode =~ s/;.*//;
$Zipcode =~ s/,.*//;
$Zipcode =~ tr/[0−9]\−/\b/cs;
```

To filter a phone or fax number, you could use:

```
$Phone =~ s/;.*//;
$Phone =~ s/,.*//;
$Phone =~ tr/\(\)\−[0−9] /\b/cs;
```

Filtering street addresses

Here we have to accept numbers, letters, spaces and possibly a period. We still reject the commas, semicolons, angle brackets (they could contain a JavaScript) and most other symbols. So here's a filter for them:

```
$Address =~ s/;.*//;
$Address =~ s/,.*//;
$Address =~ tr/[A−Z][a−z][0−9]\t \./\b/cs;
```

Filtering email addresses

Email addresses can contain a lot of characters you don't usually see in other strings, but they do follow a well−defined pattern. Using what is known about them in general, it's possible to make a filter that will reject anything that simply can't be an email address. Here's one that many people use:

```
$Email =~ s/;.*//;
$Email =~ s/,.*//;
if ($Email !~
/^[\w\d][\w\d\,\.\−]*\@([\w\d\−]+\.)
+([a−zA−Z]{3}|[a−zA−Z]{2})$/) {
    $Email = "";
}
```

When an email address is not formatted correctly, the filter replaces it with an empty string. The following line will make the script call an error handler if the email address fails to pass the test (because it's an empty string):

```
if ($Email eq "") {
    }
```

There are millions and millions of possible combinations of letters and symbols that fit this pattern, but which are not legitimate addresses. Sometimes the domain does not exist. Other times the username is bogus. The account may have existed at one time but is now closed or inactive. Still others may be addresses you want to block for some reason. In Appendix C, I'll show you a filter I use to get rid of email addresses I don't want to process and in Chapter 13 I'll explain how it was developed.

Test your filters

Once you've chosen and implemented the filters, you need to test them. Bring up the page with the form and enter some sample data. Then check to see if you captured exactly what you expected. To speed up this process, it's a good idea to have the script display a page with the user's data. You can remove the code that produces this page after you're finished testing if you don't need it any more.

For each input, try everything that it should allow and make sure it works. Then try everything it should reject and make sure nothing unwanted can get past the filters. Once you're 100% satisfied with the filters, then move on to the questions of what you'll do with the data. I really mean it. Be as certain as you can be that your data is going to be exactly what you expect before you start writing code to process it!

More security ideas

For a more thorough guide to security problems and their solutions, visit this site:

http://www.w3.org/Security/Faq/www−security−faq.html

Chapter 12: Using Subroutines

What IS a subroutine, anyway?

In Perl, as in many other languages, a subroutine is a separate block of code designed to do a particular job. When we want to do that job somewhere in a script, we "invoke" the subroutine by using its name, preceded by an ampersand (&) character. You can do this anywhere in the script, even inside another subroutine. So, subroutines can be "nested". Subroutines are most often placed after the end of the main body of the script. In this way, they are made available to any code in the main body that needs them.

Why use subroutines?

There are two main arguments for using them:

They break your program into smaller chunks and make it easier to read. When you look at the main body and see a subroutine being called, you can scroll down to where the subroutine is and see what happens there. You can then go back to where it was called and see what happens next.

They let you use one block of code to do the same task many times. This prevents having to write the same instructions over and over. Not only that, but you can take a subroutine that performed well in one program and use it an another program where you need to do the same job.

Where do I get them?

As I just said, you can recycle them from other programs you've written. You can copy and paste them from other programs you've obtained elswhere. You'll find some of my favorites included with this book. See Appendix C for a list of these. Then visit some of the sites listed in Chapter 4. Download some of the free scripts that sound interesting or useful. You're bound to find some cool subroutines. They may be usable exactly as written, or you may need to make minor adjustments. Anyway, there's a nearly endless supply of them!

How do I use them?

It doesn't get much easier than this! Put them at the bottom of the script and then call them whenever you need to perform the tasks they do. A subroutine is called by a line that looks like this:

&write_data;

Or this:

$result = &write_data;

where "write_data" is the name of the subroutine. This was covered in Chapter 7 in a little more detail. If you skipped that chapter, please go back and read it now. Otherwise, just check out the subroutines that come with the book. There's a brief description of each one in Appendix C. Also, read the comments within the subroutines themselves. This can give you more insight into how to use each one.

Note: Sometimes you'll use the result (return value) of a subroutine in a if−statement. Most subroutines have a "return" statement at the end. It usually takes the form:

return $variable; or return;

Parentheses around the value to be returned are optional, and you can use empty ones if nothing will be returned. The value, if any, returned is the final "result" of running the subroutine. Your code might call another subroutine if the result of the call to the first subroutine was (or was not) zero. For example:

if (&bad_data) {&ShowError;}

Tip: Always put a return statement in each subroutine, even if you don't plan to use it. You may find a use for it later. If you do, a simple text search for the word "return" will take you to the return statement, where you can change or add the return value.

Chapter 13: Getting Subscribers

List server basics

A list server is a software program that allows you to add/drop subscribers manually, automatically or both. It allows you to send a message to a certain address and then mails copies of the message to everyone on your list. It may have other functions that vary from one program to another.

Some list servers are operated by list hosting companies such as Topica and Yahoo! Groups. When you host your list with them, whatever feature set they provide becomes available to you. Other list servers are operated on your own PC, Mac, etc. Some of these use your ISP to send out all the mail. This can introduce a few problems such as speed, reliability and the occasional complaint from your ISP.

Still others exist in the form of software, such as Lyris, that you buy and install onto a dedicated server or that your host has installed on their servers and lets you use. The mailing list feature of the iMail server on an NT host is an example of the last type of list server.

Note: If your host provides a mailing list feature, be sure to find out if there are any limits on how many subscribers can be on a list or how many messages you can send per day, per month, etc.

Tip: If their answer is "unlimited", you'll save some money. If there are limits, and they're too low...you'll eventually need to find (and probably pay for) professional list hosting. Consider this a cost of doing business.

Regardless of the type and features of your list server, your goal is to use it to keep in touch with a growing list of people via email. This can take the form of a discussion list, an update list or a regularly scheduled newsletter. In all cases, you want to keep adding new subscribers. So, how do you do that?

Manual subscription

Some people manage their lists by hand. In this case, a subscription request may be a short message to one of their email addresses with something like "subscribe" or "join" in the subject or body of the message.

An unsubscribe request would be similar, but might contain "remove", "leave" or "unsubscribe". The name of the list may be part of the message. The list owner uses some type of program to maintain the list, which allows adding or removing email addresses. I personally don't care for this method; it seems too much like work.

Automatic subscription

Most list server software provides you with a "subscribe" and an "unsubscribe" address. Any person can send a (usually blank) message to one of the addresses and be automatically added to, or removed from, your list. This may be used on a Web site in several ways: Post a mailto: link for people to click on, add a form, use a pop−up window, etc. Most of the time the Web site mechanism is designed only for subscribing, with the unsubscribe process built into the outgoing messages. Sometimes it's a link; sometimes it's an address to copy and paste. Other times, replying to the message will get you unsubscribed.

Any time you have people subscribing from your Web site, you have opportunities to use CGI scripts. For example, clicking on a link could trigger a script which sends out the current issue immediately. Or perhaps you are going to deliver a message to new subscribers telling them where to pick up a thank you gift. Notice that giving people a tangible reason to subscribe will often help increase sign−ups. Perhaps you'd like to collect some information about the person as the time they sign up. A script could save the results of a brief survey you asked them to fill out. The results could be stored in a database for analysis. The more you know about your subscribers, the better equipped you are to offer them what they want.

Tracking subscribers

Continuing the theme of automated subscriptions and CGI, here's a thought: Your script can capture the domain from which the request came and log that in your database. Many other types of data can be captured, with or without asking the new subscriber to supply it. It's also quite simple to determine which ad brought in a new subscriber, if you code the email address and check the code when they join. This is often suggested as a manual operation, but it can be scripted. An email address link can have a query string including the subject and/or body of a message. Counting the number of subscriptions that came in with a specific character string shows you how effective the ad was.

Why not script the process and log the responses to several different ads? That way, you can test and refine the ads until they get the best response possible. Once you've determined which ads work, use them over and over. Continue to monitor their response rates in case the rates change. Save money by only paying to run ads that you already know will produce good results.

Dealing with spammers: advanced filtering

Ours is not a perfect world and people do things that will annoy you. One such irritation is spamming. If you can identify certain domains where a high percentage of spam originates, why not block all email addresses in that domain? I do. I compare the email addresses that people supply to a list of domains that are mostly trouble. If an address matches one of these, the person is told that the email address is "not allowed" or "unacceptable".

Perl's string−comparison functions make this a piece of cake! See Appendix C for a subroutine called "verify_email" that will reject a lot of "bad" email addresses. Once you look at it, it will be easy to see how you can add more domains to block. Drop this bit of code into a script that is handling subscriptions and the addresses of millions of spammers and autoresponders are denied access to your list, FFA page, etc.

I developed part of this code by searching for autoresponders in a search engine and looking at how the addresses were constructed. In most cases, all of the addresses were in the domain of the provider. For example, an address in the domain "getresponse.com" is nearly always an autoresponder.

Tip: Sending any message to an autoresponder may result in *you* getting unwanted email. Never let this happen unless you want it to. Since you may be hitting one that sends multiple, time−delayed responses, you might have to take time to unsubscribe − again and again. Just don't accept these addresses in the first place!

Chapter 14: Sending Email

Using sendmail (Unix systems)

Cardinal Rule #2: Using sendmail **requires** the −t switch. You can include it when you save the path to sendmail as a local variable like this:

$mailprog = "/bin/sendmail" . "−t";

Or you can include it in the command line where it's being used, as in the subroutine below. There, the contents of "MAIL" are being fed ("piped") as an input to the sendmail command and the −t switch is added to the command line as an "argument" or option.

Most Web hosts are using the Unix or Linux operating system. They will, with few exceptions, have the sendmail program. You can use sendmail to send an email message from within your script. It's easy. Here's an example:

```
# $mailprog contains the location of the sendmail program
$mailprog = "/bin/sendmail";
# the email address must already be stored in the
# variable $Recipient
sub send_mail {
    # Open The Mail Program
    open(MAIL,"|$mailprog −t");
    print MAIL "To: $Recipient\n";
    print MAIL "From: Bob Jackson \<noreply@example.com\>\n";
    print MAIL "Subject: Thanks!\n\n";
    print MAIL "Hi $Name,\n\n";
    print MAIL "Thanks for visiting my Web site!"\n";
    print MAIL "Sincerely,\n";
    print MAIL "Bob Jackson, Webmaster\n\n";
    close (MAIL); # this sends the message
}
```

Sendmail treats the mail message being constructed as a temporary file. You can "print" some saved text from a real file into the message body. By doing this, you are able to update the outgoing message by simply replacing the file. That's easier than editing the script. Or you can add, remove or change the 'print MAIL "...\n";' statements in the script. The "\n" means drop to the next line. Two of them (\n\n) will give you a blank line. That's used to separate paragraphs, etc. In some scripts you'll see several lines "concatenated" (joined) by the dot (.) operator. This can be done to avoid having to put "print MAIL" in front of every line. Don't use the dot on the very last line. The script would expect more text, find none and then crash.

Using qmail

Some systems will have qmail insead of sendmail. It behaves basically the same as sendmail − with one major exception: Don't use the −t switch with qmail!

Using iMail (NT systems)

Most, or at least many, Windows NT servers have the iMail program installed. This feature−rich program is sold by:

http://www.ipswitch.com

If your host uses iMail, I strongly suggest you visit their site and download the manual. It's in PDF format, so you should have no trouble reading it.

The manual is geared towards the system administrators who purchase and install the program on their servers, so a lot of what you see will not apply to you as a user. However, a lot of it will be very useful to in getting the most out of this mail program. Besides, it plays by a different set of rules than sendmail, and it's handy to have the manual so you'll understand how it works. You're not likely to find a book on this one in a library or bookstore!

Unlike sendmail, iMail *must* get the message body from a file. This file need not be persistent, though. It can be created for the message and then deleted. In fact, sometimes it's very important to remove the temporary file(s), as they can waste a lot of disk space if they're not removed. Because it's a Windows NT program, the path will contain backslashes. These must be "escaped" with another backslash so Perl will understand the path. Here's the code:

```
# the message must get its body from a file; a complete
# path to this temporary file must be supplied
$temp_msg = "D:\\users\\example\\htdocs\\email.txt";
# You may need to change the path to imail.
$mailprog = "C:\\IMAIL\\IMAIL1.EXE";
# $email must contain the To: address

sub SendEmail
{
  my $line;
  if (−e "nosend.txt")
  {
    my $emailfound = 0;
    open (FH, "nosend.txt");
    while (($line = <FH>) && ($emailfound == 0))
    {
```

```
    chomp $line;
    if ($line eq $email) { $emailfound = 1; }
  }
  close (FH);

  if ($emailfound != 0) { return 0; }
 }

 open(MAIL,">$temp_msg");
 $subject = "Your site has been added!";
 $owner   = "cul_de_sac\@example\.com\n";
 print MAIL "Learn all types of new skills and much \n"
 print MAIL "more by subscribing to free newsletters \n";
 print MAIL "and discussion lists. It's fast, easy \n";
 print MAIL "and convenient\n\n";
 print MAIL "Visit my directory today:\n";
 print MAIL "http://www\.merrymonk.com/maillist\.html\n";
 print MAIL "\n\n\n";
 print MAIL "You submitted the following link data:\n\n";
 print MAIL "Title: $title\n";
 print MAIL "URL: $url\n";
 print MAIL "Section: $section\n";
 print MAIL "Submitted by: $email\n\n";
 print MAIL $_[0];
 print MAIL "\n\n";

 close (MAIL);
 system("$mailprog −u $owner −f $temp_msg −s $subject
       −t $email");
 unlink $temp_msg;
}
```

The work begins by opening a file path or handle to the temporary message file. It may contain ready−to−use text or not, as you choose. Don't delete it in the last step if you need to re−use it, though! Then the subject is filled in, and the return address is assigned. Next, text is "printed" to the mail file. Closing the mail file doesn't send the message as sendmail does. You use the "system" command to send an explicit command to the mail program. This command is enclosedin double−quotes and parentheses. This is how Perl gives commands to the operating system itself, or to a program supported by the system. The last step is to delete the temporary file, which is usually optional.

You can create a file called nosend.txt if you wish. It can have a different name if you prefer, but make sure the script uses the same name. If an email address is found in nosend.txt, the script will not send mail to it. The $owner variable holds the "From:" address for all outgoing mail. The subject of the message is stored in $subject. Check the manual for rules about the use of double−quotes. Sometimes you have to store them explicitly in a string! Here's how that's done:

```
$email = "joe\@smith.org";
$email = "\"$email\"";
```

The back−slash characters tell Perl to save the double−quotes and the "@" symbol instead of treating them as special characters.

Using autoresponders

Autoresponders are special email addresses designed to deliver messages "on demand". When you send a message to an autoresponder, it replies with a stored message. Some can deliver a number of follow−up messages over several days, weeks, etc.

The autoresponder address may or may not save messages sent to it. Some are totally free, some are "free" but supported by paid advertising sent along with your message and others are paid for on a monthly basis. There are even some that you buy once and then get to them use indefinitely.

Search around the Web and you can find scripts that provide you with unlimited autoresponders for one price. Meanwhile, here's a few of the "free" autoresponders:

- SmartBotPro
- MyReply.com
- GetResponse.com

And here's a couple of paid ones:

- QuickTell
- Aweber

Note: You could use a CGI script to get an email address from a form and generate a message to one or more autoresponders with it. This way, someone could subscribe to your newsletter, receive their first issue and some other special information in just a few seconds. How's that for automation?

Coping with spam

Some of your messages will eventually reach people who decide to send you spam, junk mail, etc. They will usually send it to the "From" or "Reply−To" address in the message header. You probably don't want this mail at all. To avoid this problem, just set up an email account that filters everything to the trash can. Let it delete all incoming mail and you won't have to waste your time reading it. In the mail routines in this chapter, you saw where the "From" address is inserted into the message. Use the address that deletes everything as the "From" address.

It's a good practice to include some sort of disclaimer that advises people not to reply to the message. You may also want to tell them that all incoming mail will be deleted. You can further reinforce this concept by naming the account something like "noreply@example.com".

In the body of the message, you are certainly free to include an address where you *do* accept mail. This address won't be seen by the "harvesting" programs that scour the Web looking for addresses to spam. You may get some unwanted mail at this address, but that's just the price of doing business by email. Accept it, deal with it and get on with your life.

OK, so maybe you're not willing to just let it go at that. You want some sort of revenge. Well, here's a trick to annoy the heck out of spammers. First, create a basically empty web page and name it spambait.html or some such thing. Start a list or a table on the page (depending on whether you want to bother with multiple columns or not.)

Now, whenever you get spammed, copy the email address where the spam came from onto this page. The harvester programs will find it and take the bait. Soon the spammers' email addresses will be on those lists of "millions" of email addresses sold to people doing bulk mail. They'll get a dose of their own medicine!

Chapter 15: Working With Databases

Definitions:

Database − a collection of data items

DataBase Management System (DBMS) − a computer program that manages one or more databases.

Relational DataBase Management System (RDBMS) − a computer program that manages one or more databases and allows you to get groups of data items from the whole database, in which the items are related in some way.

Column − in a database, a name for all the data items of the same exact kind. For example, one column might be email addresses (of a text type) and another may be IDs (could be a number).

Row − a set of data items containing one from each column. Sometimes a particular column in a particular row contains 0 or NULL (no value at all, which is *not* the same as 0 or "").

Table − a set of one or more Rows, where the Columns in each Row are hold the same kind of item, in exactly the same order as every other Row in the Table.

Database types

Databases fall into two broad categories: Flat−file and relational. Flat−file databases are usually just made of text files (ASCII) with one entry per line. They can be programmed easily, but they lack any special methods of indexing the data. For a very small database, they may be acceptable. The problem with this type is that the larger the database, the longer it takes to locate something specific. If you are just reading big chunks of the data in the same way over and over, this is not a bad way to go. But for general−purpose work, especially with a lot of data, stick with a relational database.

As a side note, entries in a flat−file database may contain more than one data item. Separate the "fields" in the entry with a character such as a colon. The choice of field separator (also called a delimiter) is up to you.

For best results, pick a character that won't occur in any field. Perl can take this entry as a string and split it up into separate fields. These fields would then be stored as an array. It's possible to make one field behave as an index, and do some fairly decent searching based on that.

Relational databases have one or more index tables that make it easy for the code to zero in on exactly what you want to find. Searching is fast, because your code doesn't have to read everything in the file until it gets to the item(s) you're seeking. This is all managed by the database engine and the indexes (also called hash tables). MySQL is a popular DBMS of this type. It's fast, low−cost (or free) and it's available online. You can download MySQL from:

http://www.mysql.com/

Requirements

For anything beyond a *very* small database application, you'll need access to a DBMS or RDBMS. There are several of these in common use: SyBase, Oracle, Microsoft SQL Server, MySQL and others. You'll need a way to send them commands they will be able to process. They have widely different code, features and behavior ... but they speak a common standardized language called SQL (Structured Query Language.) It is this common language that makes it easy for you to work with them in Perl.

Check the information on your Web host to see whether you have access to a DBMS. If you do, be sure to find out which one is available. Next, you'll need to be sure that the DBI module for Perl is installed on the server. Finally, make certain that the correct driver module for your particular database is installed.

All this info should be available from technical support in one way or another. Often the type of database server you can use is listed with the feature set for your hosting package. For Perl−related items, you'll probably have to call or email someone in tech support to get the answers.

Database parameters

There are certain parameter values you'll have to know and use in any script or program that must communicate with a database.

Here's an example, where the DBMS is a MySQL database program:

```
# connection data for the SQL server

my $db_name       = "customers";    # name of the database
my $db_server     = "example.com";  # location of the server
my $db_user       = "catwoman";     # username; may be optional
my $db_password   = "rrowwwrr";     # password; may be optional

# this line fully defines the database we will connect to
# it says we will be using the DBI module, using a MySQL
# "engine" called $db_name on machine $db_server

my $database = "DBI:mysql:$db_name:$db_server";
```

Once you have this information, you can paste it into any script that will be using the same database. Keep it private, though. Don't invite trouble from crackers who'd like to mess up your system.

The DBI module allows you to connect with almost any popular database server. It is *not* database−specific, meaning it doesn't really care which database you will be using. Its job is to make the connection and translate your database queries into a generic form for the database driver. The driver is the last link to the actual database server. Driver modules exist for Oracle, SyBase, MySQL, Ingres, MS SQLServer and a host of others. There is a wealth of documentation for all of the driver modules at:

CPAN and www.perl.com

Methods of access

Before you can do anything with a database, you must connect to it. As with so many things, Perl makes this easy. You declare a variable to be a database "handle" and use the connect() function to make the connection. Here's how it looks:

```
my $dbh = DBI−>connect($database, $db_user, $db_password);
```

If you have sufficient privileges set up in the MySQL database, you may not need to supply a username($db_user) and password($db_password). If this is the case, you can just put "undef" (without quotes) in place of $db_user and $db_password. If you do need a username and password, either define them in your code (refer to Database parameters above) or put them into the DBI−>connect() line. When you are finished using the database, you disconnect from it using this simple line:

$dbh−>disconnect;

While you are connected, you send SQL statements to the database to get data, add or remove data and update data in the database. There are just four basic types of statements in SQL you'll need to know:

INSERT: adds data to a table

SELECT: retrieves data from your database

UPDATE: modifies data in a table

DELETE: removes one or more rows from a table

Let's add an entry (a row) to a contact database:

$sth = $dbh−>do("**INSERT INTO** people (id, email, name, age, sex) **VALUES** ($Id, '$Email', '$Name', '$Age', '$Sex')");

This is simple. You must use single−quotes around all non−numeric items; numbers don't need them. If you're passing in the items as variables, be sure they contain exactly what you want stored. For goodness' sake, make sure that the items you're inserting (the **VALUES**) are in the correct order!

The Perl interfaces to your database server hides all the details of how the process works. This allows you to focus on what you want to do. For a **SELECT** statement, you first "prepare" the statement and then "execute" it. This is quite straightforward. You declare a variable to be your "statement handle" and use it until you don't need it anymore. It's just two lines, that look like this:

my $sth = $dbh−>prepare("**SELECT** * **FROM** mydata **WHERE** id 25");
$sth−>execute();

The statement handle comes back as a pointer to all the data that was fetched from the database. This can be one row or many rows. To get all the data items from a row, you use the fetchrow_array() function. This copies one row from a table into an array. The array indexes let you pick out any item from the row. The array will have the items in the order in which they were stored in the row of the table. Here's an example:

```
my @data;
# create the array
@data = $sth−>fetchrow_array();
# get a row of data
```

Suppose the row contained columns labeled ID,email, name, age and sex − in that exact order. Then $data[0] contains the ID from this row, $data[1] contains the email address, $data[2] is the name, $data[3] is the age and $data[4] is the person's sex. Now, you can do whatever you like with the data you just retrieved. Save it, display it, email it to Timbuktu, etc. You can repeat the process for as many rows as you want by putting these two statements in a loop.

The other statements are even simpler to use. Here's a typical **UPDATE**statement:

$sth = $dbh−>do("**UPDATE** mydata **SET** price = price+5 **WHERE** prod_id <= 10");

This statement will increase the stored values of price in your database by $5 for all items whose prod_id (stock number? model number?) is less than or equal to 10. Can you see how powerful this is?

Now suppose you want to remove some old products from your catalog database that haven't been ordered since Jan. 1, 1999. You could wipe them all out with this one line:

$sth = $dbh−>do("**DELETE** * **FROM** catalog **WHERE** last_order_year < 1999");

Chapter 16: Accepting Payments

Payment methods

Once you've moved beyond the stage of selling only other people's products via affiliate programs, you'll need to think about how to get paid. In many ways, a full−blown merchant account is the ultimate solution to taking payments. However, it takes a considerable amount of money (and time, in most cases) to get one of your own. Most of you will be far better off starting with a 3rd−party payment system like the ones described in the next few paragraphs. All the payment methods involving these 3rd−party payment processors require minimal scripting, if any.

PayPal

www.paypal.com

One of the most popular services, PayPal lets you send and receive money by email. Their Personal accounts can accept a maximum of $100/month in credit−card payments. Setup is free. Personal accounts pay no transaction fees but their features are too limited for business use. What you'll need is a Premiere or Business Account with PayPal.

With the PayPal business accounts, sellers pay $0.30 per transaction. Transactions over $15 cost an extra 2.2% (credit card payments) or 1.6% (checking account or PayPal balance). Setup is free. These accounts are ideal for taking payments on the Web for anything you want to sell. You may have all payments made to your Business account "swept" into a primary checking account daily. This will cost you an additional 0.6% See their Web site for full details.

ClickBank

www.ClickBank.com

ClickBank offers two types of accounts: ClickBank accounts and TMS Merchant Accounts. ClickBank accounts are designed for selling digital goods or services. For "tangible", or physical goods that are packed, shipped, etc., they require you to use a merchant account. Most of you will be more interested in selling digital products anyway, so ClickBank may serve your needs.

There are pros and cons to consider. On the positive side, being a ClickBank Merchant makes it easy to run an affiliate program. Affiliates sign up for a free affiliate account and sell your goods for a percentage of the price. ClickBank is very easy for the customer to use; it accepts credit cards online 24/7/365.

On the negative side, there are higher transaction fees than with PayPal. You'll be charged $1 + 7.5% on every sale. There's a one−time $49.95 set−up fee. Do the math before you decide about this one. If your product sells for $10, your transaction fee is $1.75, leaving you with $8.25 less any commissions paid if the sale was made by an affiliate.

Note: I opened a ClickBank account for the sole purpose of selling certain e−books I bought, which came with resell rights. All I had to do was put my ClickBank link in the sales pages that came with the books and upload them to my site. Sometimes it's worth a few extra bucks to get that kind of convenience.

Instabill

www.Instabill.com

Instabill provides a turnkey solution to e−commerce. They accept all major credit cards on your behalf and provide you with a shopping cart system you can customize easily. You can offer your customers discounts, special promotions, etc. You can add or change items without editing any HTML. It's all managed from a Merchant Control Panel.

The downside of all this convenience and flexibility is the cost. They take 10% of every sale. There are no other charges.

Verza/Verotel

www.Verza.com and www.Verotel.com

This is more or less two companies in one. To sell tangible, physical products online, you use Verza. They charge 4.9% + $.99 on each transaction. Both Verza and their sister company accept all major credit cards, and can take online debits as well. This is the equivalent of using your ATM/debit card at a store. Verza operates a shopping cart system on their Web site, and provides you with links into the system. You copy and paste your product links into your own Web site and begin selling at once. You can also set a commission percentage for resellers of your products. As with ClickBank, others may choose to sell your products for commission.

Verza also handles selling your services online. This is billed based on what could best be described as your "customer satisfaction index". Your customers are polled about whether or not they had a positive experience with your services. If 0−24% were pleased, your service is rated at 1 star. Satisfy 25−49% and you are rated 2 stars. If 50% or more are satisfied, you get 3 stars. Transaction fees all include $.99 each. Verza charges you 6.9%, 5.9% or 4.9%, respectively, for 1−star, 2−star and 3−star services. You start with a 2−star rating, which may go up or down after 10 transactions.

For sellers of software, content and other purely digital products, you are directed to use Verotel (if you want to work with basically the same company). Verotel accepts all major credit cards, plus online checks.

Verotel also offers 1−900 number billing. They will handle recurring charges, such as monthly or annual subscription fees for access to members−only Web sites, etc. There is no setup fee and no monthly minimum charge.

They use a "ticket" system, where they sell digital tickets to your content. Your customers buy tickets from Verotel at prices you determine, up to $75. Verotel charges you 19.0% on tickets priced from $3.75 to $9.99, 15.0% for $10.00 to $19.99 tickets and 12.0% on $20.00 to $75.00 tickets. These rates apply to credit cards and checks. The 1−900 charges are at considerably higher percentages.

Revecom

www.Revecom.com

Based on my analysis of the info on their site, Revecom is the closest thing to a merchant account I've seen. They charge a $79 setup fee and a flat rate of $25/month. There are two monthly settlement cycles: Day 1−15 (A) and day 16 to month's end (B). Charges settled in period A are paid to the merchant on day 17. Charges settled in period B are paid on the 2nd of the following month.

They will process payments for both digital and tangible products. VISA, MC, Novus/Discover, AmEx and Diners Club are accepted in 180 countries and 22 currencies. They also accept checks online using ACH. As with ATM/debit cards, online checks are processed immediately in real time. This means that they don't bounce; either the money is in the checking account and the transaction is approved or else it's declined.

Merchant accounts

Conditions in the online marketplace are changing rapidly, but one important fact remains: Accepting credit cards is almost mandatory. Most online purchasers want the speed and convenience of credit cards, as well as the safety they provide. For instance, if a merchant fails to deliver the goods, a customer can dispute the charges to the credit card and get them reversed. Your sales will increase an average of 40% or more the instant you're able to say, "We accept credit cards." These are the magic words!

Some consumers may be leery of 3rd−party payment processors, but virtually none of them will have any reservations about doing business with someone who has a merchant account.

In many ways, a full−blown merchant account is the "Cadillac" of payment systems. Partly, it makes a statement about your commitment to your business and your customers. Since it's well−known that getting one requires a commitment of time and money, most people perceive merchant account owners as being more dependable.

To accept payments using a merchant account, you need at least some type of order form. The form I used to accept credit card payments for ezine ads has already been discussed. You may want to go back and look at it again. The file is called form1.html. It collects the basic information

and does not ask for the credit card number. The reason for this is that the page is not secure.

The customer's information could, in theory, be intercepted during its trip to the Web server. Pages such as this are common. In fact some shopping cart systems are not secured during the first phase of order entry.

An order may be built up and certain information collected before the process is handed over to a secure server or to a page with SSL enabled on the server where the order is being placed.

Up to this point, we're dealing with an html page. Then, when the Submit button is pressed, a script begins the second, secure, phase of the ordering process. A page is sent to the user which shows their order information and asks them to make sure it's all correct. They may go back to the first page and make changes if necessary.

The second page, which was created by the script, contains a lot of hidden inputs. Look at the script file, ad_proc.pl, which accompanies the order form page. You'll notice that it builds the page where the customer will agree to continue on to the secure server.

In this example, I call a script on the secure server (of my merchant account provider) and pass it the information I collected. Once the customer is there, it's a secure page where their most sensitive data (credit card number, expiration date, etc.) is kept private and safe.

The inputs that are being passed to the program on the secure server have to be in the form that the program expects to get. Some of this stuff is controlled by settings I've made in the control panel for the merchant account. When you have a merchant account that provides credit card processing online in real time, you get instructions on how to send in the data. These vary from one provider to another.

I've used mine only as an example, so you could get an idea of how the process works. The bottom line is that your script has to send the purchase amount and any other important info to the secure server, using whatever software they run.

Note: It's usually up to you how much information to ask for prior to sendng your customer to the secure gateway. About the only things that are always required (by a merchant account provider) are the amount of the transaction and your account number. You can pass these items as hidden inputs anyway. The only visible part of your form would then be the "Submit" button.

My own merchant account is with E−commerce Exchange, one of the older and larger merchant account providers. Their service and support have been excellent. They, as most merchant account providers do, set me up to take VISA and MasterCard initially.

Later, I was contacted by American Express. They offered me the ability to accept their card, using the same merchant account. All I had to do was give Quick Commerce (formerly known as E−Commerce Exchange) my AmEx account number and they took care of the rest. AmEx emailed me a .GIF file of their logo to display on my Website. Now, people can see at a glance that I accept these 3 cards. Getting set up to take the Novus / DiscoverCard will be equally easy.

A newer player in the merchant account arena is the Internet Marketing Center. They will set you up and give you a less−expensive lease for the Web software license than most. It's $29.95/month for 48 months. For $10 more per month, you can shave 12 months off the lease period. They also have an affiliate program that has a generous payout.

There are a number of costs to look at when choosing a merchant account provider. First, you'll need a license to use their real−time Web software so you can take payments online. Costs for this part vary widely, and it will probably be one of the largest costs you have to pay. Usually, this will be in the form of lease payments over a period of months. You normally have the option of paying off the lease early.

If you start making money fast, pay off your lease as quickly as possible. Once it's paid off, you own the license permanently. There's seldom, if ever, any charge for software upgrades.

Then there are monthly charges you'll pay forever. These include a statement fee which is usually $10/month. This covers the cost of the provider preparing and mailing you a monthly statement of all credits and debits to your account. There may be a "gateway fee" of around $10/month for the use of their secure server.

Each transaction will cost you about $0.30−$0.35, plus a small percentage of the sale amount. The percentages vary, but usually fall in the 2−3% range. There's usually a monthly minimum on this part of the bill. A typical minimum is $25. You'll be charged this amount for any month where the transaction charges total less than the minimum. Compare *all* the charges you'll have to pay before you make a final decision.

Doing the math

OK. You're going to be taking payments, possibly for multiple items and possibly including sales tax, shipping charges, etc. You may have to multiply quantity by unit price or add several items with different prices together.

You may need to add in some or all of the other charges. In other words, you'll have to do some simple math. Perl can handle it easily in your script. Review the ReadParse example in Chapter 9. You'll see:

```
$Subtotal = $Quantity1 * $Unit_Price;
```

If you have to do subtotals on different items, add Sales Tax and Shipping Charges, it's as simple as:

```
$Subtotal_1 = $Quantity_1 * $Unit_Price_1;
$Subtotal_2 = $Quantity_2 * $Unit_Price_2;
$Subtotal_3 = $Quantity_3 * $Unit_Price_3;
$Subtotal = $Subtotal_1 + $Subtotal_2 + $Subtotal_3;
$Total = $Subtotal + $Tax + $Shipping;
```

You're not likely to need anything any more complicated than that, unless you're offering a discount percentage under certain conditions. In that case, you'd multiply the Unit_Price by {1.00 − the discount (expressed as a decimal fraction)} and continue totaling things up.

For Shipping Charges, a few "if" statements can be written to handle charges based on size of order, cost of order, weight, where it's being shipped, shipping method, etc. You could have certain choices built into your form to narrow the range of possibilities your script has to consider at one time. You can look up shipping rates for UPS, FEedEx, and UPSP at their web sites.

For sales tax, you could have a table of rates in which your script reads a small database with the tax rates in one table. Add a table with a set of shipping charges and let the database supply the numbers you need. All the things that affect your shipping rates can be built into the shipping rate table. When it's time to update the shipping rates, a few SQL statements can be used to update the table. Get the information for your tables direct from the shipping companies you plan to use.

Tip: If you are selling digital goods such as ebooks or software, be aware that easy ways exist for people to take your products without paying for them. ClickBank and Paypal merchants are the most vulnerable, but some merchant account−based ordering systems have similar loopholes. For the solution to this and 12 other information products disasters, you should read this report by Andy Brocklehurst.

Giving a receipt

Depending on how your payment system works, there can be many different ways to generate a receipt. In a lot of cases, this will be done automatically. If it isn't, just feed all the information that should be on the receipt into a block of code that will compose an email message and send it to the person who ordered something from you. Review Chapter 14: Sending Email to see how this is done. Your customer should always get an email receipt. It's also a good practice to show the customer a summary of their order just before they commit to making the payment. That's a good time to suggest something else they might like to order. This is known as "upselling". It can easily increase the size and value of the orders you get.

I found that Perl will be somewhat arbitrary in the way it displays calculated numbers. You naturally want them to have a consistent appearance. Fortunately, this is also easy. Look at the ReadParse example again, and you'll see these lines:

```
if ($Total !~ /\./) {
    $Total .= ".00";
}
if ($Total !~/\.[0−9][0−9]/) {
    $Total .= "0";
}
```

For any item where you'll show a price in dollars and cents, these lines will ensure that it has a decimal point and 2 digits to the right of that. If your testing reveals that a price may come out to a fraction of a cent, add a line to round to the nearest cent and you'll have no trouble with the display.

Chapter 17: Where To Now?

Returning the user to another page

I use this technique in the script that operates my ezine directory. A number of ezine publishers have listed their ezines in my directory. When they joined, they were given customized links. Each of these links contains an item in the query string that serves as a return path. Once a visitor has finished using the Quick−Subscribe form, they are sent to the page named in this return path. Here's an example of a return path:

?referrer=http://relay101.listbot.com

Within the script, this is parsed and the URL is assigned to a variable called $return_path. That code looks like this:

$return_path="http://relay101.listbot.com";

Now, whenever the script calls itself, calls decode.pl, etc., the variable is passed along with the call. That way, it's always available when the script finally finishes and it's time to send the visitor back home. That is done with this line:

print "Location: $FORM{'return_path'}\n\n";

At this point , the return address has become part of the group of variables stored in the array $FORM. If your usage is different than this, you might need to code it as:

print "Location: $return_path\n\n";

In any case, $return_path holds the URL of a particular Web page. The command to "print" this "Location" tells the visitor's browser to load the page located at that URL. You can use this method anytime you want the user to continue automatically to another page after your script has run.

Tip: This method is great for serving up a Thank−You page. You can update this page whenever you want, without having to edit the script. The script only needs to know *where* the page is located. Such pages should be displayed any time visitors do what you wanted them to do − buy something, subscribe to something, etc. Sure, it's an automated response, but it's still a part of good CUSTOMER SERVICE.

Serving up a custom Web page

When your script takes input from the user and possibly also from a database, you can create a new page whose content will be different every time. The mechanics of doing this are explained in the next chapter, but I want you to start thinking of some of the ways this could be useful. A very simple example (on the surface, at least) is what happens when you use a search engine. It retrieves the info you asked for and displays it on one or more pages. These pages are created at the time you ask for the data. They present it in a type of list, with list items the contain a title, a description and a link.

Now suppose you have an online catalog, with all your product descriptions, prices, stock numbers, etc. stored in a database. A customer is interested in only certain products or groups of products. He or she fills in an search form and is rewarded with a page showing info about his or her product selections.

Here's another example: Suppose you have a form that asks people a few survey questions. After the data is tabulated, a request is sent to the database to get an up−to−the−minute summary of the survey results.

The results from the database can be plugged into Web page elements and used to construct a customized page (or pages). If there are more results than can be shown on one page, links to the next/previous pages can be provided. All this fancy automation can give your customers the perception that your site is "user friendly". That's a **good thing**.

Refer to Chapter 15: Working with Databases, to see how to get information from a database. Once your script has that information stored in local variables, you can build a page with them and "print" it to the visitor's screen.

Chapter 18 will provide more details on how this is done.

Do the work once (writing the script), keep the content fresh (update the database) and let the server do most of the work. You'll give customers most of the help they need making buying decisions without having to do much ongoing work. Make a contact phone number and email address available for those questions that will need your expert help. Make the most of your precious time.

Chapter 18: Creating Web Pages With Scripts

How to create a page

It's very simple to create a page that your script can display on a Web browser. Here are the steps you should follow:

Compose the page in your favorite HTML editor. Be sure the file name ends in ".htm" or ".html". View it as a local file in your browser. If you're using Windows, open the Windows Explorer and locate the file. Double−click the file name. It will open in Internet Explorer. Edit the page until it looks just right. Each time you make a change, click the "Reload" button in the browser to see the new version.

With either browser (I.E. or Netscape), you can load the file by typing its full path name in the address bar of the browser window. I strongly recommend that you look at the page in both browsers to be sure that it will display correctly for the majority of users. Some extra effort at this stage will prevent users of one browser or the other from seeing a page that doesn't look right to them.

Be aware that some HTML editors optimize code for a certain browser. Some tags are not interpreted the same way by different browsers. To look good to the largest possible audience, try to avoid using code that is not compatible with both of the "big two". Also, keep in mind that most people are using a resolution of 800x600. You should also check to see that it looks right to all those millions of AOL users.

Note: AOL's browser is based on I. E. In general, if something looks good in I.E, it'll also look good on AOL.

Tip: It's certainly true that more people use Internet Explorer than Netscape. This is partly because the browser came with their Windows installation. Do be aware, however, that many people who use I.E. at home surf the net on their lunch hour at the office and may be using Netscape. In many corporate environments, Netscape rules! This lunchtime surfer may be financially well−off and thus a good prospective customer. Don't let a sloppy page cost you business.

Now, remove any blank lines from the .htm or .html file. This is very important, because a script loads the page differently than a browser does.

Start a subroutine in your script by typing these lines:

```
sub page_name {
   local ($page_source) = <<"HTML";
Content−type: text/html\n\n
HTML
   return $page_source;
}
```

Copy your page (the .html file) between the line that begins with "Content−type" and the line that only contains "HTML".

In order for your script to display the page, the subroutine must be called. Here, all of the HTML code, plus the header, are stored in a variable called $page_source. This variable is returned when the subroutine is called. So, this subroutine call will display the page:

```
print &page_name;
```

The print command/function of Perl copies the output of the subroutine ($page_source) to the standard output device. In this case, that's the user's Web browser.

Notice the first line of the subroutine. It's assigning "something" to $page_source. That "something" includes the page header (telling the browser what type of content to expect) and everything between the labels. The << symbols are there to say redirect everything within the block labeled "HTML" to the variable $page_source.

Everything that is found between the first line and the label "HTML" at the end is assumed to be part of the page you want to display.

Caution! If there's a blank line before the label "HTML", the script will stop adding to the contents of $page_source before reaching your end label. Before you try to use a script that makes HTML pages, be sure you've searched for and removed all blank lines in blocks of code like this.

So far, you've seen how to present a page that you've already created. In the example shown, all the code is in one block in the script and would look the same every time. However, you can do a great deal more than that! You can, for example, use a common block of code to create the top part of the page and use it every time. The bottom part, or footer, of the page could be common as well. So, you create a block that starts building the page and end it with a label. For clarity, you might make the label "HEADER", and code your block like this:

```
   local ($page_source) = <<"HEADER";
# put in the opening tag for the HTML document itself
# then fill in the HEAD and TITLE parts, the META−tags, etc.
```

```
# Open the BODY tag and put in any HTML you want to use
# at the top of the page, such as an image, a navigation
# bar, etc. Finally, end the block with a label but *not*
# the return statement
HEADER
```

Now create a similar block, that contains the last bits of code for your page, including the closing /BODY and /HTML tags. You could use a descriptive label for this block such as FOOTER, making it look like this:

```
$page_source .= <<"FOOTER";
# type or paste in your HTML code for the footer here
FOOTER
   return $page_source;
}
```

Notice that you don't localize $page_source again. If you did that, the previous HTML code would be overwritten and lost. Also, you don't include the Content−type line again, since this is to be part of the page you've already started. The footer code is simply added to what's already in $page_source.

Between the header and footer blocks, you're free to open new labels and add more code at will. Remember the rule about blank lines! You can't put any of them here, either. This middle portion of you page code can contain tables, etc., where all your custom code will appear. Anything you brought in as variables will be replaced with the values of those variables. Here's where you can get as creative as you please. You can call subroutines, read from files, make decisions based on user input, etc. Any info you want included in the resulting page can be placed within HTML tags and displayed however you want it.

Error pages

Sometimes a user enters something completely wrong. At other times, he/she forgets to fill in a required field. There are many other possible error conditions as well.

When one of these errors is detected, it's a good practice to display a special page that does three things:

- Explains what the error was
- Tells the user how to fix it
- Tells the user how to return to where the error occurred

These error pages can be easily created with a subroutine which contains all the HTML statements that make up the page. Add code to your script that tests for these error conditions and returns a value of "TRUE" or "1" when an error occurs. Whenever the test condition is "TRUE", call the subroutine that displays the error page. It's simple and totally automatic. Besides, it will keep you from getting a bunch of emails from people asking "What did I do wrong?" and " How do I...?"

Look in Appendix C for the list of subroutines included with this book. You'll find some that display different kinds of error pages. You can use them "as is" or modify them by adding/removing/changing any of the HTML code they contain.

Tip: You can make a generic error routine and pass a string describing the error when you call the error routine. The subroutine call would look something like this:

```
&ErrMsg("This is the error we noticed...!", "");
```

The string you passed to the error routine wiould be stored in a special system array variable called @_. Your error routine would then look in $_[0] for the first line of the error message, in $_[1] for the next line, etc. For multiple lines of error text, just put each line inside double−quotes and separate them with commas. Look at the subroutine ErrMsg in the script ffa.pl for an example of a one−line error screen. If you're using a multi−line message, just loop through all the items in @_ and display each one. By using a loop, it won't matter if some error messages are longer than others. One ErrMsg routine will handle them all, regardless of size. It will just display lines of text until it's shown them all.

Thank−You pages

Using the techniques I've outlined, you can make pages that thank your customer for placing an order. This serves several purposes. First, it lets them know the order was processed successfully. Second, it shows you are making the effort to provide good customer service. Finally, it is one of your best possible chances to sell them something else or deliver an extra bonus item.

Suppose you decided to add an extra report, e−book, etc., to the offer and make it a surprise. Perhaps you gave them a free trial of some product or service you provide. You got them in a buying mood, made a sale and then gave them more than they expected. What will they tell their friends about you?

Why make dynamic pages?

I'll define dynamic pages as those that can look different each time they appear on a visitor's monitor. There can be any number of reasons why they are different. The point is that they do. If you're providing content that changes often, you don't really want to rewrite pages every time the content changes...do you? Why not store the content in text files or a database and let a script construct a new page from it. Now, you just update the database or upload a new version of the text file and your pages are automatically made current.

For example, say you have a page describing your newsletter. It may have a link to a file containing your most recent issue. This link might also call a script that creates an HTML page with all of your content plus any banners, etc. that you'd like to display. The script might call another script that rotates the banners. Now, you display a page with a random banner, your current content and any graphical enhancements that make your page more attractive and interesting.

Custom pages

This is just an extension of the idea of dynamic pages. Perhaps you're running a contest on your site. You can have a page display the status of the contest, who's currently ahead (if it's a competitive contest) or simply be updated whenever a small text file is changed. This file could have the description of the current prize(s), new Terms and Conditions, etc.

Maybe you have a searchable database. A custom page could display the results of a query sent to the database. Maybe you have an affiliate program or a members−only area on your site. A custom page could be used to grant people access to the private area. It could assign them a username and password (either generated by some random process or entered by the user). It could say that the user info has been emailed to the user and advise them what to do in case they didn't receive it.

Custom pages could be used to provide updated pricing, delivery and availability info if you sell tangible products. A database could have all the info, and the custom page could simply respond to a user's query with the info they want at that moment. It could, of course, provide contact info in case they have questions not answered by the database. Use your imagination! The Web can be impersonal; make the user's experience feel more personalized and they'll be back to do business with you again.

Appendix A: Recommended Reading

Perl

Logging Perl CGI Script Traffic Mine the traffic to your scripts and discover a wealth of information about your visitors. What you can learn could mean the difference between a little extra income and lucrative new career. Also get a handy script to cloak, redirect and track your links. Add a little extra code and increase the power of your scripts with this excellent ebook by master Perl programmer,William Bontrager. Click Here

Do−it−Yourself CGI Security Are your CGI programs leaving your site open to attack? Don't take chances with your valuable investments. Learn what the holes are and how to find and fix them. You wouldn't leave your car unlocked in a bad neighborhood ... would you? The Internet has a dark side. Protect yourself with these simple but powerful techniques. Click Here

"Teach Yourself PERL 5 in 21 days", by David Till
Available at Barnes & Noble ISBN 0672308940
Publication Date: June 1996, Sams, 912pp.

This is the book I used to learn Perl. It's clear, well−written and chock−full of everything you need to know about Perl 5. There are numerous tables, references and examples. A CD−ROM is included which contains the code used in the book and the files to install Perl on your PC. Once installed, you can write, test and debug on your own computer before you try to run scripts on your server!

"Perl 5 How−To: The Definitive Perl 5 Problem−Solver", by Aidan Humphreys, Ed Weiss, Stephen Asbury, Mike Glover, S. Jason Mathews and Stephen Asbury (Editor) Available at Barnes & Noble ISBN 1571691189
Publication Date: September 1997, Sams, 2nd. ed., 867pp.

I skimmed through a few chapters and I knew I had to have this one. It's well−organized and presents a problem −> solution approach that's a serious time−saver. Buy it; you won't regret it.

Tip: When there's a "ham−fest" or "computer fair" in your area, by all means GO! You'll find bargains on lots of excellent books, some of which may be out of print or otherwise hard to find.

MySQL and SQL

"MySQL", by Paul DuBois
Available at Barnes & Noble ISBN 0735709211
Publication Date: 1st Edition December 1999, New Riders 800 pp.

Notes: Buying this book supports the development of MySQL. It seems to be the most complete reference available. Prior to publication, it was reviewed by Monty Widenius, the main developer of MySQL. His insight contributed to making this book a must−have for those who are serious about MySQL.

"MySQL & mSQL", by Randy Jay Yarger, George Reese & Tim King
Available at Barnes and Noble ISBN 1565924347
Publication Date: 1st Edition July 1999, O'Reilly 506 pp.

Notes: Implementation details of MySQL have changed somewhat since the book was published. There are a few cases where a function call has a new name and the name used in the book is no longer valid. Still, the O'Reilley books are very thorough. This is the one I used for learning MySQL.

"Sams' Teach Yourself SQL in 10 Minutes", by Ben Forta
Available at Barnes and Noble ISBN 0672316641
Publication Date: July 1999, Sams, paperback, 208 pp.

Notes: This is an excellent quick reference and it may be all you need to get started with SQL. It gives you a series of lessons you can finish in 10 minutes each. You can read this in one night and be writing queries right away. This a must−have book! I bought this one and you should, too − regardless of which database you'll be using.

Miscellaneous Topics

How to Hire CGI Programmer There are hundreds, even thousands of CGI programmers for hire. You can seek out and hire one with a few clicks of your mouse. But do you find the right one? You need to hire one who will craft exactly what your business needs − on time and at a reasonable cost. In short − the right person for the job. William Bontrager shows you how to get the best CGI programming talent for your money. Click Here

How to Set Your Fees as a Freelancer or Independent Consultant So, you've decided to sell your services! You've got the skills, you've made the contacts and there's paid work waiting for you. How will you know what to charge for your work? Will you sell yourself too cheap and lose money? Will you price yourself out of the market? Don't make mistakes; get the facts in a nutshell. Click Here

Appendix B: Resources

Email consulting

If you are having trouble using the strategies, scripts or subroutines in this book, I will do my best to help you. You are entitled to up to three (3) email messages. Send your questions to:

mailto:question@merrymonk.com?subject=cgi_help

Please be aware that this address is for questions related to the material and techniques presented in this book. Do not send offers, joint venture proposals, etc., to this address. Any messages without the correct subject line will be deleted automatically without being read. If you are not able to send an email by using the link above, just copy and paste the address into your email program. Be sure to make "cgi_help" (without the quotes) the Subject of your message.

Web Resources

Perl Coders − www.perlcoders.com − Get access to 133 free scripts with more being added all the time. Luink to other free−code site, tons more resources, all for a small monthly fee.

Perlmasters − www.perlmasters.com − Access to dozens of professionally written, powerful Perl scripts in many categories. Buy direct from the authors; most will install their script(s) for a fee.

CGI City − CGI−City − Programmers available for all your CGI needs. They offer original CGI scripts; some are free, some aren't.

CGI Tutorial (for MacHTTP servers) − Mac users Click Here

WebCGI − install any of their scripts using an online installer program WebCGI

Script installers

Frank Bauer − author of Add2it Mailman Pro
Web: www.Add2It.com
Email: webmaster@add2it.com

Mike Allton − ASU Service Web: www.asuservice.com or lightning.asuservice.com
Email: scripts@asuservice.com
Phone: 208−484−0833

WebFresh − They promise to install any script for US$19 and any script using a MySQL database for US$29.
Web: www.webfresh.com

Custom programming

If you feel that programming in Perl is not for you, but would like a script written, feel free to ask for my help. I will advise you as to whether I have time available for your project and whether I believe I can complete it in a timely fashion. I reserve the right to accept or reject any project I am offered. To discuss hiring me for custom programming, please write to:

mailto:coding@merrymonk.com?subject=programming

As with the "question" address, do not send offers, spam, jokes, or anything besides programming projects to this address; they will be summarily ignored and deleted. As before, the Subject "programming" is mandatory if you want your message to be read.

Web Hosting

Based on all the things I've covered, it should be clear that your Web hosting company has a lot to do with how easy (or difficult) it will be to run CGI scripts. For unlimited potential to install and run CGI scripts, you'll need a host that provides everything I've outlined. If any of the elements I've mentioned aren't available from your current hosting company, then I strongly suggest moving your site to another host.

The ultimate combination of features, speed, reliability and customer service is found at only one company I'm aware of. This company is called Host4Profit. Here's a quick description of their features:

No setup fee, 300MB disk space, 10GB traffic per month, full CGI−BIN access, ftp, telnet, MySQL database, Perl v5.6, PHP4, ASP, file manager, mail manager, control panel, *unlimited* POP email accounts, email aliases, mailing lists and autoresponders, FrontPage 98/2000 extensions, shopping cart, search engine, daily backups, site statistics, page counters, password−protected directories, use of their secure server and online support manual.

Just to make their deal even more irresistible, you'll have the opportunity to refer other clients to them and earn a $10 commission each month for each referred account. If you refer just 3 new accounts, you'll be paid more in commission than it costs to host your own site.

Appendix C: Subroutines

Where appropriate, certain lines of code are supplied in addition to the subroutine. These lines must be included near the top of the script in order for the subroutine to work. Read the comments to see how they apply to your situation.

verify_email

− Checks for worthless email addresses, returns a "1" if address is bad, otherwise returns a "0".

Explanation: This subroutine will compare the supplied email address to a list of domains and patterns which have been shown to be worthless for marketing purposes. Some domains are all autoresponders; others do not exist; still others have a consistently high ratio of bounced mail. You may add others to this "list" as needed. In your scripts, call this routine before taking any action on the email address stored in $email. Here's how I use it:

```
if (&verify_email() == 1) {
    &bad_address;
}
```

You'll have to decide what the action of the subroutine "bad_address" will be. I suggest using one of the error−page subroutines to display a page advising the user that the email address has been rejected.

```
sub verify_email {
    $Bad = 0;
    if (($email =~ /\@getresponse/i) ||
        ($email =~ /\@smartbotpro\.net/i) ||
        ($email =~ /\@autobotinfo\.com/i) ||
        ($email =~ /\@builditfast\.com/) ||
        ($email =~ /\@quicktell\.net/i) ||
        ($email =~ /\@themail\.com/i) ||
        ($email =~ /\@aweber\.com/i) ||
        ($email =~ /\@infogeneratorpro\.com/i) ||
        ($email =~ /\@autoresponder\.nu/i) ||
        ($email =~ /\@sendfree\.com/i) ||
        ($email =~ /\@autoreplying\.com/i) ||
        ($email =~ /\@WebMailStation\.com/i) ||
        ($email =~ /\@BizMailBot\.com/i)  ||
        ($email =~ /\@MyReply\.com/i) ||
        ($email =~ /\@FreeAutoresponders\.net/i) ||
        ($email =~ /\@ezrobot\.net/i) ||
        ($email =~ /\@zwallet/i) ||
        ($email =~ /\@usa\.com/i) ||
        ($email =~ /\@usa\.net/i) ||
        ($email =~ /\@clienttell\.com/i) ||
        ($email =~ /\@mail\.com/i) ||
        ($email =~ /b−i−z/i) ||
```

```
($email =~ /\@hotyellow/i) ||
($email =~ /\@email\.com/i) ||
($email =~ /\@adexec\.com/i) ||
($email =~ /\@mailseeker\.com/i) ||
($email =~ /myfreeoffice/i) ||
($email =~ /\@biz−tool\.com/i) ||
($email =~ /juno\.com/i) ||
($email =~ /\@doglover\.com/i) ||
($email =~ /\@winning\.com/i) ||
($email =~ /\@inorbit\.com/i) ||
($email =~ /\@hot−shot\.com/i) ||
($email =~ /\@yours\.com/i) ||
($email =~ /\@post\.com/i) ||
($email =~ /\@representative\.com/i) ||
($email =~ /\@write\.com/i) ||
($email =~ /\@cliffhanger\.com/i) ||
($email =~ /\@teacher\.com/i) ||
($email =~ /\@madrid\.com/i) ||
($email =~ /\@catlover\.com/i) ||
($email =~ /\@iname\.com/i) ||
($email =~ /\@execs\.com/i) ||
($email =~ /\@gtemail\.net/i) ||
($email =~ /\@wealthstream\.com/i) ||
($email =~ /\@consultant\.com/i) ||
($email =~ /\@ccnmail\.com/i) ||
($email =~ /\@stare\.com\.au/i) ||
($email =~ /\@soon\.com/i) ||
($email =~ /get.*\@excite\.com/i) ||
($email =~ /^[\d]{4}\@excite\.com/i) ||
($email =~ /\@writeme\.com/i) ||
($email =~ /\@europe\.com/i) ||
($email =~ /^vic.*\@excite\.com/i) ||
($email =~ /makemoney/i) ||
($email =~ /\@cheerful\.com/i) ||
($email =~ /\@toad\.net/i) ||
($email =~ /\@mindless\.com/i) ||
($email =~ /\@2die4\.com/i) ||
($email =~ /\@freewebsiteclub\.com/i) ||
($email =~ /\@clerk\.com/i) ||
($email =~ /\@goingplatinum\.com/i) ||
($email =~ /junk/i) ||
($email =~ /\@potspotterymore\.com/i) ||
($email =~ /\@cutey\.com/i) ||
($email =~ /\@graphic−designer\.com/i) ||
($email =~ /www\./i) ||
($email =~ /\@suggesthis\.com/i) ||
($email =~ /\@jairtel\.net/i) ||
($email =~ /powerpromotion/i) ||
($email =~ /\@pros2000\.net/i) ||
($email =~ /\@eidosmail.co\.uk/i) ||
($email =~ /\@dollars4sense\.com/i) ||
($email =~ /parsupply/i) ||
($email =~ /\@lawyer\.com/i) ||
($email =~ /\@adoortosuccess/i) ||
($email =~ /\@in−box\.com/i) ||
($email =~ /\@interban\.com/i) ||
($email =~ /\@2kdesigns\.com/i) ||
($email =~ /\@mindgates\.com/i) ||
($email =~ /ffa/i) || ($email =~ /bogus/i) ||
($email =~ /\@india\.com/i) ||
($email =~ /\@no\.com/i) ||
($email =~ /\@webriches\.net/i) ||
($email =~ /\@dr\.com/i) ||
($email =~ /\@moremail\.com/i) ||
```

```
        ($email =~ /\@engineer\.com/i) ||
        ($email =~ /\@ouvert24h\.com/i) ||
        ($email =~ /\@cyberdude\.com/i)  ||
        ($email =~ /\@doctor\.com/i) ||
        ($email =~ /[,]/) ||
        ($email =~ /\@mytown\.net/i) ||
        ($email =~ /anotherelegend/i) ||
        ($email =~ /\@myself\.com/i) ||
        ($email =~ /\@bikerider\.com/i) ||
        ($email =~ /\@sanfranmail\.com/i) ||
        ($email =~ /\@freeautobot\.com/i) ||
        ($email =~ /anotherlegend/i) ||
        ($email =~ /\@sonicnetmail/i) ||
        ($email =~ /\@hisnhers\.com/i) ||
        ($email =~ /kjg/i) || ($email =~ /kgj/i) ||
        ($email =~ /\@1234\.net/i) ||
        ($email =~ /\@financier\.com/i) ||
        ($email =~ /\@mad\.scientist\.com/i) ||
        ($email =~ /\@intelligencia\.com/i) ||
        ($email =~ /makemorenow/i) ||
        (($email =~ /\@[a−z]+[\d]{3}[\d]+[a−z]+/)
        &($email !~ /2000/) &
        ($email !~ /2001/) &($email !~ /2002/)) ||
        (($email =~ /\@[a−z]+[\d]{3}[\d]+[a−z]+/)
        &($email !~ /2000/) &
        ($email !~ /2001/) &($email !~ /2002/)) ||
        ($email =~ /anotherlegend/i) ||
        ($email =~ /spam/i)) {
        $Bad = 1;
        return $Bad;
    }
}
```

test_banned

− Here's a different way of checking for addresses or domains you've decided to reject. It uses a text file to store the addresses and domains to block. This way, you only have to add to the text file to update your process. Here, the variable $data_dir represents the path to your text file and banned.idb is the file itself.

Explanation: This subroutine will compare the supplied email address to a list of domains and patterns which have been shown to be worthless. The big difference is that you don't have to update the script to add another "banned" address. Just add it to the text file. Remember to put one address or domain per line in the text file! Here's an example of how to call the subroutine:

```
if ( &test_banned($email) ) {
  $message .= "That Email address is banned from our system ...\n";
  ;
  exit;
}

# determine if a username or address is banned
sub test_banned {
        my $addr = $_[0];
    open (BANNED, "$data_dir/banned.idb");
    @banned = ;
    chop @banned;
    close BANNED;
    $Banned = 0;
    foreach $banned (@banned) {
                $banned = quotemeta($banned);
        if (($addr =~ m/$banned/i) &($banned ne "")) {
            $addy = $banned;
            $result = 1;
            last;
        }
    }
    return $result;
}
```

Tip: You can save the previous subroutine under another name and use it to check against a different list of banned words, etc. This would simplify the ffa.pl script, for instance. Make a text file for banned words and one for banned domains. Then, put those new files in the $data_dir. In the new subroutine(s), change the filename in this line:

```
open (BANNED, "$data_dir/banned.idb");
```

Now, you don't have to keep editing the script to add new domains to block. Add the new subroutine to the script, remove the @badwords array and the for−loop. Modifiy the if−statement so it compares the return value of your new subroutine to 1. Example:

```
if (&bad_words == 1) {&error("Forbidden word in site name");}
```

bad_address

− An implementation of the action I suggested when a "bad" address is sent

Explanation: This routine sends back an error page when the user enters an unacceptable email address. It explains why the address was rejected and offers a way to correct the problem.

```
sub bad_address {
   local ($page_source) = <<"ERROR";
Content−type: text/html\n\n
<html><head><body bgcolor="#FFFFFF">
<TITLE>Sorry, but that email address was rejected</TITLE>
<H2>Sorry, but that email address was rejected</H2>
Our research has shown that this address domain is
either:<ul>
<li>Comprised entirely of autoresponders OR<br><br>
<li>Has a high percentage of bouncing addresses<br>
<br></ul>
<p>We want our newsletters delivered to valid addresses
only.<br> Also, automatic replies to every issue we mail
are a waste<br>of time and bandwidth.  Thank you for your
understanding.</p>
<p>Please click your <STRONG>BACK</STRONG> button to
return<br>to the form and select one or more newsletters.
<br>Your other answers should still be there.</p>
</body></html>
ERROR
   return $page_source;
}
```

SendEmail_Unix

− Sends an email message using Unix's sendmail program.

Explanation: This routine sends an email message using the sendmail program on a Unix or Linux system. The body of the message is contained within the print MAIL statements.

```
### The local mail program
### You may need to change the path to sendmail.
$mailprog = "/usr/bin/sendmail" . ' −t';

sub SendEmail_Unix {
    open(MAIL,"|$mailprog");
    print MAIL "To: $email\n";
    print MAIL "From: ";
    print MAIL "cul_de_sac\@merrymonk.com\n";
    print MAIL "Subject: Thanks for submitting a site";
    print MAIL "\n\n";
    print MAIL "You submitted the following data:\n\n";
    print MAIL "Title: $title\n";
    print MAIL "URL: $url\n";
    print MAIL "Section: $section\n";
    print MAIL "Submitted by: $email\n\n";
    close (MAIL);
}
```

SendEmail_NT

− Sends an email message using the iMail program for Windows NT.

Explanation: This routine sends an email message using the iMail program on a Windows NT system. Notice that it requires that the body of the message be in a file. This file can be created, used for the message and then deleted. The body of the message is contained in the print MAIL statements.

```
# for iMail on Windows NT systems the message
# must get its body from a file; a complete
# path to this file must be supplied
# Note: be sure the path is correct!
# since this is used on Windows NT, the path will contain
# backslashes ("\") which must be "escaped" with a leading
# backslash in order to make the path valid

# consult your system administrator for the correct path to imail
$mailprog = "C:\\IMAIL\\IMAIL1.EXE";
# correct this path to point to a file in your web directory
$temp_msg = "C:\\users\\example\\htdocs\\mail.txt";

sub SendEmail_NT {
  open(MAIL,">$temp_msg");
  $subject = "Your site has been added!";
  $owner   = "cul_de_sac\@merrymonk.com\n";
  print MAIL "You submitted the following data:\n\n";
  print MAIL "Title: $title\n";
  print MAIL "URL: $url\n";
  print MAIL "Section: $section\n";
  print MAIL "Submitted by: $email\n\n";
  print MAIL "\n\n";
  close (MAIL);

  # this line tells iMail to send the message
  system("$mailprog −u $owner −f $temp_msg
        −s $subject −t $email");

  # the last line deletes the message file
  unlink $temp_msg;
}
```

SendEmail

− Can send an email message using either iMail or sendmail.

Explanation: This routine will check for which mail program is available and use the appropriate code for whichever one it finds. It can be expanded to cover other mail programs should the need arise. The body of the message is contained in the print MAIL statements.

```
# uncomment the following line for UNIX systems.
# $mailprog = "/usr/bin/sendmail" . ' −t';

# for iMail on Windows NT systems, the message
# must get its body from a file; a complete
# path to this file must be supplied
# Note: be sure the path is correct!
# since this is used on Windows NT, the path will contain
# backslashes ("\") which must be "escaped" with a leading
# backslash in order to make the path valid

# uncomment the following lines for Windows NT systems
# $mailprog = "C:\\IMAIL\\IMAIL1.EXE";
# $temp_msg = "C:\\users\\example\\htdocs\\mail.txt";

sub SendEmail
{
  my $emailfound;
  my $line;
  # you can set up a text file with one address per
  # line called, in this example, nosend.txt  Mail
  # will not be sent to addresses in this file.
  if (−e "nosend.txt")
  {
    $emailfound = 0;
    open (FH, "nosend.txt");
    while (($line = ) &($emailfound == 0))
    {
      chomp $line;
      if ($line eq $email) { $emailfound = 1; }
    }
    close (FH);

    if ($emailfound != 0) { return 0; }
  }
#  open sendmail program for Unix systems
  if ($mailprog =~ /sendmail/i) {
      open(MAIL,"|$mailprog");
      print MAIL "To: $email\n";
      print MAIL "From: ";
      print MAIL "cul_de_sac\@merrymonk.com\n";
      print MAIL "Subject: Thanks for submitting a site";
      print MAIL "\n\n";
      print MAIL "You submitted the following data:\n\n";
      print MAIL "Title: $title\n";
      print MAIL "URL: $url\n";
      print MAIL "Section: $section\n";
      print MAIL "Submitted by: $email\n\n";
      close (MAIL);
  }
```

```
#  use imail if sendmail not available
  else {
      open(MAIL,">$temp_msg");
      $subject = "Your site has been added!";
      $owner   = "cul_de_sac\@merrymonk.com\n";
  }
  print MAIL "You submitted the following data:\n\n";
  print MAIL "Title: $title\n";
  print MAIL "URL: $url\n";
  print MAIL "Section: $section\n";
  print MAIL "Submitted by: $email\n\n";
  print MAIL $_[0];
  print MAIL "\n\n";
  close (MAIL);
  system("$mailprog −u $owner −f $temp_msg
        −s $subject −t $email");
  unlink $temp_msg;
}
```

dollar

− Converts a dollar amount calculated by the script to dollars and cents fromat, rounding to the nearest cent.

```
sub dollar {
    $num = shift;
    if ($num !~ /\./) {$num .= ".00"; return $num;}
    if ($num !~/\.[0−9][0−9]/) {$num .="0"; return $num;}
    $num =~ /(^[\d]+\.[\d]{2})([\d]*)/;
        $num = $1;
        $p2 = $2;
        if ($p2 =~ /^[5−9]/) {
              $num += .01;
        }
        return $num;
}
```

Call it like this:

```
$total = &dollar($total);
```

Tip: When you need to display a dollar amount in HTML code or an email message created by your script, be sure to "escape" the dollar sign (\$). Otherwise the script will mistake it for something else. Example:

```
print MAIL "Your order total is \$$total.\n";
```

get_data

− Extracts data items from the query string passed to your script. These items are stored in local variables.

Explanation: This routine extracts data items from the input stream (query string) and stores them in local variables. Once a variable has been extracted, you can apply filters to it. It assumes that you have declared a CGI object, so that it can get parameters from that object. For this to work, you must include the following lines in your script:

```
use CGI;           # enable the CGI.pm module
$query = new CGI;  # declare a CGI object
```

The subroutine will look for parameters whose names match those used in lines like this:

$Value = $query−>param('Key');

where $Value is a local variable and Key is the name of some parameter passed to the script. Notice that if an item has no value, name is stored in the array @missing_fields, which is initially an empty list. This array variable should be visible to the next subroutine (incomplete_page).

```
sub get_data {
    @missing_fields = ();
    $Name1 = $query−>param('First_Name');
    $Name1 =~ s/;.*//;
    $Name1 =~ s/,.*//;
    $Name1 =~ s/[\W]//g;
    if ($Name1 eq "") {
        $missing_fields[0] = "First Name";
    }
    $Name2 = $query−>param('Last_Name');
    $Name2 =~ s/;.*//;
    $Name2 =~ s/,.*//;
    $Name2 =~ s/[\W]//g;
    if ($Name2 eq "") {
        $missing_fields[1] = "Last Name";
    }
    $Sex = $query−>param('Gender');
    $Age = $query−>param('Age');
    if ($Age eq "") {
        $missing_fields[2] = "Age";
    }
    $Email = $query−>param('Email');
    if ($Email !~
        /^[\w\d][\w\d\,\.\−]*\@([\w\d\−]+\.)+
      ([a−zA−Z]{3}|[a−zA−Z]{2})$/) {
        $Email = "";
    }
    if ($Email eq "") {
        $missing_fields[3] = "Email";
    }
```

```
    $State = $query−>param('State');
    $Country = $query−>param('Country');
    if (scalar(@missing_fields) > 0) {
        print &incomplete_page;
        exit(0);
    }
}
```

incomplete_page

− Checks for any items in the array @missing_fields. If any are found, it builds a page to tell the user what is missing and how to correct the error(s).

Explanation: This routine will search for missing entries and tell the user which items were not filled out on the form. It assumes that the array $missing_fields contains the names of any fields that weren't filled in.

```
sub incomplete_page {
    local ($page_source) = <<"HTML";
Content−type: text/html\n\n
<html><head><body bgcolor="#FFFFFF">
<TITLE>Thanks, but oops ...</TITLE>
</head>
<H2>Thanks, but oops ...</H2>
Your information is important to us, and the
following<br>items on the form were not filled
in correctly:<UL>
HTML
    for ($m=0; $m<4; $m++) {
        if ($missing_fields[$m] ne "") {
            $page_source .= "<LI>$missing_fields[$m]";
        }
    }
$page_source .= <<"HTML";
</UL>
Please click your <STRONG>BACK</STRONG> button to
return to the formand add<br> that information. Your
other answers should still be there.
</body></html>
HTML

    return $page_source;
}
```

create_record

- − Inserts a new record into a database table. Values to be inserted can be obtained from the variables in your script.

Explanation: For the database defined in $database, with user $db_user (or undefined) and password $db_password (or undefined), this routine creates a new entry (row) into table $table. The order of items and their names within the table must match those used when the table was created! In this example, the table was defined with an AUTOINCREMENT attribute for the id field. This means that whenever a row is inserted, the id will be the next number in sequence, starting at 0. This value is stored by the local variable $ID if the row is created and inserted into the table.

If a record containing any of the items in this INSERT statement already exists in the table, it cannot be created. When this happens, a subroutine (record_found) is called, which reports the error to the user.

```
sub create_record {
   # connect to the database
   my $dbh = DBI−>connect($database,$db_user,
                          $db_password);
   my $sth = $dbh−>do("INSERT INTO people (id, email,
                       name, region, sex, age) VALUES
                      (NULL, '$Email', '$Name',
                       '$Region', '$Sex', '$Age')");
   $ID = $dbh−>{mysql_insertid};

   if (not $sth) { # record exists, can't create it
      # disconnect from the database
      $dbh−>disconnect;
      print &record_found;
      exit(0);
   }
   else {
      #retrieve ID number from the new record
      # disconnect from the database
      $dbh−>disconnect;
      return(0);
   }
}
```

record_found

− A user attempted to add a row which already exists in a table in the database.

Explanation: This routine sends an error page to the user if the user already had a record (row) in the table. Notice that the user's first name is used to personalize this error page.

```
sub record_found {
    local ($page_source) = <<"ERROR";
Content−type: text/html\n\n
<html><head><body bgcolor="#FFFFFF">
<TITLE>Thanks! You were already entered in
the database</TITLE>
<H2>Thank you for giving us your information.</H2>
<p>
$Name , you're already entered in our database.</p>
<p>To return to our home page,
<a href="http://www.example.com">Click Here</a></p>
</body></html>
ERROR
    return $page_source;
}
```

update_record

− Modifies data in one column of one row of a specified table in a database.

Explanation: This routine modifies a column in a certain row of a table. It looks to see whether the person whose record is to be updated is in the table. If not, it calls an error routine and advises the user that there is no available record. Otherwise, it changes the value of the user's age entry in the table.

```
sub update_record {
    # connect to the database
    my $dbh = DBI−>connect($database,$db_user,
                      $db_password);
    my @data;

    # prepare a SELECT statement
    my $sth = $dbh−>prepare("SELECT * FROM $table
                         WHERE id = $ID");
    $sth−>execute();

    # get the referrer's ID
    while(@data = $sth−>fetchrow_array()) {
        $ID = $data[0];
    }
    if ($sth−>rows == 0) { # $ID was not found
        $dbh−>disconnect;
        print &no_id;
        exit(0);
    }
    else {
        if ($sth) { # $ID was found
            # execute the UPDATE statement
            $sth = $dbh−>do("UPDATE $table
                         SET age = age+1
                         WHERE id = $ID");

            # disconnect from the database
            $dbh−>disconnect;
        }
        else {
            # disconnect from the database
            $dbh−>disconnect;
            exit(0);
        }
    }
}
```

no_id

− Handles invalid IDs.

Explanation: this routine creates an error page to tell the user that the ID number entered was not found in the database.

```
sub no_id {
    local ($page_source) = <<"ERROR";
Content−type: text/html\n\n
<html><head><body bgcolor="#FFFFFF">
<TITLE>Thanks, but oops ...</TITLE>
<H2>Thanks, but oops ...</H2>
The ID Number you entered, $ID , was not found
in the database.<br><br>Please click your
<STRONG>BACK</STRONG>button to return to the
form and enter<br> the number again. Perhaps
you typed it incorrectly. Anyway,<br> your other
answers should still be there.</body></html>
ERROR
    return $page_source;
}
```

check_url

− Verify that a user of your script is legitimate/acceptable.

Explanation: This routine checks to see if the request to use your script originated from a server you control. You may list as many servers in the array as you like. The script will exit instantly if the request is from an unlisted server. Note that it is possible for some people to fake the server name in the request. This is known as "IP spoofing". That means that this routine can't defend against all unauthorized use of your script. Still, it provides some security at little or no cost.

```
# @referrers allows script to be executed only on
# servers which are defined in this array.
@referrers = ('www.example.com','204.12.32.175');

sub check_url {
    # Localize the check_referrer flag which determines
    # whether the user is valid.
    local($check_referrer) = 0;
    # make sure a valid referring URL was passed in.
    if ($ENV{'HTTP_REFERRER'}) {
        foreach $referrer (@referrers) {
            if ($ENV{'HTTP_REFERRER'} =~
                m|https?://([^/]*)$referrer|i) {
                $check_referrer = 1;
                last;
            }
        }
    }
    else {
```

```
        $check_referrer = 1;
    }
    # If the HTTP_REFERRER was invalid, exit silently.
    # if you prefer, you can send back an error page
    if ($check_referrer != 1) { exit(0); }
}
```

error1

− Delivers a single error message as a Web page

Explanation − This routine reads the first element (a line of text) from $_ and displays it as the content of a very simple page. It also captures the location from which it was called and displays a link to that page.

```
sub error1
{
print S "QUIT\n";
print "Content−type: text/html\n";

print "\n";
print <<EOF;
<body bgcolor="#ffffff" text="#000000">
<font color=red>
The following error took place:
</font>
<p>
<b>$_[0]</b>
<p>
Click
 <a href="$ENV{HTTP_REFERER}">here</a> to return.
<p>
OR, click on your browser's back button to restore your data.
Correct any mistakes, then submit again.
EOF
exit(0);
}
```

error_multi

error_multi − delivers a multi−line error message as a Web page

Tip: To pass several lines of text into the error_multi routine, call it this way:

&error_multi("First line", "Second line", "Last line");

Explanation − This routine reads all the elements (lines of text) from @_ and displays it as a Web page. For every subroutine, there is a unique copy of @_. This array holds all the arguments passed to the subroutine. Since we don't know in advance how many lines we'll get, the foreach loop reads them in one at a time until there are no more. Thus, we can pass different numbers of lines in different calls to error_multi. The code will find and display them all. Notice that each time the loop runs, it declares a start and stop label (LINE). This ensures that the page will display correctly.

```
sub error_multi
{
print S "QUIT\n";
print "Content−type: text/html\n";
print "\n";
print <<TOP;
<body bgcolor="#ffffff" text="#000000">
<font color=red>
The following error took place:
</font>
<p>
TOP

foreach $i (0 .. $#_) {
print <<LINE;
<b>$_[$i]</b><br>
LINE
}

print <<BOTTOM;
<p>
Click on your browser's back button to restore your data.
Correct any mistakes, then submit again.
BOTTOM
exit(0);
}
```

Appendix D: Complete Scripts and Other Files

Scripts

- **Note:** Due to limitations of the PDF compiler, some program lines may be broken into two lines on your screen. In a few cases, this will prevent a script from working.

  Watch for lines that don't begin with a **#** and don't end with a **;** ! Some of these program lines will have to be pasted together with the next line.
- **findprograms.cgi** − this will check out your host server and tell you where the vital programs such as sendmail, perl and date are located. This will work well for those who don't have telnet access. However, it probably won't work on a Windows NT host, since this is a Unix "shell script", as opposed to a Perl script.

```
#!/bin/sh

echo 'Content−type: text/html'
# look for sendmail

sendmail=`which sendmail`;
 if [ $sendmail = '' ]; then
    echo 'Oops, could not find sendmail. Attempting to
    search the disk...'
    $sendmail=`find /usr −name sendmail −print`
    echo "Sendmail is $sendmail<br><br>"
 else
    echo "Sendmail is $sendmail<br><br>"
 fi
echo 'Do we have /usr/local/bin/perl?'
perllocver=`/usr/local/bin/perl −version`
echo "$perllocver<br><br>"
perl=`which perl`
echo "Perl is $perl<br>"
perlversion=`$perl −version`
echo "$perlversion<br><br>"

perl5=`which perl5`
echo "perl5 is $perl5<br>"
perl5version=`$perl5 −version`
echo "$perl5version<br><br>"

grep=`which grep`
echo "grep is $grep<br><br>"

date=`which date`
echo "date is $date<br><br>"
```

- **ad_proc.pl** − accepts inputs from the user via the form displayed by form1.html. Creates an order number, performs calculations and dollar−amount formatting. Displays a new page with the customer's information, total amount of order, etc. When the submit button on the new page is clicked, the info is sent to the secure server for final processing. Necessary info for customer and merchant receipts are passed to the secure server, which delivers these receipts by email.

  Note: Some lines had to be broken to make the code fit on the page (PDF limitation). If you find that this script doesn't work, you may have to look for lines that *don't* end in a ";". You'll just need to remove the line−feed, so it's all one line again. Lines that begin with "#" won't cause errors.

```
#!/usr/bin/perl

# Configuration Variables

$Tax = 0.00;

# @referrers allows forms to be used only on servers
# that are defined in this field. Edit it by replacing
# "yourdomain" with your domain's name and
# "xxx.xx.xx.xx" with your domain's numeric address
# @referrers = ('www.yourdomain.com','xxx.xx.xx.xx');
# You add as many domains to @referrers as you want.

# Main body of script

# seed the random number generator; this is only needed
# if you plan to generate something at random, like a
# component of an order number, for instance
# srand();

# Check Referring URL


# parse the form data


# any required data? if so, has user supplied it?
if ( $in{'.required_data'} & ) {
    print
    exit;
}


# do you want the environment variables?
if ( $in{'.environment'} ) {
    $message .= "\nThe environment variables are:\n\n";
    foreach (keys %ENV) {
        $message .= "\t$_: $ENV{$_}\n";
    }
  } else {
    ## let's at least report a few
    $message .= "\nReferring page: $ENV{HTTP_REFERRER}";
    $message .= "\nUser address: $ENV{REMOTE_ADDR}";
    $message .= "\nUser host: $ENV{REMOTE_HOST}";
```

```
 }


# create a customer ID; you could also import an
# affiliate ID and append it to the order number.
# Doing that would make the affiliate ID become
# a visible part of the receipt you get!
$Cust_id =
$Cust_id .= "$State";

# build a desciption of the order
# $Describe = "Classified Ad(s)";

# Regenerate page with user's input and
# calculated/hidden values
print


# End of main body of script.
# Subroutines appear below.


# Get customer number, increment by one and save

sub get_cust_num {
   open (FH, "invnum.txt");
   $c_num = <FH>;
   close (FH);
   $next_num = $c_num + 1;
   open (FH, ">invnum.txt");
   print FH $next_num;
   close(FH);
   return $c_num;
}

# Return an array of missing fields among the data, or
# undefined. You'll need to name the fields descriptively
# so users will know what form data items iare missing.

sub missing_data {
   local ($result) = 0;
   foreach ( split (/::/, $in{'.required_data'} ) ) {
      unless ($in{$_}) {
         s/^..// if $in{'.remove_indexing'};
         push (@missing_fields, $_);
         $result = 1;
      }
   }
   return $result;
}


# Return the HTML code for an incomplete submission page
# The return value of this routine, $page_source, contains
# all the HTML code, including the required header

sub incomplete_page {
   local ($page_source) = <<"FIRST_HALF";
Content−type: text/html\n\n
<TITLE>Thanks, but oops ...</TITLE>
<H2>Thanks, but oops ...</H2>
Your information is important to us, and the following
items on the form were not filled in:
<UL>
```

```
FIRST_HALF

    foreach ( @missing_fields ) {
        $page_source .= "<LI>$_\n";
    }

    $page_source .= <<"SECOND_HALF";
</UL>
Please click your <STRONG>BACK</STRONG>
button to return to the form and add that information.
Your other answers should still be there.
SECOND_HALF

    return $page_source;
}

sub check_url {

    # Localize the check_referrer flag which
    # determines whether the user is valid.
    local($check_referrer) = 0;

    # If a referring URL was specified, for each valid
    # referrer, make sure that a valid referring URL was
    # passed to the script.

    if ($ENV{'HTTP_REFERRER'}) {
      foreach $referrer (@referrers) {
        if ($ENV{'HTTP_REFERRER'} =~
              m|https?://([^/]*)$referrer|i) {
              $check_referrer = 1;
             last;
         }
      }
    }
    else {
        $check_referrer = 1;
    }

    # If the HTTP_REFERRER was invalid, send  an error.
    if ($check_referrer != 1) {
       rer');
    }
}

# Note: this code is based on an older version of
# StephenBrenner's ReadParse. It is shown for
# illustration only. I suggest using the newer
# version in his cgi−lib.pl, included with this book.

sub ReadParse {
    local (*in) = @_ if @_;
    local ($i, $key, $val);

    if ( $ENV{'REQUEST_METHOD'} eq "GET" ) {
        # replaced his MethGet function
        # don't accept GET, to make it a little harder
        # to spoof the script
        print "Content−type: text/html\n\n";
        print "Sorry, this script only accepts
             METHOD=POST. ";
        print "Use that inside your <FORM ...> tag";
        exit;
    }
```

```
elsif ($ENV{'REQUEST_METHOD'} eq "POST") {
    read(STDIN,$in,$ENV{'CONTENT_LENGTH'});
}
else {
    # Added for command line debugging
    # Supply name/value form data as a command line
    # argument Format: name1=value1\.
    # (need to escape for shell)
    # Find the first argument that's not a switch (−)
    $in = ( grep( !/^−/, @ARGV )) [0];
    $in =~ s/\\
}
@in = split(/

foreach $i (0 .. $#in) {
    # Convert plus's to spaces
    $in[$i] =~ s/\+/ /g;

    # Split into key and value.
    ($key, $val) = split(/=/,$in[$i],2);
    # splits on the first =.

    # Convert %XX from hex numbers to alphanumeric
    $key =~ s/%(..)/pack("c",hex($1))/ge;
    $val =~ s/%(..)/pack("c",hex($1))/ge;
          $val =~ s/;.*//;

    if ($key eq "Quantity1") {
        $Quantity1 = $val;
        $Quantity1 =~ s/,.*//;
        $Quantity1 =~ tr/0−9/\b/cs;
    }
    if ($key eq "Name1") {
        $Name1 = $val;
        $Name1 =~ s/,.*//;
        $Name1 =~ tr/[A−Z][a−z]/\b/cs;
    }
    if ($key eq "Name2") {
        $Name2 = $val;
        $Name2 =~ s/,.*//;
        $Name2 =~ tr/[A−Z][a−z]/\b/cs;
    }
    if ($key eq "Email") {
        $Email = $val;
        $Email =~ s/,.*//;
        if ($Email !~
            /^[\w\d][\w\d\,\.\−]*\@([\w\d\−]+\.)+
        ([a−zA−Z]{3}|[a−zA−Z]{2})$/) {
            $Email = "";
        }
    }
    if ($key eq "Phone") {
        $Phone = $val;
        $Phone =~ tr/\(\)\−[0−9]/\b/cs;
    }
    if ($key eq "Address") {
        $Address = $val;
        $Address =~ s/,.*//;
        $Address =~ tr/[A−Z][a−z][0−9]\t \./\b/cs;
    }
    if ($key eq "City") {
        $City = $val;
        $City =~ s/,.*//;
        $City =~ tr/[A−Z][a−z]/\b/cs;
```

```
    }
    if ($key eq "State") {
        $State = $val;
        $State =~ s/,.*//;
        $State =~ tr/[A−Z][a−z]/\b/cs;
    }
    if ($key eq "Zipcode") {
        $Zipcode = $val;
        $Zipcode =~ s/,.*//;
        $Zipcode =~ tr/[0−9]\−/\b/cs;
    }
    if ($key eq "Country") {
        $Country = $val;
        $Country =~ s/,.*//;
        $Country =~ tr/[A−Z][a−z]/\b/cs;
    }
    if ($key eq "Company") {
        $Company = $val;
    }
    if ($key eq "Fax") {
        $Fax = $val;
        $Fax =~ tr/\(\)\−[0−9]/\b/cs;
    }
    if ($key eq "Adtype") {
        $Adtype = $val;
        if ($Adtype eq "Regular") {
            $Describe = "Classified Ad(s)";
            $Unit_Price = 5.00;
        }
        if ($Adtype eq "Sponsor") {
            $Describe = "Top Sponsor Ad(s)";
            $Unit_Price = 10.00;
        }
        if ($Adtype eq "Solo") {
            $Describe = "Solo Ad(s)";
            $Unit_Price = 15.00;
        }
    }
    $Subtotal = $Quantity1 * $Unit_Price;
    $Total    = $Subtotal;
    if ($Subtotal !~ /\./) {
        $Subtotal .=".00";
    }
    if ($Subtotal !~/\.[0−9][0−9]/) {
        $Subtotal .="0";
    }
    if ($Tax !~ /\./) {
        $Tax .=".00";
    }
    if ($Tax !~/\.[0−9][0−9]/) {
        $Tax .="0";
    }
    if ($Total !~ /\./) {
        $Total .=".00";
    }
    if ($Total !~/\.[0−9][0−9]/) {
        $Total .="0";
    }
    # Associate key and value
    # \0 is the multiple separator
    $in{$key} .= "\0" if (defined($in{$key}));
    $in{$key} .= $val;
}
```

```
    return length($in);
}

# Generate the new page based on input data

sub regen_page {
    local ($page_source) = <<"END";
Content−type: text/html\n\n
<html>
<head>
<META http−equiv="Content−Type" content="text/html;
charset=IS0−8859−1">
<title>Classified Ad −− Order Page </title>
</head>
<body BGCOLOR="#FFFFCC" TEXT="#000000" VLINK="#FF0000"
ALINK="#FFFFFF">
<div align="center"><center>
<table border="0" width="600" cellspacing="0"
cellpadding="0">
  <tr><td><HR></td></tr>
</table>
</center></div>
<div align="center"><center>
<table border="0" width="600" cellspacing="0"
cellpadding="0">
  <tr>
    <td width="100%" align="center" height="30"
    valign="bottom">
    <strong><font color="#000080" face="Arial" size="2"
    size="3">
    Order Your Classified Ad(s) through our </font>
    <font face="Arial" size="2" color="#FF0000">Secure
    Server.</font></strong>
    </td>
  </tr>
</table>
</center></div>
<div align="center"><center>
<table border="0" width="600" cellspacing="0"
cellpadding="0">
  <tr>
    <td width="596">
    <FORM method="POST"
    action="https://secure.qc.net/transact.dll">
    <INPUT type="hidden" name="x_Version"
    value="3.0">
    <INPUT type="hidden" name="x_Login"
    value="XXXXXXX">
    <INPUT type="hidden" name="x_Show_Form"
    value="PAYMENT_FORM">
    <INPUT type="hidden" name="x_Amount"
    value="$Total" maxlength="8">
    <INPUT type="hidden" name="x_Cust_ID"
    value="$Cust_id" maxlength="20">
    <input type="hidden" name="x_Description"
    value="$Describe">
    <INPUT type="hidden" name="x_First_Name"
    value="$Name1" maxlength="15">
    <INPUT type="hidden" name="x_Last_Name"
    value="$Name2" maxlength="20">
    <INPUT type="hidden" name="x_Address"
    value="$Address" maxlength="60">
    <INPUT type="hidden" name="x_City"
    value="$City" maxlength="40">
```

```
<INPUT type="hidden" name="x_State"
value="$State" maxlength="20">
<INPUT type="hidden" name="x_Zip"
value="$Zipcode" maxlength="10">
<INPUT type="hidden" name="x_Country"
value="$Country" maxlength="60">
<INPUT type="hidden" name="x_Phone"
value="$Phone" maxlength="20">
<INPUT type="hidden" name="x_Company"
value="$Company" maxlength="60">
<INPUT type="hidden" name="x_Fax"
value="$Fax" maxlength="20">
<INPUT type="hidden" name="x_Email"
value="$Email" maxlength="48">
<INPUT type="hidden" name="x_Method"
value="CC">
<INPUT type="hidden" name="x_Email_Customer"
value="TRUE">
<INPUT type="hidden" name="x_Email_Merchant"
value="TRUE">
<INPUT type="hidden" name="x_Receipt_Link_Method"
value="LINK">
<INPUT type="hidden" name="x_Receipt_Link_Url"
value="http://www.yourdomain.com/order.html">
<table border="0" width="100%">
   <tr>
     <td width="250" colspan="2">
     <strong>
     <font face="Arial" size="2" color="#0000FF">
     Order Information:</font>
     </strong></td>
   </tr>
   <tr>
     <td width="250"><strong>
     <font face="Arial" size="2">Customer ID: </font>
     </strong></td>
     <td width="350"><strong>
     <font face="Arial" size="2">$Cust_id</font>
     </strong></td>
   </tr>
   <tr>
     <td width="250"><strong>
     <font face="Arial" size="2">Quantity: </font>
     </strong></td>
     <td width="350"><strong>
     <font face="Arial" size="2">$Quantity1</font>
     </strong></td>
   </tr>
   <tr>
     <td width="250"><strong>
     <font face="Arial" size="2">Description: </font>
     </strong></td>
     <td width="350"><strong>
     <font face="Arial" size="2">$Describe</font>
     </strong></td>
   </tr>
   <tr>
     <td width="250"><strong>
     <font face="Arial" size="2">First Name: </font>
     </strong></td>
     <td width="350"><strong>
     <font face="Arial" size="2">$Name1</font>
     </strong></td>
   </tr>
```

```
<tr>
  <td width="250"><strong>
  <font face="Arial" size="2">Last Name: </font>
  </strong></td>
  <td width="350"><strong>
  <font face="Arial" size="2">$Name2</font>
  </strong></td>
</tr>
<tr>
  <td width="250"><strong>
  <font face="Arial" size="2">Company: </font>
  </td>
  <td width="350"><strong>
  <font face="Arial" size="2">$Company</font>
  <br>
  </td>
</tr>
<tr>
  <td width="250"><strong>
  <font face="Arial" size="2">Email: </font>
  </strong></td>
  <td width="350"><strong>
  <font face="Arial" size="2">$Email</font>
  </strong></td>
</tr>
<tr>
  <td width="250"><strong>
  <font face="Arial" size="2">Phone: </font>
  </strong></td>
  <td width="350"><strong>
  <font face="Arial" size="2">$Phone</font>
  </strong></td>
</tr>
<tr>
  <td width="250"><strong>
  <font face="Arial" size="2">Fax: </font>
  </td>
  <td width="350"><strong>
  <font face="Arial" size="2">$Fax</font><br>
  </td>
</tr>
<tr>
  <td width="250"><strong>
  <font face="Arial" size="2">Total: </font>
  </strong></td>
  <td width="350"><strong>
  <font face="Arial" size="2">\$$Total</font>
  </strong></td>
</tr>
<tr>
  <td width="250" colspan="2"><strong>

  <font face="Arial" size="2" color="#0000FF">
  Address Information:</font>
  </strong></td>
 </tr>
<tr>
  <td width="250"><strong>
  <font face="Arial" size="2">Address: </font>
  </strong></td>
  <td width="350"><strong>
  <font face="Arial" size="2">$Address</font>
  </strong></td>
</tr>
```

```
  <tr>
    <td width="250"><strong>
    <font face="Arial" size="2">City: </font>
    </strong></td>
    <td width="350"><strong>
    <font face="Arial" size="2">$City</font>
    </strong></td>
  </tr>
  <tr>
    <td width="250"><strong>
    <font face="Arial" size="2">State/Province: </font>
    </td>
    <td width="350"><strong>
    <font face="Arial" size="2">$State</font>
    </td>
  </tr>
  <tr>
    <td width="250"><strong>
    <font face="Arial" size="2">Zipcode: </font>
    </td>
    <td width="350"><strong>
    <font face="Arial" size="2">$Zipcode</font>
    </td>
  </tr>
  <tr>
    <td width="250"><strong>
    <font face="Arial" size="2">Country: </font>
    </td>
    <td width="350"><strong>
    <font face="Arial" size="2">$Country</font><br>
    </td>
  <tr>
    <td valign="top" width="250"><strong>

    <font face="Arial" size="2">Ad Type: </font>
    </td>
    <td width="600"><strong>
    <font face="Arial" size="2">$Adtype</font>
    </td>
  </tr>
</table>
<div align="center">
<center><table border="0" width="100%">
  <tr><td><HR></td></tr>
  <tr>
    <td><p><STRONG>PLEASE:</STRONG>
    Look this information over carefully and be sure
    everything is correct. If you find any errors,
    please click your <STRONG>BACK</STRONG>
    button to return to the form and correct them. Your
    other answers will still be visible.
    </p></td>
  </tr>
  <tr><td><HR></td></tr>
  <tr>
    <td width="100%" height="50"><div
    align="center"><center><p>
    <input type="submit" value="Click Here for
    Secure Payment Form"></p>
    </center></div><div align="center">
    <center></td>
  </tr>
</table>
</center></div>
```

```
    </FORM>
    </td>
  </tr>
</table>
</center></div>
</body>
</html>
END

    return $page_source;
}

# if there's an error, display an error page

sub error
{
print S "QUIT\n";
print "Content−type: text/html\n";

print "\n";
print <<EOF;
<body bgcolor="#ffffff" text="#000000">
<font color=red>
The following error took place:
</font>
<p>
<b>$_[0]</b>
<p>
Click<a href="$ENV{HTTP_REFERRER}">here</a>
 to return.
<p>
OR, click on your browser's back button to restore your
data. Correct any mistakes, then submit again.
EOF
exit(0);
}
```

- **txt2web.pl** − converts a text file such as a newsletter article to an HTML page. I wrote this one to help me when I was publishing a newsletter. It may be modified in any way that suits your purposes.

```
#!/usr/bin/perl
# make sure that the FIRST line of this script contains
# the path to the Perl5 interpreter on your system

# Configuration Variables

$wordcount = 0;
$title = FALSE;
$author = FALSE;
$url_pat = "http://";
# lines shorter than this many characters will get a
# <br> added
$short_line = 45;
# put your keywords into the next variable ($keys)
# if desired
$keys = "";
# edit this variable to insert your ezine's name
# if desired
$pub = "My Own Ezine";
# edit the publication date here
$pub_date = "January 1";
# edit this variable for table width in % or pixels
# if you wish
$width = "60%";


# Main body of script

# capture name of input file, for later use
$file_name = $ARGV[0]

# if you want pass the issue date to the script, uncomment
# this line and $pub_date will be filled in automatically
# $pub_date = $ARGV[1];

open (FH,">outfile");

# insert HTML header for the page
# replace the HEX values for bgcolor, text, link,
# vlink if you want different colors
print FH ("<HTML>\n");
print FH ("<HEAD>\n");
print FH ("<META http\−equiv=\"Content−Type\"
content=\"text/html; charset=IS0\−8859\−1\">\n");

while ($line = <>) {
   if ($title eq FALSE){
      if ($line !~ /\S/){
         next;
      }
      $line =~ s/^[ \t]+//;
      $line =~ s/[ \t]+\n$/\n/;
      chop $line;
      print FH ("<TITLE>$line<\/TITLE><br><br>\n");
      print FH ("<META name=\"description\"
      CONTENT=\"$line\">\n");
      # comment out this line if you don't want a
    # "keywords" meta−tag:
      print FH ("<META name=\"keywords\"
```

```
        CONTENT=\"$keys\">\n");
        print FH ("</HEAD>\n");
      # insert a line to load your page
      # background if desired
        print FH ("<BODY bgcolor=\"#FFFFEE\"
        text=\"#000000\"
        link=\"#0000FF\"
        vlink=\"#990099\">\n");
        print FH ("<p>
        <div align=\"center\">
        <H2>$line<\/H2>\n");
        $title = TRUE;
        next;
    }
    if ($author eq FALSE){
        if ($line !~ /\S/){
            next;
        }
        $line =~ s/^[ \t]+//;
        $line =~ s/[ \t]+\n$/\n/;
        chop $line;
        print FH ("<H2>$line<\/H2><\/p>\n");
      # comment out or remove this line if you wish:
        print FH ("<p><H4>{Published on $pub_date via
      $pub}</H4></p><\/div>\n");
        print FH ("<div align=\"center\"><table
        width=\"$width\"><tr><td>\n");
        $author = TRUE;
        next;
    }

# capture blank lines and add a paragraph tag
    if ($line !~ /\S/){
        $line = "<p>\n";
        print FH ($line);
        next;
    }

# remove leading whitespace from non−blank lines
     $line =~ s/^[ \t]+//;
     $line =~ s/[ \t]+\n$/\n/;

# convert URLs to clickable links
    if ($line =~ /$url_pat/){
        $string = $line;
        $line = "";
        chop ($string);
        @words = split(/[\t ]+/, $string);
        foreach $word (@words){
            if ($word =~ /$url_pat/){
                $str1 = "<A HREF=\"$word\">$word</A>";
                $word = $str1;
            }
            $line .= $word;
            $line .= " ";
        }
        if (@words < 3){
            $line .= "<br>";
        }
    }

# convert email addresses to clickable links. Be aware that
# the script is pretty mindless about what you mean; if you
# get weird output, check your input file!
```

```
    if ($line =~ /[\da-zA-Z\-\_\.]+
       \@[\da-zA-Z\-]+\.[\da-zA-Z\-]+/){
       $string = $line;
       $line = "";
       chop ($string);
       @words = split(/[\t ]+/, $string);
       foreach $word (@words){
          if ($word =~ /\@/){
             $word =~ s/[\S]+://;
             $word=~ s/>//;
             $str1 = "<A HREF=\"mailto:
           $word\">$word</A>";
             $word = $str1;
          }
          $line .= $word;
          $line .= " ";
       }
    }

# insert <br> after very short lines
# the value of $short_line will determine
# if the line needs a line_feed added
    $linesize = length($line);
    if ($linesize < $short_line){
       chop ($line);
       $line .= "<br>";
    }

# insure that the break between two sentences
# on a line is uniform ->   [space]

     $line =~ s/\!\"  (?=[A-Z]|\"[A-Z]|\`)/\!\"\\ /g;
     $line =~ s/\)\"  (?=[A-Z]|\"[A-Z]|\`)/\)\"\\ /g;
     $line =~ s/\?\"  (?=[A-Z]|\"[A-Z]|\`)/\?\"\\ /g;
     $line =~ s/\.\"  (?=[A-Z]|\"[A-Z]|\`)/\.\"\\ /g;
     $line =~ s/\.  (?=[A-Z]|\"[A-Z]|\`)/\.\\ /g;
     $line =~ s/\.\n/\.\\ /g;
     $line =~ s/\. \n/\.\\ /g;
     $line =~ s/\.  \n/\.\\ /g;
     $line =~ s/\?  (?=[A-Z]|\"[A-Z]|\`)/\?\\ /g;
     $line =~ s/\?\n/\?\\ /g;
     $line =~ s/\? \n/\?\\ /g;
     $line =~ s/\?  \n/\?\\ /g;
     $line =~ s/\!  (?=[A-Z]|\"[A-Z]|\`)/\!\\ /g;
     $line =~ s/\!\n/\!\\ /g;
     $line =~ s/\! \n/\!\\ /g;
     $line =~ s/\!  \n/\!\\ /g;
     $line =~ s/\)  (?=[A-Z]|\"[A-Z]|\`)/\)\\ /g;
     $line =~ s/\)\n/\)\\ /g;
     $line =~ s/\) \n/\)\\ /g;
     $line =~ s/\)  \n/\)\\ /g;
     $line =~ s/\"\n/\"\\ /g;
     $line =~ s/\" \n/\"\\ /g;
     $line =~ s/\"  \n/\"\\ /g;
     $line =~ s/\n(?=[A-Z]|\"[A-Z])/\n\\ /g;

# save the current line
     print FH ("$line\n");

# count words only, ignore separator lines
# remove these lines if word count is not desired
    chop ($line);
    if ($line =~ /\w/){
       @words = split(/[\t ]+/, $line);
```

```
        $wordcount += @words;
    }
}

print FH ("</td></tr></table></div>\n");
print FH ("<br>\n");

# end of article body

# print footer for page
# using these lines as an example, replace with your
# normal page footer
print FH ("<div align=center><table
width=500><tr>\n");
print FH ("<td align=center>\n");
print FH ("<a href=\"http://www.roibot.com/r_fsfe.cgi
?R637_sigfmei\">\n");
print FH ("<img border=0 src=\"http://www.roibot.com/
actionplan/freemoneysig.gif\"></a>\n");
print FH ("</td></tr></table>\n");
print FH ("<div align=\"center\"><br>
<img src=\"../images/colorbar\.gif\" width=\"600\"
height=\"1\" alt=\"colorful line\">
<br></div>\n");
print FH ("</BODY>\n");
print FH ("</HTML>\n");

# remove if word count is not desired
print ("Total number of words: $wordcount\n");

# DO NOT EDIT BELOW THIS LINE!
close(FH);

# open "outfile" for reading
unless (open (INFILE,"outfile")) {
    die ("cannot open input file outfile\n");
}

# open $file_name (original source file)
unless (open (OUTFILE,">$file_name")) {
    die ("cannot open output file $file_name\n");
}

# copy "outfile" into original source file
$line = <INFILE>;
while ($line ne "") {
    print OUTFILE ($line);
    $line = <INFILE>;
}

# End of main body of script
```

- **rem_dupe.pl** − processes a text file containing one email address per line and removes any duplicate addresses. The output file is also sorted alphabetically.

```
#!/usr/bin/perl
# script name = rem_dupe.pl

$filename = $ARGV[0];
@input_list = ;
chop (@input_list);
@input_list = sort (@input_list);
open (OUTFILE, ">$filename");
$last_addr = $input_list[0];
print OUTFILE ("$last_addr\n");
$j = 1;
foreach $i (1 .. $#input_list) {
  $next_addr = $input_list[$i];
  if ($next_addr !~ /$last_addr/) {
    print OUTFILE ("$next_addr\n");
    $j++; # count the saved address
  }
  $last_addr = $next_addr;
}
close(OUTFILE);
print ("Finished processing $filename\n");
print ("Saved $j unique addresses\n");
exit(0);
```

- **ffa.pl** − bare−bones FFA page; can screen email addresses, site names and URLs for porn sites and known bad email address domains (and patterns). In the subroutine "BuildFile", add any HTML code desired to customize the generated page. Create and upload an empty file named emails.lst for email address storage. This file can be downloaded at any time and analyzed with ListMaster Pro to weed out bogus addresses. Get ListMaster Pro from www.analogx.com. It's free.

```
#!/usr/bin/perl

$mailprog = "/bin/sendmail";
$remote  = $ENV{'REMOTE_ADDR'};
$linksurl       = "http://www.example.com/links.shtml";

$success = 0;
$urlfound = 0;
$inputApproved = 0;
$emailApproved = 0;

@badwords = (
"69",
"adult",
"amateur",
"anal",
"anus",
"ass",
"assfuck",
"asshole",
"babe",
"balls",
"beastiality",
"beaver",
"bitch",
"bitches",
"bitchs",
"blowjob",
"boobs",
"breast",
"breasts",
"brothel",
"butt",
"butts",
"chicks",
"cleavage",
"clevage",
"clit",
"cock",
"crap",
"cum",
"cunt",
"cybersex",
"cybersluts",
"dick",
"dildo",
"erotic",
"fag",
"fantasies",
"fantasy",
"fuck",
```

```
"fucker",
"fuckers",
"fucking",
"gay",
"girls",
"hole",
"homo",
"hooters",
"horny",
"hump",
"jack−off",
"lesbian",
"lesbians",
"lick",
"masturbate",
"mistress",
"muff",
"naked",
"nasty",
"naughty",
"nude",
"oral",
"pamela",
"penis",
"pic",
"pics",
"pink",
"piss",
"pix",
"porn",
"porno",
"prostitute",
"pussy",
"queer",
"ramrod",
"screw",
"sex",
"sexual",
"sexy",
"sleazy",
"slut",
"sluts",
"snatch",
"sperm",
"suck",
"sucking",
"swallow",
"teen",
"teens",
"tgp",
"tit",
"tits",
"twat",
"vagina",
"virgin",
"virgins",
"virtualsex",
"voyeur",
"wet",
"whore",
"whores",
"xx",
"xxx",
);
```

```
read (STDIN, $input, $ENV{'CONTENT_LENGTH'});
@pairs = split(/$input);

foreach $pair (@pairs) {
  ($name, $value) = split(/=/, $pair);
  $value =~ tr/+/ /;
  $value =~
    s/%([a−fA−F0−9][a−fA−F0−9])/pack("C", hex($1))/eg;
  $value =~ s/<([^>]|\n)*>//g;
  $value =~ s/\</</g;
  $value =~ s/\>//g;
  $FORM{$name} = $value;
}

$title   = $FORM{'title'};
$email   = $FORM{'email'};
$section = $FORM{'section'};
$url     = $FORM{'url'};


$url =~ s/\s//g;



if (($title eq '') &($email eq '') &
    ($section eq '') &
    ($url eq ''))
{


  print "Pragma: no−cache\n";
  print "Location: $linksurl\n\n";

  exit;
}

if ($email eq '') {
    You Forgot To Enter Your Email Address!",
            "");
}

if ($email !~
/^[\w\d][\w\d\,\.\−]*\@([\w\d\−]+\.)+
([a−zA−Z]{3}|[a−zA−Z]{2})$/)
{
    You Entered An Invalid Email Address!",
            "");
}

if ( == 1 ) {
    You Entered An Unacceptable Email Address!",
            "");
}

# Save each new user's email address


$emailApproved = 1;

if ($title eq '') {
  You Forgot To Enter Your Site's Title!",
```

```
  "Your link could not be added, because no site title" .
  " was provided.\n\n");
}

if (length($title) > 100) {
  Your Title Is Longer Than 100 Characters!",
  "Your link could not be added, because the site title" .
  " was longer than 100 characters.\n\n");
}


$badwordslen = @badwords;
$found = 0;

for($i = 0; ($i < $badwordslen) &($found == 0); $i++)
{
  if ($title =~ /\b$badwords[$i]\b/i) { $found = 1; }
}

if ($found == 1)
{
    There's A Restricted Word In Your Title!",
    "Your link could not be added, because there was" .
    " a restricted word in the title.\n\n");
}


if (($url eq 'http://') || ($url eq '')) {
  You Forgot To Enter Your Site's URL!",
  "Your link could not be added, because no URL was" .
  " specified.\n\n");
}

if ($url !~ /^http:\/\/[\w\d\−\.]+\.[\w\d\−\.]+/) {
  You Entered An Invalid URL!",
  "Your link could not be added, because the URL" .
  " was invalid.\n\n");
}

@badurls = (
"1stlinks.com",
"206.190.84.80",
"2469eat.com",
"2findporn.com",
"3x−collections.adultsearch.net",
"4freeporno.com",
"69code.com",
"69xx.com",
"a−panty−raid.com",
"a2.pbtech.net/~arkangel",
"aaafreexxx.com",
"aboutsex.com",
"absoluteboy.com",
"absoluteebony.com",
"absolutelyebony.com",
"actualamateurs.com",
"acumtime.com/users/sex4u",
"ad−and−sale.com/penistretcher",
"adult.bulkweb.dragondata.com",
"adult.slick.net",
"adultaustraslia.com",
"adultcheck.com",
"adulthosting.com",
"adultpartypage.com",
```

"adultplaypen.com",
"adultservers.net",
"adultsexdirectory.com",
"aikane.com/asian",
"aikane.com/maylin",
"aimnet.com/~mdwelter",
"alajuela.com/glg",
"alexis.ai.net/~dutchsex",
"algonet.se/~gimbert",
"all−teen−porn.com",
"all−tna.com",
"allcumshots.com",
"allpaths.com/nonvan/esthetic",
"allsexpasswords.com",
"altbinaries.com",
"amalone.net/fab",
"amateur−pix.com/reyes",
"amateurexhibitionists.com",
"amateurs−n−more.com",
"ameram.com",
"amylane.com",
"anal.xcarole.com",
"ancas.com",
"angeljuice.com",
"animebabes.com",
"aplussex.com",
"applehead.com",
"arcticera.com",
"around−midnight.com",
"artistic−link.com",
"asiagirl.com",
"asian−sex.com",
"asianbabe.com",
"asianporn.nu",
"asiaview.com",
"ask4coeds.com",
"assgallery.com",
"atamys.com",
"awetfantasy.com",
"babe.raap.net",
"babepics.com",
"babes−international.com",
"backdoor.com",
"badbrad.com",
"baronet.net",
"bast.yiws.com",
"bekkoame.or.jp/ro/ha15562/adult",
"bekkoame.or.jp/ro/hb16551/bdsm",
"bekkoame.or.jp/~toravis",
"bellatlantic.net/~werewolf",
"bernabe−int.com/lld",
"best.com/~boxes",
"best.com/~erotismo",
"best.com/~fokus",
"best.com/~fokus/best",
"best.com/~jakalope",
"best.com/~lawler",
"best.com/~mcstudio",
"best.com/~silvar",
"best.com/~wetpics",
"best.com/~xposure",
"bestxxxlinks.com",
"bi−kinky.com",
"biggirls.com",

"bigtittycity.com",
"bigtittycity.dccweb.com",
"bj.janey.com/~blackduck",
"blacksexgoddess.com",
"blarg.net/~george1",
"blond−girls.com",
"bluemoons.com",
"bondagecastle.com",
"boxlick.com",
"boybabes.com",
"boyhollywood.com",
"boystoomen.com",
"bradshouse.com",
"brownhoney.com",
"brownsugar.com",
"brujah.com",
"brujah.com/bbw",
"brujah.com/bdsm",
"bsi−service.com/t/annikacollection",
"bsi−service.com/t/porn−heaven",
"btinternet.com/~arkangel",
"btinternet.com/~cvms",
"btinternet.com/~discokingdave",
"btinternet.com/~dixprop",
"btinternet.com/~jandl",
"btinternet.com/~k.d.r",
"btinternet.com/~roberta.godfrey",
"btinternet.com/~sas",
"buffstud.com",
"buns−n−beaver.com",
"buttfukme.com",
"buttpix.com",
"buttsnnuts.com",
"california.com/~dmarhx",
"candicums.com",
"candiland.com",
"career−pro.com",
"catalog.com/thrull",
"cathouse.com",
"cathouse.com/fuckparty",
"cathouse.com/poolboys",
"celeb−porn.com",
"celebdome.com",
"celebritydomain.com",
"celebritypink.com",
"celebrityplanet.com",
"celebs.heaven.nl",
"celebs.hypermart.net",
"celebs4free.com",
"cerridwen.ml.org/pa",
"chad.simplenet.com/carmen",
"chad.simplenet.com/jenny",
"cheetahx.com",
"chloeland.com",
"choiceweb.com/pleasure",
"cimkraf.com",
"cjnetworks.com/~tigg321",
"claudiaxxx.com",
"cleopatrasden.w1.com",
"climaxzone.com",
"closeshave.com",
"club−sex.com",
"clubgw.com",
"cnv.com",

"come.to",
"communique.net/~tc605",
"concentric.net/~Jmklass",
"concentric.net/~graphix7",
"concentric.net/~oneiros",
"concentric.net/~platlynx",
"condotiere.com",
"conk.com/world",
"contabel.com/adults",
"corvette3.com",
"coqui.net/asthar",
"crankshaft.com",
"crazypasses.com/~Jubei",
"creampuff−park.porncity.net",
"cris.com/~Anonman",
"cris.com/~Shogge",
"crl.com/~nocrap",
"crooked.demon.co.uk",
"csinfo.it/FIREWALL",
"cubescollection.raap.net",
"cumalong.com",
"cumcity.com",
"cumpany.com",
"cyberboys.net",
"cyberclit.com",
"cyberenet.net/~tman",
"cyberenet.net/~tube",
"cyberhighway.net/~mewzwoof",
"cybernuts.com",
"cyberpass.net/~miller",
"cyberpictures.com",
"cybersexsite.com",
"cyberskin.net",
"cyberzines.com",
"danish.yiws.com",
"darkheart.com",
"datcat.com/babes",
"deepdesire.com",
"deepwell.com/~horsey",
"delicious−pussy.com",
"deluxeedition.com",
"deninc.com/~celebcentral",
"deninc.com/~inferno",
"derekshomepage.com",
"desade.com",
"dessertdiner.com",
"detroitgirls.com",
"dhtt.net/zone",
"diablo.hypermart.net",
"dialspace.dial.pipex.com/town/lane/xog46",
"direct.ca/sunnydayz",
"dirty−sex.com",
"dman−presents.com",
"domina.com",
"donrod.com/skinnypimp",
"dontaskdonttell.com",
"dr−strangelove.com",
"dragondata.com/~scorpio",
"dreamcore.com",
"dreamlands.net",
"dreamweb.net",
"dsoj.com",
"dspace.dial.pipex.com/kris.brown",
"dspace.dial.pipex.com/nude.celebrities",

"dspace.dial.pipex.com/town/drive/yai61",
"duckme.com",
"dwalsh1.demon.co.uk",
"dynamix.lachesis.com",
"eagle.ca/~fleming",
"edgenet.net/sdouglas",
"efn.org/~richard",
"ekgd.com/eqe",
"ekgd.com/thesource",
"endowed.nthost.net",
"epix.net/~djdez",
"er.uqam.ca/merlin/fa191813",
"erectguys.com",
"eropics.com",
"erotic−fine−art.com",
"eroticarama.com",
"eroticpunishment.com",
"escherent.com",
"esoup.com/hollywood",
"everdark.com/cyber−affairs",
"execonn.com/entertain",
"exoticclubhouse.com",
"extreme2.x−perts.com",
"extremebabes.com",
"exul.com/13",
"exul.com/danni",
"exxxtc.com",
"fam.aust.com/hotxs",
"fantasycorp.com:81/~straycat22",
"fantasystore.com",
"fast1.com",
"fatfred.com",
"fetish.3x−sites.com",
"finike2.com/protect",
"finstersgayworld.fantasycorp.com",
"flash.net/~1kevin",
"flowergirls.com",
"forbidden.animearchive.org",
"foxx.net",
"fppro.com/manor/amerimen",
"fpw.isoc.net/byron",
"fred.net/cyberdan/adult",
"free−pics.com",
"free−sex.de",
"free−xxx−porno.com",
"freeasiansex.com",
"freeclubs.com",
"freehotxxx.com",
"freek.ttsg.com",
"freepornforall.com",
"freepussypic.com",
"freesexclub.com",
"freesexpic.net",
"freesexstories.net",
"galaxyinternet.com/hotnsexy",
"galaxyinternet.com/night",
"gamberro.com",
"gayinternetsites.com",
"gayla.com",
"gaymenaction.com",
"gayvid.com",
"gayville.com",
"gayxxxtra.com",
"gcwp.com/nakedtony",

"gecko.mdn.net/zurok",
"genesisprime.com",
"genesisprime.com/erotica",
"geeocities.com/Area51/Cavern/5521",
"geocities.com/Hollywood/Lot/7526",
"geocities.com/HotSprings/Spa/6921",
"geocities.com/Paris/7806",
"geocities.com/SoHo/5831",
"geocities.com/SouthBeach/Docks/2477",
"geocities.com/SunsetStrip/Alley/2811",
"geocities.com/WestHollywood/3017",
"geocities.com/Yosemite/Rapids/8319",
"gilcrest.com",
"girl−nextdoor.com",
"girliepix.com",
"girls4u.com",
"girlschool.com",
"girlstodiefor.nu",
"girltown.tierranet.com",
"gix.or.jp/~ki−3902",
"glasscity.net/users/jthomson",
"globalxs.nl/home/r/rhapsody",
"goodnet.com/~hoover",
"greatxxx.com",
"groupsex.com",
"guypages.com",
"happypig.com",
"hard−core.net",
"hardcore−domain.com",
"hardcorejunky.com",
"hardcoresex.nu",
"harddrive.nu",
"hardplay.com",
"hawaiianheat.com",
"he.net/~oe",
"hem2.passagen.se/zachary",
"hey−babe.com",
"hiline.net/~medic911",
"hkcelebs.com",
"hoef.com",
"holoship.com/celarc",
"holoship.com/td",
"home.att.net/~snakepit1",
"home.att.net/~the.zekester",
"home.bc.rogers.wave.ca/scullen",
"home.earthlink.net/~captnifty2",
"home.earthlink.net/~dolllady1",
"home.earthlink.net/~larrynichols",
"home.earthlink.net/~superheroine",
"home.earthlink.net/~wln",
"home.ici.net/~lasarus",
"home.intekom.co.za/jennifer",
"home.istar.ca/~mml",
"home.rmci.net/roger",
"home.worldnet.fr/vonachen",
"home1.swipnet.se/~w−18077",
"home2.inet.tele.dk/mgn",
"home5.swipnet.se/~w−51951",
"home6.inet.tele.dk/cbf",
"home6.swipnet.se/~w−60171",
"home7.swipnet.se/~w−71199",
"homepage.oberland.net/cmorron",
"homepages.enterprise.net/teers",
"homepages.force9.net/ratzo",

"hootisland.com/hc",
"horndog.net",
"hornyamateurs.com",
"hornytoadspad.com",
"hot4you.netaxs.com",
"hotbabes.co.uk",
"hotblonde.com",
"hotceleb.com",
"hotcyber.net/hotbelle",
"hotdish.com",
"hotfoxes.com",
"hoth.com/backdoor",
"hoth.com/bondage",
"hoth.com/lesextreme",
"hotmamas.com",
"hotmen4u.com",
"hotnclassy.com",
"hotsex1.com",
"hotsitez.com/~indians",
"hotsitez.com/~students/babes",
"hotxxxsex.com",
"hounddogstudio.com",
"houseofparadise.com",
"hudson.idt.net/~dgm89",
"hunterlink.net.au/~ddmn",
"hypermart.net/edual",
"ici.net/customers/myfriend",
"idiom.com/~netwit",
"ienetcorp.com/~ca",
"iloveamateurs.oz.to",
"ilynx.com",
"im−gay.com",
"imagesque.com",
"imagesque.com/blackbox",
"in2nett.com/pjohal/celebs",
"inetnow.net/~midiman",
"inexpress.net/~dgroulx",
"info.com.ph/~efran",
"infobahnos.com/~yoland",
"infonie.fr/public_html/cui−cui",
"inter9.com/analencounter",
"inter9.com/buttsrus",
"inter9.com/cumgalaxy",
"inter9.com/hotstuds",
"inter9.com/studcentral",
"intergirls.com",
"interlog.com/~ryallt",
"interlog.com/~toguy",
"internet−club.com/Usa/Marieizsxy",
"internet−dienst.de/rund",
"internetaffairsdiv.com",
"internetaffairsdiv.com/freehotteens",
"inthecyberlife.com",
"inthecyberlife.com/bestof",
"iop.com/~smut",
"iop.com/~xxx",
"istar.ca/~mml",
"jacksnatch.com",
"jasmine.ch/BDSM",
"jasroc.com",
"jaybirds.com",
"jerk−celebrities.com",
"jerky.net/~kachs",
"jessegambini.com",

"jmentx.com",
"jmentx.com/vs",
"johnparrow.com",
"jonsden.com",
"jring.inter.net",
"justnet.or.jp/ebf/uhdsppag/gogocat",
"kalimera.com",
"kandykisses.com",
"kenzkorner.com",
"kingcard.com/index−6",
"kobys.com",
"larissa.net/amanda",
"larissa.net/anita",
"larissa.net/judy",
"larissa.net/nikki",
"latentimages.com",
"latinoguys.com",
"lava.net/xxxpersonals",
"lazypays.com/animal",
"legend.yorks.com/~bettster",
"legland.holowww.com",
"lesbianlove.com",
"letzgo.com",
"lgn.com/loop/sexxx",
"lightlink.com/michael",
"livelust.com",
"llegs.com",
"loose−productions.com",
"love−bug.com",
"lovebyte.com",
"lovelylatinas.com",
"lovezoo.com",
"lust101.com",
"lust4fun.com",
"lust4life.com",
"lustpuppy.com",
"lvdi.net/~reddeath",
"magic−city.com",
"magicone.com",
"mangastore.com",
"maniaxxx.com",
"manpics.com",
"married−match.com",
"mars.he.net/~jmp",
"masturbations.com/AMSITE",
"mattmen.com/direct.html",
"mbcenterprise.com/backroom.htm",
"mcstudio.com/isabella",
"melis.com",
"members.aol.com/Bblof",
"members.aol.com/Hempmania",
"members.aol.com/binpimp",
"members.aol.com/bondysub",
"members.aol.com/drackerman",
"members.aol.com/heyprinz",
"members.aol.com/mscandice2",
"members.aol.com/smbaernrw",
"members.easyway.net/~alladin",
"members.spree.com/systemcrash",
"members.tripod.com/~Scooter1",
"members.tripod.com/~goro3",
"members.tripod.com/~interpro",
"members.tripod.com/~norbs",
"members.xoom.com/wolf0000",

"metromale.com",
"mid−night.com",
"mindspring.com/~maw01",
"mindspring.com/~ronlinran",
"minxxx.com",
"missmew.com/gay",
"missmew.com/sweet−endings",
"mistressm.com",
"mistresss.com",
"momo−net.com",
"mpegcity.com",
"mrxtc.com",
"mustangtech.net/b1.htm",
"mvm−hands.com",
"mwt.net/~chiasa",
"mychoice.com",
"nakedceleb.com",
"nakednights.com",
"nastyamateurs.com",
"nastysex.com",
"nastythings.com",
"nates.com",
"naturalbods.com",
"naughty.com",
"ncgweb.com/alicia",
"net−jollies.com",
"net−porn.com",
"netasia.net/users/xtronix",
"netcbc.com/bevsoc",
"netcom.com/~byteme2",
"netcom.com/~sat",
"netcomuk.co.uk/~rjpoyle",
"netsafari.com",
"netsafari.com/compuvision",
"netsafari.com/twister",
"netsurprise.com",
"nettaxi.com/citizens/Nomis",
"nettaxi.com/citizens/STARSPIC",
"nettaxi.com/citizens/gayview",
"nettaxi.com/citizens/hooters",
"netusa1.net/~dillmanr",
"newbourbon.com",
"night−visions.com",
"npgw.com/allison",
"nudeadultpics.com",
"nudeblondes.net",
"nudecelebrity.ml.org",
"nudeindia.com",
"nudeweb.com",
"nudybar.com/hollywood",
"nwinfo.net/~risenhrt",
"nwp.astrax.com",
"nympho.com",
"obertauern.ycom.or.at/shadowlands",
"ohboyz.com",
"omnidata.net/~amazing",
"omnisexo.com",
"orgias−en−vivo",
"orbitel.com/~dsunga",
"ott.net/~cute",
"ourfreepics.com",
"ozemail.com.au/~amirage",
"ozemail.com.au/~artyzac/adult",
"ozemail.com.au/~badboy2",

"ozemail.com.au/~intro1a/toys",
"ozemail.com.au/~lais",
"ozemail.com.au/~spika",
"ozemail.com.au/~sporn",
"ozemail.com.au/~uncut",
"ozgirlz.com",
"pages.prodigy.com/WORKS",
"pages.prodigy.com/ZLNJ83A",
"pages.prodigy.com/multimedja",
"pages.prodigy.com/sultry",
"pam.usenetbabes.com",
"pandasoft1.com/super.htm",
"panty.com",
"pantyfox.com",
"paradisiaque.com",
"participatingsites.com",
"partyxxx.com",
"pcisys.net/~dmr",
"pcoutlet.net/pal",
"peeperspen.xpages.com",
"peitsche.de",
"pentagram.org",
"perigee.net/~gds",
"petersplayhouse.com",
"phoenixsystems.com",
"photos−biron.com",
"pinkpage.com",
"pinkrose.com",
"planetgaylywood.com",
"planetmale.com",
"planetmojo.com",
"pleasurama.com",
"porn−king.com",
"porn−play.com",
"porn−star.com",
"porn−star−hardcore.com",
"pornet.com",
"pornofever.com",
"pornogate.com",
"pornographic.net",
"pornolounge.com",
"pornoplanet.com",
"pornoport.com",
"pornpavilion.com",
"pornstarvideo.net",
"pornuha.com",
"powernet.com.tr/supermodels",
"powerotic.com",
"powertech.no/clausd/erotic",
"powerup.com.au/~kevinn",
"powerup.com.au/~titan",
"powerxxx.com",
"preggos.xcarole.com",
"premier1.net/~wloomis",
"prioritymale.com",
"private−moments.com",
"professional−office.com/sugar",
"psybermagic.com/cyberx",
"puresex4u.com",
"pussyparadise.com",
"pw.net.ph/user/mustad/web/pose",
"pw1.netcom.com/~ballzy",
"pw1.netcom.com/~ccvideo",
"pw2.netcom.com/~dhanp",

"pw2.netcom.com/~leif7",
"quiknet.com/~sancho",
"quiksite.com",
"rainbow.net.au/~glamour",
"rainbow.net.au/~lee",
"rainbow.net.au/~megaloma",
"ravenhillstudios.com",
"rdlnatl.com/homepages/notw",
"ready2play.com",
"ready4me.com",
"realtime.com/~que89",
"red−dragon.com/adult",
"red−light.com",
"redheadgirls.com",
"redroot.com/bbxxx",
"redsector.com",
"reellifevideo.com",
"reno.quik.com/eriko",
"richardsons−inc.com",
"rinkworks.com",
"ripecherry.com",
"rlaneb.com",
"sas.upenn.edu/~emeyer",
"sandiegowebgirls",
"scandinavianxxx.com",
"scoopy.com",
"servtech.com/deb",
"sex−maniacs.com",
"sex−ultrashock.com",
"sex.com",
"sexafone.com",
"sexbunnies.com",
"sexcity.com",
"seximages.com",
"sexliquidator.com",
"sexmall.com",
"sexmart.tm",
"sexovision.com",
"sexpartyline.com",
"sexpics.lustpuppy.com",
"sexplosion2.com",
"sexpuppies.com",
"sexualcity.com",
"sexwebz.com",
"sexwithsluts.com",
"sexworld.yiws.com",
"sexxxations.com",
"sexxxtra.com",
"sexxxypics.com",
"sexy−ladies.com",
"sexybabespics.com",
"sexycheerleaders.com",
"sexycom.com",
"sexygirlslive.com",
"sexygirlz.com",
"sexykristen.com",
"sexyman.com",
"sexynurses.com",
"shacklealley.com",
"shavedpink.holowww.com",
"shemale−cream.com",
"shemale−express.com",
"shoga.wwa.com/~awdp",
"shutakuxxx.com",

"shylock.com",
"simplecom.net/mscols",
"simplecom.net/relax",
"sin−sational.com",
"skankboy.com",
"slamdunk.hypermart.net",
"sloppyseconds.com",
"slutwife.cavecreek.com",
"smaq.com",
"smoky−business.com",
"smutzone.com",
"sna.com/acropbbs/horsey",
"snmworld.com",
"sororitysex.net",
"south−hub.net/adult",
"southcorp.com",
"southern−charms.com",
"spellbinder.bc.ca/lss",
"stdesire.com",
"sterndevelopments.com",
"storiesandpics.com",
"streetgirl.com",
"streetwalkerpics.com",
"studlinx.com",
"studweb.com",
"supererotic.com",
"supermodelpics.com",
"supermodels2.com",
"superx.net",
"suprasex.com",
"sweetlilthings.com",
"swegirls.dccweb.com",
"swiftsite.com/micasa",
"tailblazer.com",
"tattletails.com",
"teenhideout.com",
"teensensation.com",
"teenstories.com",
"teenxxx.com",
"teleport.com/~jhjh",
"telisphere.com/~sixx",
"templeofsin.com",
"terror.u−net.com",
"tfp−promo.com",
"tgpmadness.com",
"the−catwoman.com",
"the−dutch.demon.nl",
"the−hideout.xxx−sex.com",
"thecyberpimps.com",
"thedawg.net",
"thefetishfactory.com",
"thehouse.superx.net",
"thelynx.com/www/thelynx/thelynx",
"thematrix.com/~fpanaro",
"thenastypages.com",
"thepeepingbeauty.com",
"thepussyking,com",
"thor.he.net/~holly",
"thrill−me.com",
"thrillsu.com",
"thumbnetwork.com",
"tiac.net/users/grecco",
"tiac.net/users/maxim",
"tld.net/users/freky",

"toilettarts.com",
"toolcage.com",
"top.net/private",
"topbear.com",
"tor−pw1.netcom.ca/~sexdog",
"totallyteriweigel.com",
"towerofsex.com",
"tribute−to.com/babes",
"tripleXXX.com",
"tripod.com/~atlantis_project",
"tripod.com/~looksharp",
"tryasia.com",
"turnmeon.com",
"tuttipussy.com",
"twhiteinc.com",
"twinkys.com",
"tygersden.com/xxx",
"ugotabud.com",
"ukhardcore.com",
"upsidedownseven.com/draculass",
"upsidedownseven.com/oralia",
"upsidedownseven.com/secretservice",
"uptowncabaret.com",
"user.tninet.se/~tdl900q",
"users.dircon.co.uk/~derry",
"users.dircon.co.uk/~matman",
"users.zetnet.co.uk/rdonoghue",
"userworld.com/users/ffjacob",
"usstrippers.com",
"vcity.net/kiss",
"vcity.net/pics",
"venturi.net/celebs",
"veronicag.com",
"vexations.com",
"vip−escorts.com",
"virgin.net",
"virtualisys.com/vr/pdowd",
"voyeur.com",
"voyeursparadise.com",
"voyeursworld.com",
"vtoy.fi/~beagle",
"w3links.com/raw",
"w3space.net/Galianos/xpix",
"want2play.com",
"way2much.com",
"web.demon.co.uk/photouk",
"web.demon.co.uk/photouk/www/hc2k_home.html",
"web2.airmail.net/jki659",
"webcom.com/erotique",
"webcom.com/gowl",
"webcom.com/shadow40",
"webmoves.com",
"website.lineone.net/~studeesmith",
"weeklypics.com",
"weird−sex.net",
"wet−n−sexy.com",
"wetpanties.com",
"wetwonders.com",
"wetwonders.com/1stclass",
"whackshack.com",
"wicced.net",
"wifesluts.com",
"wildcherry.net",
"win.bright.net/~ptanula",

```
"winternet.com/~dcooper",
"womenover30.com",
"worldsex.nu",
"worldwidebabes.com",
"worldwidewebservers.com/~peek",
"wwmen.com/horsey",
"wwmen.com/mendoit",
"wwwild.com",
"wwwindows.com/domains",
"wwwsexycom.com",
"wyrm.com/wyrmhole",
"x−offender.com",
"x−tacy.com",
"x0.com/~redbottom",
"xcarole.com",
"xmaster.com",
"xpics.com",
"xtrek.com",
"xxx−18.com",
"xxx−pics.net",
"xxx−picture.com",
"xxxcash.net",
"xxxchange.com",
"xxxdreamer.com",
"xxxfacials.com",
"xxxfree.com.ar",
"xxxgalaxy.com",
"xxxlesbians.com",
"xxxnetbabes.com",
"xxxnewspics.com",
"xxxreferral.com",
"xxxsluts.com",
"xxxstories.com",
"xxxstudio1.com",
"xxxtasy.com",
"xxxtatic.com",
"xxxxxxxx.com",
"yanthi.com",
"year2010.com/preg",
"year2010.com/stars",
"yiws.com",
"youllgoblind.com",
"youngerotica.com",
"youngnhorny.com",
"ziplink.net/~magicl",
"xoom.com/aalowen2",
"xoom.com/SEXYBABES",
"xoom.com/xxx_palace"
);

$badurlslen = @badurls;
$found = 0;

for($i = 0; ($i < $badurlslen) &($found == 0); $i++)
{
  if ($url =~ /$badurls[$i]/i) { $found = 1; }
}

if ($found == 1)
{
    You Can't Post A Link To That URL Here!",
    "Your link could not be added, because that URL" .
    " cannot be added here.\n\n");
}
```

```
$inputApproved = 1;




if ($urlfound eq 1)
{

  "Your url was already on the page. Your existing entry" .
  " was replaced by your new entry.\n\n");
}

elsif ($success eq 1)
{

  "Thanks for submitting your site to our links list. ");
}
else
{
  the system is busy.",
  "Your link could not be added, because of too much" .
  " traffic to the page.\n\n");
}

print "Pragma: no−cache\n";
print "Location: $linksurl\n\n";


exit;

# Determine whether this address has been banned
# You can also create a text file containing one
# address per line for specific addresses you want
# to ban. Within this subroutine, you could open
# that file and compare the current address to each
# one in the banned file. On any match, return a 1.
# Updating this text file can be manual or automated.
# Either way, you won't have to modify the script.
sub test_banned {
    $Banned = 0;
    if (($email =~ /\@getresponse/i) ||
        ($email =~ /\@smartbotpro\.net/i) ||
        ($email =~ /\@autobotinfo\.com/i) ||
        ($email =~ /\@builditfast\.com/) ||
        ($email =~ /\@quicktell\.net/i) ||
        ($email =~ /\@themail\.com/i) ||
        ($email =~ /\@aweber\.com/i) ||
        ($email =~ /\@infogeneratorpro\.com/i) ||
        ($email =~ /\@autoresponder\.nu/i) ||
        ($email =~ /\@sendfree\.com/i) ||
        ($email =~ /\@autoreplying\.com/i) ||
        ($email =~ /\@WebMailStation\.com/i) ||
        ($email =~ /\@BizMailBot\.com/i)  ||
        ($email =~ /\@MyReply\.com/i) ||
        ($email =~ /\@FreeAutoresponders\.net/i) ||
        ($email =~ /\@ezrobot\.net/i) ||
        ($email =~ /\@zwallet/i) ||
        ($email =~ /\@usa\.com/i) ||
        ($email =~ /\@usa\.net/i) ||
        ($email =~ /\@clienttell\.com/i) ||
        ($email =~ /\@mail\.com/i) ||
        ($email =~ /b−i−z/i) ||
```

```
($email =~ /\@hotyellow/i) ||
($email =~ /\@email\.com/i) ||
($email =~ /\@adexec\.com/i) ||
($email =~ /\@mailseeker\.com/i) ||
($email =~ /myfreeoffice/i) ||
($email =~ /\@biz−tool\.com/i) ||
($email =~ /juno\.com/i) ||
($email =~ /\@doglover\.com/i) ||
($email =~ /\@winning\.com/i) ||
($email =~ /\@inorbit\.com/i) ||
($email =~ /\@hot−shot\.com/i) ||
($email =~ /\@yours\.com/i) ||
($email =~ /\@post\.com/i) ||
($email =~ /\@representative\.com/i) ||
($email =~ /\@write\.com/i) ||
($email =~ /\@cliffhanger\.com/i) ||
($email =~ /\@teacher\.com/i) ||
($email =~ /\@madrid\.com/i) ||
($email =~ /\@catlover\.com/i) ||
($email =~ /\@iname\.com/i) ||
($email =~ /\@execs\.com/i) ||
($email =~ /\@gtemail\.net/i) ||
($email =~ /\@wealthstream\.com/i) ||
($email =~ /\@consultant\.com/i) ||
($email =~ /\@ccnmail\.com/i) ||
($email =~ /\@stare\.com\.au/i) ||
($email =~ /\@soon\.com/i) ||
($email =~ /get.*\@excite\.com/i) ||
($email =~ /^[\d]{4}\@excite\.com/i) ||
($email =~ /\@writeme\.com/i) ||
($email =~ /\@europe\.com/i) ||
($email =~ /^vic.*\@excite\.com/i) ||
($email =~ /makemoney/i) ||
($email =~ /\@cheerful\.com/i) ||
($email =~ /\@toad\.net/i) ||
($email =~ /\@mindless\.com/i) ||
($email =~ /\@2die4\.com/i) ||
($email =~ /\@freewebsiteclub\.com/i) ||
($email =~ /\@clerk\.com/i) ||
($email =~ /\@goingplatinum\.com/i) ||
($email =~ /junk/i) ||
($email =~ /\@potspotterymore\.com/i) ||
($email =~ /\@cutey\.com/i) ||
($email =~ /\@graphic−designer\.com/i) ||
($email =~ /www\./i) ||
($email =~ /\@suggesthis\.com/i) ||
($email =~ /\@jairtel\.net/i) ||
($email =~ /powerpromotion/i) ||
($email =~ /\@pros2000\.net/i) ||
($email =~ /\@eidosmail.co\.uk/i) ||
($email =~ /\@dollars4sense\.com/i) ||
($email =~ /parsupply/i) ||
($email =~ /\@lawyer\.com/i) ||
($email =~ /\@adoortosuccess/i) ||
($email =~ /\@in−box\.com/i) ||
($email =~ /\@interban\.com/i) ||
($email =~ /\@2kdesigns\.com/i) ||
($email =~ /\@mindgates\.com/i) ||
($email =~ /ffa/i) || ($email =~ /bogus/i) ||
($email =~ /\@india\.com/i) ||
($email =~ /\@no\.com/i) ||
($email =~ /\@webriches\.net/i) ||
($email =~ /\@dr\.com/i) ||
($email =~ /\@moremail\.com/i) ||
```

```
        ($email =~ /\@engineer\.com/i) ||
        ($email =~ /\@ouvert24h\.com/i) ||
        ($email =~ /\@cyberdude\.com/i)  ||
        ($email =~ /\@doctor\.com/i) ||
        ($email =~ /[,]/) ||
        ($email =~ /\@mytown\.net/i) ||
        ($email =~ /anotherelegend/i) ||
        ($email =~ /\@myself\.com/i) ||
        ($email =~ /\@bikerider\.com/i) ||
        ($email =~ /\@sanfranmail\.com/i) ||
        ($email =~ /\@freeautobot\.com/i) ||
        ($email =~ /anotherlegend/i) ||
        ($email =~ /\@sonicnetmail/i) ||
        ($email =~ /\@hisnhers\.com/i) ||
        ($email =~ /kjg/i) || ($email =~ /kgj/i) ||
        ($email =~ /\@1234\.net/i) ||
        ($email =~ /\@financier\.com/i) ||
        ($email =~ /\@mad\.scientist\.com/i) ||
        ($email =~ /\@intelligencia\.com/i) ||
        ($email =~ /makemorenow/i) ||
        (($email =~ /\@[a−z]+[\d]{3}[\d]+[a−z]+/) &
        ($email !~ /2000/) &
        ($email !~ /2001/) &($email !~ /2002/)) ||
        (($email =~ /\@[a−z]+[\d]{3}[\d]+[a−z]+/) &
        ($email !~ /2000/) &
        ($email !~ /2001/) &($email !~ /2002/)) ||
        ($email =~ /anotherlegend/i) ||
      ($email =~ /spam/i)) {
        $Banned = 1;
        return $Banned;
    }
}

# create or re−create the HTML file

sub BuildFile {

$success = 0;

for($nbr_tries = 0;
    ($nbr_tries < 4) &($success == 0);
    $nbr_tries++)
{
    ($d1, $d2, $d3, $d4, $d5, $d6, $d7, $d8, $d9,
     $last_mod_time, $d11, $d12, $d13) =
        stat "../htdocs/links.shtml";

    if (−e "../htdocs/links.shtml")
    {
        open (FH, "../htdocs/links.shtml");
        @body = <FH>;
        close (FH);
    }
    else
    {
        @body = ("");
    }

  $nbrprefix = "<!−−\"#−−>";

  for($j = @body − 1;
      ($j > 0) &
      (substr($body[$j], 0,
        length($nbrprefix)) ne $nbrprefix);
```

```
    $j−−)
 {
 }

 $nbrsubmits = 0;
 if (substr($body[$j], 0, length($nbrprefix))
      eq $nbrprefix) {
  $nbrsubmits = substr($body[$j], length($nbrprefix));
 }

 $nbrsubmits++;

 $temp_file_name = $remote . time;
 $temp_file_name =~ s/\.//g;

 open(FH, ">$temp_file_name");

 @sectionName = ("Business", "Computers", "Education",
               "Entertainment", "Government",
               "Health", "Miscellaneous", "Personal",
               "Recreation", "Web Stuff");

 @sectionRef = ("business", "computers", "education",
              "entertainment", "government",
              "health", "miscellaneous",
              "personal", "recreation", "webstuff");

 print FH <<EndBeginEndBegin;
<html>
<head>
<META
 NAME="description"
 CONTENT="Add your URL to this link list for FREE.">
<META
 NAME="keywords"
 CONTENT="add urls submit urls add ffa links pages
      submit links">
<title>Add URL to My FFA page</title>
</head>
<body bgcolor="#FFFFFF" TEXT="#000080" VLINK="#FF0000"
      ALINK="#FFFFFF">
<p>
EndBeginEndBegin

 print FH
"";
;
 print FH <<EndBeginEndBegin2;
<p><font face="Arial" size="2" size="+3"><B>
<div align=center>Friends of the Good Samaritan
</B></font></div><p>
<!−−
 IF YOU WANT TO ADD SOMETHING HERE, MODIFY THE
 FFA.PL FILE−−>
<center><table border="0" cellpadding="3">
<tr>
<td colspan=5><a href="#addyours">Add your
      link</a><br> </td>
</tr><tr>
<td><a href="#business">Business</a></td>
<td><a href="#computers">Computers</a></td>
<td><a href="#education">Education</a></td>
<td><a href="#entertainment">Entertainment</a></td>
<td><a href="#government">Government</a></td>
```

```
</tr><tr>
<td><a href="#health">Health</a></td>
<td><a href="#miscellaneous">Miscellaneous</a></td>
<td><a href="#personal">Personal</a></td>
<td><a href="#recreation">Recreation</a></td>
<td><a href="#webstuff">Web Stuff</a></td>
</tr></table></center>
<FORM METHOD="POST"
       ACTION="http://www.example.com/cgi−bin/ffa.pl">
<center><table border="0" cellpadding="3">
<tr><td colspan=3>
EndBeginEndBegin2

  for($sectionNbr = 0;
      ($sectionNbr < @sectionName) &
      ($section ne $sectionName[$sectionNbr]);
      $sectionNbr++)
  {
  }

  $href = "<li><a href=\"$url\">";

  for($i = 0; $i < @sectionName; $i++)
  {
    $linkprefix = "<!−−\"$i−−>";
    print FH "<a name=\"$sectionRef[$i]\"><b>" .
    "$sectionName[$i]</b></a><p>
    <ul>\n";


    $NbrLinksInSection = 0;

    if (($sectionNbr eq $i) &($inputApproved == 1))
    {
      print FH "$linkprefix$href$title</a>\n";
      $NbrLinksInSection++;
  }

    for($j = 0;
        ($j < @body) &($NbrLinksInSection < 50);
        $j++)
    {
      if (substr($body[$j], 0, length($linkprefix))
            eq $linkprefix) {
        if (substr($body[$j], length($linkprefix),
              length($href)) eq $href)
        {
          $urlfound = 1;
        }
        else
        {
          print FH $body[$j];
          $NbrLinksInSection++;
        }
      }
   }

    print FH "</ul>\n";
  }

  print FH <<EndFileEndFile;
</td></tr>
<tr><td colspan=3>
<a name="addyours"><br><br>
```

```
<u><b>Add your link here:</b></u>
<br> </a>
</td></tr>
<tr><td>   </td>
<td><b>Site Title:</b></td>
<td><INPUT TYPE="text" NAME="title"
SIZE=50 MAXLENGTH=100></td></tr>
<tr><td></td>
<td><b>Site URL:</b></td>
<td><INPUT TYPE="text" NAME="url" SIZE=50
VALUE="http://"></td></tr>
<tr><td></td>
<td><b>Email:</b></td>
<td><INPUT TYPE="text" NAME="email" SIZE=50
MAXLENGTH=100></td></tr>
<tr><td></td><td>
<b>Section:</b></td><td>
<select name="section">
EndFileEndFile

  for($i = 0; $i < @sectionName; $i++)
  {
  print FH "<option>$sectionName[$i]</option>\n";
  }

  print FH <<EndFileEndFile2;
</select></td></tr>
<tr><td colspan=3> <br>
<input type=submit value="Add Your Link">

<input type=reset value="Clear Form">
</td></tr></table></center>
</FORM><center>
EndFileEndFile2

  print FH
""
;
  print FH "</center><p>
  <small>\n$nbrprefix$nbrsubmits\n" .
  "</body></html>\n";
  close(FH);



  ($d1, $d2, $d3, $d4, $d5, $d6, $d7, $d8, $d9,
   $new_last_mod_time, $d11, $d12, $d13) =
     stat "../htdocs/links.shtml";
  if ($new_last_mod_time ne $last_mod_time)
  {
    unlink($temp_file_name);
    next;
  }

  unlink("../htdocs/links.shtml");
  rename $temp_file_name, "../htdocs/links.shtml";

  if (-e $temp_file_name)
  {
    unlink($temp_file_name);
  }

  if (!(-e "../htdocs/links.shtml"))
  {
```

```
    next;
  }

  $success = 1;
}
}

# display an error page when an input is rejected

sub ErrMsg
{
  print "Content−type: text/html\n\n" .
        "<html><head>
          <title>$_[0]</title></head>\n" .
        "<body bgcolor=white>" .
        "<br><center><font size=5
          color=blue>\n" .
        "<!−−#echo banner=\"\"−−>\n" .
        "<br clear=left><br>\n" .
        "<b>$_[0]</b></font>
          <br><br>\n" .
        "<b>Please hit your browser's
          \"back\" button and" .
        " try again.</b></center><br>\n" .
        "</body></html>";

  if (($_[1] ne "") &($emailApproved == 1))
  { _[1]); }

  if (!(−e "../htdocs/links.shtml"))
  {

  }

  exit;
}

# Save every unique address exactly once

sub SaveAddress {
    $new_addr = "";
    if (−e "emails.lst") {
        open(FH,"emails.lst");
        $curr_addr = <FH>;
        chop($curr_addr);
        # check for a unique address
        while ($curr_addr ne "") {
            if ($curr_addr =~ /$email/i){
                $new_addr = $curr_addr;
                last;
            }
            $curr_addr = <FH>;
        }
        close(FH);
        # if new address not in emails.lst,
        # append it to emails.lst
        if ($new_addr eq "") {
            open (FH, ">>emails.lst");
            print FH ($email,"\n");
            close(FH);
            # also, add to mailing list for "special_news"
            # this is an optional feature; use it only if
            # you tell people (on the FFA page) that they
            # will be added to this mailing list.
```

```
            open(MAIL,"|$mailprog −t");
            $recipient =
            "join−special_news\@mail.example.com";
            print MAIL "To: $recipient\n";
            print MAIL "From: $email\n";
            close (MAIL);
        }
    }
}

# deliver a confirmation message, unless the user's
# address is in the nosend.txt file Maintain nosend
# as you wish, if you use it at all. If it doesn't
# exist, the code will just skip that function
sub SendEmail_Unix
{
  my $refersubj;
  my $emailfound;
  my $line;


  if (−e "nosend.txt")
  {
    $emailfound = 0;
    open (FH, "nosend.txt");
    while (($line = <FH>) &
        ($emailfound == 0))
    {
      chomp $line;
      if ($line eq $email) { $emailfound = 1; }
    }
    close (FH);

    if ($emailfound != 0) { return 0; }
  }

#  open sendmail/qmail program for unix systems
#  add anything you like to the outgoing message; just
#  add more statements of the form−>  print MAIL "???\n";
  open(MAIL,"|$mailprog −t");
  print MAIL "To: $email\n";
  print MAIL "From: ";
  print MAIL "respond\@example\.com\n";
  print MAIL "Subject: Thanks for submitting your site";
  print MAIL "\n\n";
  print MAIL "This FFA page was established as a means
        of helping you\n";
  print MAIL "with your advertising endeavors. The first
        key to success \n";
  print MAIL "on the internet is getting the word out
        about your \n";
  print MAIL "products and services. We hope that this
        FREE service will \n";
  print MAIL "be of some help to you.\n\n";
  print MAIL "Good Luck In All Your Endeavors,\n\n";
  print MAIL "The Good Samaritan Web Site\n\n\n";
  print MAIL "You submitted the following link
        data:\n\n";
  print MAIL "Title: $title\n";
  print MAIL "URL: $url\n";
  print MAIL "Section: $section\n";
  print MAIL "Submitted by: $email\n\n";
  close (MAIL);
}
```

- **format.pl** − removes leading whitespace, enforces a two−space (optionally a one−space) break between sentences and corrects a few common errors. Can also insert a separator line in place of blank lines; this is a line of 65 ***** s which you can use to ensure that your line length is <= 65 characters. It also counts the words in the text file.

```
#!/usr/bin/perl

# Configuration Variables

$wordcount = 0;
$separator = "*****************************
**********************************\n";

# Main body of script

# capture name of input file, for later use

$file_name = $ARGV[0];

open (FH,">outfile");

while ($line = <>) {

# capture blank lines
   if ($line !~ /\S/){
      print FH ($line);
      #print FH ($separator);
      next;
   }

# remove leading whitespace from non−blank lines
    #$line =~ s/^[ ]{14}//;
    $line =~ s/^[ \t]+//;
    $line =~ s/[ \t]+\n$/\n/;

# ignore lines containing a URL or email address
   if ($line =~
        /\@|http:|mailto:|\.com|\.net|\.org/g){
      print FH ($line);
      next;
   }
# Note: these rules are optional. Add more or use
# less, it's all up to you.
# always capitalize Internet!
   $line =~ s/internet/Internet/g;
# edit "website" to "Web site"
   $line =~ s/website/Web site/g;
# always capitalize "Web"
   $line =~ s/web/Web/g;
# don't begin a sentence with "But" − it's tacky.
   $line =~ s/But /However, /g;
# make "signup" two words
   $line =~ s/signup/sign up/g;


# insure that the break between two sentences
# on a line is exactly two spaces
# for sentences that end with a !"
   $line =~ s/\!\"(?=[A−Z]|\"[A−Z]|\`)/\!\"  /g;
```

```
    $line =~ s/\!\" (?=[A−Z]|\"[A−Z]|\`)/\!\"  /g;

# for sentences that end with a )"
    $line =~ s/\)\"(?=[A−Z]|\"[A−Z]|\`)/\)\"  /g;
    $line =~ s/\)\" (?=[A−Z]|\"[A−Z]|\`)/\)\"  /g;

# for sentences that end with a ?"
    $line =~ s/\?\"(?=[A−Z]|\"[A−Z]|\`)/\?\"  /g;
    $line =~ s/\?\" (?=[A−Z]|\"[A−Z]|\`)/\?\"  /g;

# for sentences that end with a ."
    $line =~ s/\.\"(?=[A−Z]|\"[A−Z]|\`)/\.\"  /g;
    $line =~ s/\.\" (?=[A−Z]|\"[A−Z]|\`)/\.\"  /g;

# for sentences that end with a .
    $line =~ s/\.(?=[A−Z]|\"[A−Z]|\`)/\.  /g;
    $line =~ s/\. (?=[A−Z]|\"[A−Z]|\`)/\.  /g;
    $line =~ s/\.\.\.  /\.\.\./g;

# for sentences that end with a ?
    $line =~ s/\?(?=[A−Z]|\"[A−Z]|\`)/\?  /g;
    $line =~ s/\? (?=[A−Z]|\"[A−Z]|\`)/\?  /g;

# for sentences that end with a !
    $line =~ s/\!(?=[A−Z]|\"[A−Z]|\`)/\!  /g;
    $line =~ s/\! (?=[A−Z]|\"[A−Z]|\`)/\!  /g;

# for sentences that end with a )
    $line =~ s/\)(?=[A−Z]|\"[A−Z]|\`)/\)  /g;
    $line =~ s/\) (?=[A−Z]|\"[A−Z]|\`)/\)  /g;

# for single−space sentence breaks, use these lines:
#     $line =~ s/\!\"(?=[A−Z]|\"[A−Z]|\`)/\!\" /g;
#     $line =~ s/\)\"(?=[A−Z]|\"[A−Z]|\`)/\)\" /g;
#     $line =~ s/\?\"(?=[A−Z]|\"[A−Z]|\`)/\?\" /g;
#     $line =~ s/\.\"(?=[A−Z]|\"[A−Z]|\`)/\.\" /g;
#     $line =~ s/\.(?=[A−Z]|\"[A−Z]|\`)/\. /g;
#     $line =~ s/\?(?=[A−Z]|\"[A−Z]|\`)/\? /g;
#     $line =~ s/\! (?=[A−Z]|\"[A−Z]|\`)/\! /g;
#     $line =~ s/\)(?=[A−Z]|\"[A−Z]|\`)/\) /g;

    $line =~ s/([A−Z]{1}\.)[ ]{2}(?=[A−Z])/\1 /;

# save the current line
    print FH ($line);
# count words only, ignore separator lines
    chop ($line);
    if ($line =~ /\w/){
        @words = split(/[\t ]+/, $line);
        $wordcount += @words;
    }
}

print ("Total number of words: $wordcount\n");

close(FH);

# open "outfile" for reading

unless (open (INFILE,"outfile")) {
    die ("cannot open input file outfile\n");
}

# open $file_name (original source file)
```

```
unless (open (OUTFILE,">$file_name")) {
    die ("cannot open output file $file_name\n");
}

# copy "outfile" into original source file

$line = <INFILE>;
while ($line ne "") {
    print OUTFILE ($line);
    $line = <INFILE>;
}
exit;
```

- **e−merge.cgi** − accepts input from an HTML form, formats it and puts it into an out−going email message. The message can be personalized and there are several other interesting features. Documentation for e−merge is available elsewhere in this package. This script was written by Benjamin Turner, a former instructor at the University of San Francisco and is reproduced here with the author's permission.

```
#!/usr/local/bin/perl
# −−−−−−−−−−−−−−−−−
# e−merge 1.2.3 − 5/14/99
# http://www.usfca.edu/turner/e−merge/
#
# Benjamin Turner <bjturner@bigfoot.com>
# http://www.usfca.edu/turner/
#
# (c) Copyright 1995−1999, Benjamin Turner
#
# ========================================
#
# e−merge takes the input from an HTML form,
# does a 'mail merge' of the input into a specified
# merge file, and sends the mail merge as e−mail.
# Please see <http://www.usfca.edu/turner/e−merge>
# for directions on how to use e−merge.
#
# Changes in 1.2.3:
#
#   *   E−merge can now use regexp−special character as
#       a LOGDELIMITER such as | or * or . or ?
#
# Changes in 1.2.2:
#
#   *   E−merge wasn't properly handling web servers on
#       ports other than 80 (the standard http port),
#       since it was omitting the port number when it
#       printed the BASE HREF URL in the web page sent
#       back to the browser. This should now be fixed.
#
# Changes in 1.2.1:
#
#   *   Well, you fix one bug, you create another. In
#       adding the ability to use fields in the log that
#       weren't in the merge file, I broke e−merge for
#       those *not* using a log file (you'd get a message
#       like "Can't open log file: No such file or
#       directory"). Fixed by only looking for fields in
#       a log file if the logging option has been turned
#       on. A similar problem with display files was also
#       fixed.
#
# Changes in 1.2:
#
#   *   Looks like the log file had the same problem with
#       using fields that weren't in the merge file. Also
#       added the ability to use environment variables in
#       the log file. While I was mucking around, I added
#       the option to use alternate logfile field
#       delimiters; however, it's not something you can
#       set in the ACTION line, so you've got to be able
#       to alter the e−merge script itself to do it.
```

```
#
# Changes in 1.1.4:
#
#   *   Previously, e−merge could not substitute field
#       values in the display file unless those fields
#       were used in the mail merge file. This has been
#       fixed.
#
# Changes in 1.1.3:
#
#   *   Hey! e−merge now actually logs like it's supposed
#       to! Somehow, a stray 'c' wandered into the
#       function that stripped out CR/LF and caused it to
#       strip out everything *except* CR/LF. Duh!
#
# Changes in 1.1.2:
#
#   *   Fixed a problem with using relative URLs in the
#       display file that was causing data−less mail
#       messages to be sent.
#
# Changes in 1.1.1:
#
#   *   Fixed some spurious HTML preceding the output of
#       thedisplay file which made everything part of an
#       <H2> tag.
#
# Changes in 1.1:
#
#   *   e−merge can now use a 'display' file to control
#       the response page separately from the normal mail
#       'merge' file.
#
#   *   Fixed a bug with the log file code which caused
#       the pathname for the log file to get screwed up
#       in some cases. Duh...
#
# For a full version history, see
#  http://www.usfca.edu/usf/turner/e−merge/versions.html
#
# −−−−
# Permission is granted to use and modify this software
# to your heart's content, but don't charge for anything
# based on it. I'd love to be kept informed of what you've
# done with it or what you'd like to see it do, but I know
# that most of you won't bother to tell me. :) And
# remember, if this software fails or screws up or hurts
# anything, the risk is on your head, not mine −− I take
# no responsibility for how this software works for you.
# Having said that... Enjoy!
#
##############################################################
# Libraries #
#############
require 5.00;  # Perl 5 references used

##############################################################
# Global Variables #
####################

### MAILPROGRAM ###
# you may need to change this to reflect the real location
# of sendmail or to specify a different mail program to
# run (it must be able to read its headers from the data
```

```
# piped to it).

$mailprogram = "/usr/lib/sendmail −t";

### XSENDER ###
# This identifies e−merge as the sender of the mail and has
# a pointer to the e−merge page for those interested in
# e−merge. It gets spliced into the merge file near the top.

$xsender =
"e−merge <URL:http://www.usfca.edu/usf/turner/e−merge/>";

### LOGDELIMITER ###
# This is the delimiter used for the log file. About the
# only other likely−to−be−useful value is "," but you can
# put whatever you want.Just watch out that no one sticks
# this character in the data.

$LOGDELIMITER = "\t";

#################
# Main Program #
#################

# print out our header
print "Content−Type: text/html\n\n";

# Get user's data from form
%FORM =

# parse info about merge, display, and log files,
# and testing status
( $mergefile, $displayfile, $displayURL, $TESTING, $LOGGING )
  = ( $ENV{'PATH_TRANSLATED'} );

# read the merge file into an array for processing.
open ( MERGE, $mergefile )
  or ("Can't open merge file $mergefile: $!");

# slurp the file for later scanning
@MERGE = <MERGE>;
close MERGE;
splice ( @MERGE, 3, 0, "X−Sender: " . $xsender . "\n" );

# Find all fields, and look to see if any required
# fields are empty
%MERGE = ( \%FORM, @MERGE );

if ( $displayfile )
{
  # read in display file, too
  open( DISPLAY, $displayfile )
    or
      "Can't open display file $displayfile: $!" );
  @DISPLAY = <DISPLAY>;
  close DISPLAY;

  %MERGE =
      ( %MERGE,  \%FORM, @DISPLAY ) );
}

%MERGE = ( %MERGE, nLog( \%FORM, $LOGGING ) )
if $LOGGING;
```

```
@empty =
       ( \%FORM, ( %MERGE ) );
# chastise user for blanks
( @empty ) if ( @empty );

# Do field−value substitution on merge array
# (it's really that easy...)
foreach (@MERGE) { s|\[(.*?)\]|$MERGE{$1}|g; }

# Send off the merge array as mail
# (it'd better have valid headers)
unless ( $TESTING ) {
  ( @MERGE );
  ( $LOGGING, %MERGE ) if $LOGGING;
}

# if user requests a display file, merge that;
# otherwise, show mail
if ($displayfile)
{
  # Do field−value substitution for display file
  foreach (@DISPLAY) { s|\[(.*?)\]|$MERGE{$1}|g; }

  ( $displayURL, @DISPLAY );
}
else { ( @MERGE ); }


#########################
# Function Definitions #
#########################
#
#########################
# Get merge file path and testing status
# ( at end of Path )
sub GetOptions
{
  my ( $mergefile ) = @_;
  my ( $display, $testing, $logfile );

  # check for test mode
  $testing = ( $mergefile =~ s/);

  # now check to see if we should use a
  # display file (present?)
  $display = "display$1"
       if ( $mergefile =~ s/// );
  if ( $display ) {
       # if no display file was explicitly named,
       # make it mergefile.display
       unless ( $display =~ s|display=|| )
    {
      ( $display = $mergefile ) =~
       s#\.[^/]*?$|$#.display#;
      $displayURL = $ENV{'PATH_INFO'};
      $displayURL = s|
    }
    else
    {
       # save URL form for BASE HREF
      $displayURL = $display;

      # if $display is in ~user form,
       # use PATH_INFO to get partial path
```

```
    unless ( $display =~
        s|/?~[^/]+||e )

    # otherwise, just tack DOCUMENT_ROOT onto the front
    { $display = $ENV{'DOCUMENT_ROOT'} . $display; }
  }

  # Make extra−sure there are no .. refs in the
  # displayfile path (the server usually takes
  # care of this
  $display =~ s|\.\./||g;
 }

 # now check to see if we should log (present?)
 # and where
 $logfile = "log$1"
      if ( $mergefile =~ s/);
 if ( $logfile ) {
      # if no log file was explicitly directed,
      # make it mergefile.log
      unless ( $logfile =~ s|log=|| )
   { ( $logfile = $mergefile ) =~ s#\.[^/]*?$|$#.log#; }
  else
  {
    # if $logfile is in ~user form, use PATH_INFO
      # to get partial path
    unless ( $logfile =~ s|/?~[^/]+||e )

    # otherwise, just tack DOCUMENT_ROOT onto the front
    { $logfile = $ENV{'DOCUMENT_ROOT'} . $logfile; }
  }

  # Make extra−sure there are no .. refs in the logfile
  # path (the server usually takes care of this
  $logfile =~ s|\.\./||g;
 }
 # toss any other options
 $mergefile =~ s/

 return( $mergefile, $display, $displayURL, $testing,
      $logfile );
}

#########################################################
# Makes a hash of all fields in the merge file and
# their corresponding values from the user data
# from the html form.
sub FindMergeFields
{
 my ( $FORM, @MERGE ) = @_;
 my ( %MERGE, @fields );

 foreach ( @MERGE ) {
      # get fields on this line
      @fields = m/\[.*?\]/g;
      foreach $field ( @fields ) {
      # chop off the [] brackets
    $field =~ tr|[]||d;
      # if the first letter is $
        if ( $field =~ m/^\$(.+)/ )
      # then use env. var. w/o $
            { $MERGE{$field} = $ENV{$1}; }
      # otherwise, use field value
        else {
```

```
            $MERGE{$field} = $$FORM{$field};
          # if there are > 1 values
            if ( $MERGE{$field} =~ m/\0/ ) {
            # check if the field is preceded only
            # by white space
              if ( m/^(\s*)\[$field\]/ ) {
                $blankspace = $1;
            # if so, put it in + \n before each value
              $MERGE{$field} =~ s|\0|\n$blankspace|g;
              }
          # else, comma space
              else { $MERGE{$field} =~ s|\0|, |g; }
      }
     }
       }
  }
  return( %MERGE );
}

#############################################################
# Makes a hash of all fields in the log file and their
# corresponding values from the user data from the
# html form + environment variables.
sub FindMergeFieldsInLog
{
  my ( $FORM, $logfile ) = @_;
  my ( %MERGE, @fieldlist, $field, $formatline );

  if ( −e $logfile )
  {
       open (LOG, "$logfile") or
       t open log file: $!");

    # Read first line to decide in what order to
    # write field data
       chomp ( $formatline = <LOG> );
    @fieldlist =
       split ( /\Q$LOGDELIMITER\E/, $formatline );
    close LOG;
  }

  # if there isn't yet a log file, create one and
  # write format line
  else
  {
       open (LOG, ">$logfile") or
       ("Can't write log file: $!");

    # We'll write the format line to put the fields
    # in alphabetic order
       @fieldlist = sort ( keys %FORM );
    print LOG join
       ( $LOGDELIMITER, @fieldlist ), "\n";
    close LOG;
  }

  # add any fields in log to the Merge list
  # for each field in log
  foreach $field ( @fieldlist )
  {# if name starts with $ substitute with env. var.
    if ( $field =~ m/^\$(.+)/ )
      { $MERGE{$field} = $ENV{$1}; }
    else # otherwise user the field value
      { $MERGE{$field} = $FORM−>{$field}; }
```

```
  }

  return( %MERGE );
}

#################################################################
# Build a list of fields from merge file which are marked
# 'required−'
sub RequiredFields
{
  my ( %MERGE ) = @_;
  my ( @required );

  foreach ( sort ( keys %MERGE ) )
       { push ( @required, $_ ) if ( m/^required−/ ); }
  return( @required );
}

#################################################################
# Sends the Array as a mail message. Doesn't check mail
# headers!
sub SendArrayAsMail
{
  my ( @MAIL ) = @_;

  open(MAIL, "|$mailprogram") or
       ("Can't run sendmail: $!");
  foreach (@MAIL) { print MAIL; }
  close MAIL;
}

#################################################################
# Returns the path to a user's web directory if the path
# info was a ~user directory.
sub GetPathToUserDir
{
  my ( $userpath, $afterusername );

  $userpath = $ENV{'PATH_TRANSLATED'};

  # Strip out /~username part from PATH_INFO
  ( $afterusername = $ENV{'PATH_INFO'} ) =~ s|^/~[^/]+||;

  # Then see what that part got translated to in
  # PATH_TRANSLATED
  $userpath =~ s|(.*)$afterusername|$1|;
  return( $userpath );
}

#################################################################
# Record the fields to a tab−delimited file. Reads the
# first line of the file to determine the order the fields
# should be placed in.
sub LogInput
{
  my ( $logfile, %input ) = @_;
  my ( $formatline, $logline );

  # if there's already a log file, just read format
  # before appending
  if ( −e $logfile )
  {
       open (LOG, $logfile) or
       t open log file: $!");
```

```
    # Read first line to decide in what order to write
    # field data
        chomp ( $formatline = <LOG> );
    @fieldlist =
        split ( /\Q$LOGDELIMITER\E/, $formatline );
        open (LOG, ">>$logfile") or
        t write log file: $!");
  }
  # if there isn't yet a log file, create one and
  # write format line
  else {
        open (LOG, ">$logfile") or
        ("Can't write log file: $!");

    # We'll write the format line to put the fields
    # in alphabetic order
        @fieldlist = sort ( keys %input );
    print LOG join
        ( $LOGDELIMITER, @fieldlist ), "\n";
  }

  # look up values for fields and write them to
  # the logfile.
  $logline = join ( $LOGDELIMITER,
        ( \%input, @fieldlist ) );
  # change CR LF to space
  $logline =~ tr/\015\012/ /s;
  print LOG "$logline\n";

  close LOG;
}

############################################################
# Returns a list of of the values for the keys of the
# input array which are listed in the list following
# the ref to the input array.
sub ValuesForKeys
{
  my ( $hash, @keys ) = @_;
  my ( @values, $value );

  foreach $key (@keys) {
      # look up value for key
        $value = $$hash{$key};
      # replace nulls with ', '
      $value =~ s/\0/, /g;
      # pop value onto return list
        push ( @values, $value );
  }

  return( @values );
}

############################################################
# Shows the user the merged display file
sub ShowDisplay
{
   my ( $displayURL, @MERGE ) = @_;
   my ( $serverAddress ) = $ENV{'SERVER_NAME'};

   if ( $ENV{'SERVER_PORT'} != 80 )
      { $serverAddress .= ":$ENV{'SERVER_PORT'}"; }
```

```
   print
   "<BASE HREF=\"http://$serverAddress$displayURL\">\n";

    if ( $TESTING )
        {
          print "<TITLE>Test Results</TITLE>
          \n<H2>The following page ",
          "would have been displayed:</H2>\n<HR>\n";
        }
        # print each line of MERGE
        foreach ( @MERGE ) { print; }
}

###############################################################
# Merely shows the user what the mail looked like when it
# was sent
sub ShowMail
{
  my ( @MERGE ) = @_;

  if ( $TESTING ) {
        print "<TITLE>Test Results</TITLE>\n
        <H2>The following message ",
     "would have been sent:</H2>\n<HR>\n<PRE>";
  }
  else {
    print "<TITLE>Mail Sent</TITLE>\n
    <H2>The following message was ",
    "sent:</H2>\n<HR>\n<PRE>";
  }
  # print each line of MERGE
  foreach ( @MERGE ) { print; }
  print "</PRE>\n<HR>\n";
}

###############################################################
# Displays error message with a list of which fields must
# yet be filled
sub EmptyError
{
  my ( @emptyfields ) = @_;

  print "<TITLE>Error</TITLE>\n
  <H1>Some Required Fields were left ",
  "Empty</H1>\nYou did not fill in values for the
  following ", "required fields:<P>\n<UL>\n";
  # now show user which field names they must enter
  foreach ( @emptyfields ) { print "<LI>$_\n"; }
  exit;
}

###############################################################
# Print the error message and exit the program
sub FatalError
{
  my ( $error ) = @_;
  print "<TITLE>Error</TITLE>\n
  <H3>$error</H3>\n";
  exit;
}

###############################################################
# Reads the data passed to this script from either GET or
# POST methods
```

```
sub RetrieveData
{
  my ( $data, @data, %data, $name, $value );

  # get our data from the appropriate place,
  # depending on METHOD
  if ( $ENV{'REQUEST_METHOD'} eq "GET" )
       { $data = $ENV{'QUERY_STRING'}; }
  else
       { read ( STDIN, $data, $ENV{'CONTENT_LENGTH'} ); }

  # break into separate chunks for each name=value pair
  @data = split ( /$data );

  foreach $data ( @data ) {
    ($name, $value) = split(/=/, $data);

   # Decode the URL−encoding
   $value =~ tr/+/ /;
   $value =~ s/%([A−F0−9][A−F0−9])/pack("C", hex($1))/eig;

   # if there are multiple values for the field, splice
   # them together with null characters
   if ( $data{$name} ) { $data{$name} .= "\0"; }
   $data{$name} .= $value;
  }

  return( %data );
}


#################################################################
# Checks to ensure that the listed keys are in the
# associative array reference passed in in the first
# argument.
sub FindEmptyFields
{
  my ( $array, @fields ) = @_;
  my ( @empties, $key );

  foreach $key ( @fields )
    { push ( @empties, $key ) unless $$array{$key}; }

  return( @empties );
}
#################################################################
```

- **e−merge.html** − HTML document that explains e−merge. The file is included in this package because you'll need it.

Other Files

- **form1.html** − HTML page containing the form that ad_proc.pl will process >Here's the HTML code that creates it:

```
<HTML>
<HEAD>
<TITLE>Example Form 1</TITLE>
</HEAD>
<BODY BGCOLOR="FFFFFF">
<p align="center">
<br><H2>ORDER FORM:</H2><br>
<FORM method="POST" action="http://www.mysite.com/cgi−bin/ad_proc.pl">
<input type="hidden" name=".required_data"
value="Adtype::Quantity1::Name1::Name2::Email::Phone::Address::City::
State::Zipcode::Country">
<table border="0" width="100%">
<tr><td width="250">
<strong><font face="Arial">
Basic Information:</font></strong></td>
<td width="350"><font face="Arial">
( <strong><font color="#800000">*</font></strong>
 Required Information )</font></td></tr>
<tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>First Name: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Name1" size="30" maxlength="15"></font></td>
</tr><tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>Last Name: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Name2" size="30" maxlength="20"></font></td>
</tr><tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>Address: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Address" size="30" maxlength="60"></font></td>
</tr><tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>City: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="City" size="30" maxlength="40"></font></td>
</tr><tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>State / Province: </b></font>
</td><td width="350"><font face="Arial">
<input type="text" name="State" size="30" maxlength="20"></font></td>
</tr><tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>Zip: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Zipcode" size="30" maxlength="10"></font></td>
```

```
</tr><tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>Country: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Country" size="30" value="USA" maxlength="60">
</font></td></tr>
<tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>Email: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Email" size="30" maxlength="48"></font></td>
</tr><tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>Phone: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Phone" size="30" maxlength="20"></font></td>
</tr><tr><td width="250">
<font face="Arial" color="#800000">
<strong> </strong></font>
<font face="Arial"><b>Company: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Company" size="30" maxlength="60"></font></td>
</tr><tr><td width="250">
<font face="Arial" color="#800000">
<strong> </strong></font>
<font face="Arial"><b>Fax: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Fax" size="30" maxlength="20"></font></td></tr>
<tr><td width="250"><strong>
<font face="Arial">Ad Information:</font>
</strong></td><td width="350"></td></tr>
<tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>Quantity: </b></font></td>
<td width="350"><font face="Arial">
<input type="text" name="Quantity1" size="4"></font></td></tr>
<tr><td width="250">
<font face="Arial" color="#800000">
<strong>*</strong></font>
<font face="Arial"><b>Ad Type:</b></font></td>
<td width="500"><font face="Arial"><b>
<input type="radio" name="Adtype" value="Regular" checked>
Regular Classified Ad
<input type="radio" name="Adtype" value="Sponsor">
Top Sponsor Ad
<input type="radio" name="Adtype" value="Solo">
Solo Ad</b></font></td></tr>
</table>
<div align="center"><table border="0" width="65%">
<tr><td width="100%" height="50"><div align="center"><p><b>
<input type="submit" value="Click Button To Begin Order Processing">
</b></p></div></td></tr>
<tr><td><b>After you press the button, the computer will pause as it
processes your order. Please do not hit the button twice, as it is
normal for the computer to pause for a few seconds as it processes
your order.</b><br><br></td></tr>
<tr><td align="left"><font size="+1">Note: Due to intermittent server
problems, you may sometimes get an "Error 500" screen. If this happens,
simply press your browser's "BACK" button. You'll return here with all
your information intact.  Then click the button again to submit your
```

```
order.</font><br><br></td></tr></table></div>
</FORM>
</p>
</BODY>
</HTML>
```

All the lines that contain **<input>**, as well as certain other tags will collect user input from **text boxes**, **check boxes**, **radio buttons** and **drop−down menus**. These will be sent as a **query string** to the CGI program or script that is the **action** of the form.

Making your forms easy to use insures that more people will use them. Let's be honest, here: People are rushed, overworked, lazy, whatever. You can turn this to your advantage, however. For an input like State or Province, a drop−down menu is often used. It's quicker for someone to select the State from the list than it is to type it in. Right? Not only that, but the chance of an error is greatly reduced. But what's really handy for the CGI part is that you don't have to filter these inputs! You'll always get one of a list of items *you* supplied. The State of West Virginia will *always* be entered as WV, automatically. No misspellings!

Here's the code for a drop−down menu of the US States and Canadian Provinces that uses the standard abbreviations for each one.

```
<SELECT name="state">
<option value="AL">Alabama
<option value="AK">Alaska
<option value="AB">Alberta
<option value="AS">American Samoa
<option value="AZ">Arizona
<option value="AR">Arkansas
<option value="BC">British Columbia
<option value="CA">California
<option value="CO">Colorado
<option value="CT">Connecticut
<option value="DE">Delaware
<option value="DC">District of Columbia
<option value="FL">Florida
<option value="GA">Georgia
<option value="GU">Guam
<option value="HI">Hawaii
<option value="ID">Idaho
<option value="IL">Illinois
<option value="IN">Indiana
<option value="IA">Iowa
<option value="KS">Kansas
<option value="KY">Kentucky
<option value="LB">Labrador
<option value="LA">Louisiana
<option value="ME">Maine
<option value="MB">Manitoba
<option value="MD">Maryland
<option value="MA">Massachusetts
<option value="MI">Michigan
```

```
<option value="MN">Minnesota
<option value="MS">Mississippi
<option value="MO">Missouri
<option value="MT">Montana
<option value="NE">Nebraska
<option value="NV">Nevada
<option value="NB">New Brunswick
<option value="NH">New Hampshire
<option value="NJ">New Jersey
<option value="NM">New Mexico
<option value="NY">New York
<option value="NF">Newfoundland
<option value="NC">North Carolina
<option value="ND">North Dakota
<option value="NS">Nova Scotia
<option value="OH">Ohio
<option value="OK">Oklahoma
<option value="ON">Ontario
<option value="OR">Oregon
<option value="PA">Pennsylvania
<option value="PE">Prince Edward Island
<option value="PR">Puerto Rico
<option value="PQ">Quebec
<option value="RI">Rhode Island
<option value="SK>Saskatchewan
<option value="SC">South Carolina
<option value="SD">South Dakota
<option value="TN">Tennesse
<option value="TX">Texas
<option value="UT">Utah
<option value="VT">Vermont
<option value="VA">Virginia
<option value="WA">Washington
<option value="WV">West Virginia
<option value="WI">Wisconsin
<option value="WY">Wyoming
<option value="YT">Yukon Territory
<option value="OC">Not a US State or a Canadian Province
</SELECT>
```

The CGI script will then examine the **query string** and extract all the inputs as **key/value pairs**. This means that for each input, there will be a **key** with some specific **value**. The keys become variables in the program...symbolic names for bits of memory holding the values.

You determine the names of the keys by what follows 'name=' in the input tags. For example, in the menu input you just saw, you find name="state". So, the **key** for this **value** is "state".

If the visitor selected Ohio, then state=OH. when the script parses the inputs, it will assign the value of state to $State. Now, $State="OH". Anywhere the CGI script needs to know which US state was selected, it can use the value of $State.

- **cgi−lib.pl** − Perl library by Stephen E. Brenner, reproduced here by permission of the author.>

- **colorbar.gif** − a sparkly colored line; nice separator for blocks of text on a web page. Copy and save this file for later use:>