CGI PROGRAMMING 101

Programming Perl for the World Wide Web

Introduction

This book is intended for web designers, entrepreneurs, students, teachers, and anyone who is interested in learning CGI programming. You do not need any programming experience to get started - if you can write HTML, you can also write CGI programs. If you have a web page, and want to write CGIs, then this book is for you.

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

Why learn CGI?
If you're going to create web pages, then at some point you'll want to add a counter, a form to let visitors send you mail or place an order, or something similar. CGI enables you to do that and much more. From mail-forms and counter programs, to the most complex database scripts that generate entire websites on-the-fly, CGI programs deliver a broad spectrum of content on the web today. If you've ever looked at a site such as Amazon.com, DejaNews, or Yahoo, and wondered how they did it... the answer is CGI. CGI experience is also in high demand from employers now; you could substantially improve your salary and job prospects by learning CGI.

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

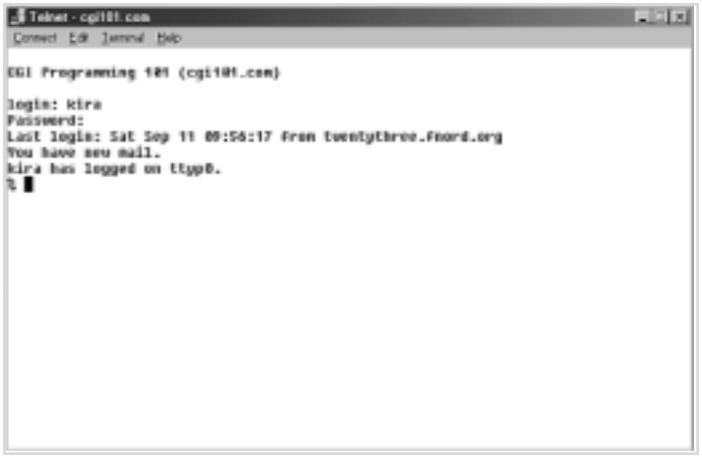
Also included are several appendices that will be invaluable references, including a list of other online Perl resources, CGI job search sites, and tutorials for Unix and password-protection.

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

What do you need to get started?
This book is written for the programmer using Perl on a Unix system. Don't worry if you're not familiar with Unix; this book will teach you all you need to know. Each chapter will show you the Unix commands you need to use with your CGIs, and there's also a Unix command reference and tutorial in Appendix B. If you don't already have access to a Unix account, you can get one from a number of ISPs, including cgi101.com, which offers telnet-only shell accounts and virtual hosting with access to CGIs, CGI helper modules, and a library of ready-to-use scripts. Visit http://www.cgi101.com/ for more information.

If you are using Windows NT instead of Unix, you can still use most of the programs in this book, and learn Perl just as easily. Most NT machines run the same Perl code that Unix machines will. But some of the examples will not work for you, since they are intended for Unix. The Perl Reference for Windows (http://www.perl.com/reference/query.cgi?windows) has some links to websites that can help you get started using CGIs on a Windows system.

You will need a Telnet client to connect to your Unix host of choice. If you are using a Windows PC, you should already have Telnet installed; just go under "Start"->"Run" and type "C:\WINDOWS\telnet.exe". A new telnet window will open. To connect to the Unix host, just pull down the"Remote" menu and select "Connect". Type the host name you want to connect to (such as cgi101.com), press "Connect", and you'll be in. Here's what a typical login window looks like once you connect:



Windows Telnet

If you are using a Mac, there are a number of free or shareware Telnet programs available online, including BetterTelnet (http://www.cstone.net/%7Erbraun/mac/telnet/), dataComet (http://www.databeast.com/), and NiftyTelnet (http://andrew2.andrew.cmu.edu/dist/niftytelnet.html).

If you still need help or info on how to connect or upload your CGIs to the Unix host, visit our website at http://www.cgi101.com/class/connect.html.

All of the code examples in this book are available on the web at http://www.cgi101.com/class/. You can download any or all of them from there, but do try writing the scripts yourself *first;* you'll learn faster that way. The website also includes some related utilities and a library of ready-to-use CGI programs, plus a mailing list you can join to get updates of new material.

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

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

All programs printed in the book are followed by a link to their respective source code:

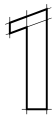
🗗 Source code: http://www.cgi101.com/class/chX/script.txt

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

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

Each chapter also has its own web page at http://www.cgi101.com/class/chX, where X is the chapter number. The full texts of chapters 1-6 are online; other chapters include only an index to the CGI scripts and HTML forms from that chapter.

So, turn to Chapter 1, and let's get started writing CGIs!



Getting Started

Our programming language of choice for this class is Perl. Perl is a simple language, easy to learn, yet powerful enough to accomplish the most difficult tasks. It is widely available, and is probably already installed on your Unix server. Perl is an *interpreted* language, meaning you don't need to compile your script - you simply write your script and run it (or have the web server run it). The script itself is just text code; the Perl interpreter does all the work. The advantage to this is you can copy your script with little or no changes to any machine with a Perl interpreter. The disadvantage is you won't discover any bugs in your script until you run it.

You can edit your Perl scripts either on your local machine (using your favorite text editor - Notepad, Simpletext, etc.), or in the Unix shell. If you're using Unix, try pico - it's a very simple, easy to use text editor. Just type `pico filename` to create or edit a file. Type `man pico` for more information and instructions for pico. If you're not familiar with the Unix shell, see Appendix B for a Unix tutorial and command reference.

The pico text editor

You can also use a text editor on your local machine, and upload the finished scripts to the Unix server. Be sure to use a plain text editor, such as Notepad (on PC) or BBEdit (on Mac), and turn off special characters such as smartquotes - CGI files must be ordinary text. Also, it is imperative that you upload your CGI as text, NOT binary! If you upload it as binary, it will come across with a lot of control characters at the end of the lines, and these will break your script. You can save yourself a lot of time and grief by just uploading *everything* as text (unless you're uploading pictures - for example, GIFs or JPEGs - or other binary data. HTML and Perl CGIs are not binary, they are plain text.)

Once your script is uploaded to the Unix server, you'll want to be sure to move it to your public_html directory (or whatever directory you have set up for web pages). Then, you will also need to change the permissions on the file so that it is "executable" (or runnable) by the system. In Unix, this command is:

```
chmod 755 filename
```

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

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

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

Basics of a Perl Script

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

If you're creating scripts on Unix, you'll need one statement as the first line of every script, telling the server that this is a Perl script, and where to find the Perl interpreter. In most scripts the statement will look like this:

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

For now, there should generally not be anything else on the same line with this statement. (There are some flags you can use there, but we'll go into those later.) If you aren't sure where Perl lives on your system, try typing these commands:

```
which perl    OR    whereis perl
```

If the system can find it, it will tell you the path name to Perl. That path is what you should put in the above statement.

After the above line, you'll write your Perl code. Most lines of Perl code must end in a semicolon (;), except the opening and closing lines of loops and conditional blocks. We'll cover those later.

Let's write a simple first program. Enter the following lines into a new file, and name it "first.pl".

```
#!/usr/bin/perl
print "Hello, world!\n";
```

🗗 Source code: http://www.cgi101.com/class/ch1/first.txt

Save the file. Now, in the Unix shell, you'll need to type:

```
chmod 755 first.pl
```

This changes the file permissions to allow you to run the program. You will have to do this every time you create a new script; however, if you're editing an existing script, the permissions will remain the same and won't need to be changed again.

Now, in the Unix shell, you'll need to type this to run the script:

```
./first.pl
```

If all goes well, you should see it print `Hello, world!` to your screen.

NOTE: This program is not a CGI, and won't work if you try it from your browser. But it's easily changed into a CGI; see below.

Basics of a CGI Script

A CGI program is still a Perl script. But one important difference is that a CGI usually generates a web page (for example: a form-processing CGI, such as a guestbook, usually returns a “thank you for writing” page.) If you are writing a CGI that’s going to generate a HTML page, you must include this statement somewhere in the script, *before* you print out anything else:

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

This is a content header that tells the receiving web browser what sort of data it is about to receive - in this case, an HTML document. If you forget to include it, or if you print something else before printing this header, you’ll get an “Internal Server Error” when you try to access the CGI. A good rule of thumb is to put the Content-type line at the top of your script (just below the `#!/usr/bin/perl/` line).

Now let’s take our original first.pl script, and make it into a CGI script that displays a web page. If you are running this on a Unix server that lets you run CGIs in your public_html directory, you will probably need to rename the file to first.cgi, so that it ends in the .cgi extension. Here is what it should look like:

```
#!/usr/bin/perl

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

print "<html><head><title>Test Page</title></head>\n";
print "<body>\n";
print "<h2>Hello, world!</h2>\n";
print "</body></html>\n";
```

🗗 Source code: http://www.cgi101.com/class/ch1/firstcgi.txt
➪ Working example: http://www.cgi101.com/class/ch1/firstcgi.cgi

Save this file and run it in the Unix shell, like you ran the other one, by typing `"./first.cgi"`. Notice how the script will just print out a bunch of HTML? This is what it should do, BUT, very importantly, if there’s an error in your script, the Perl interpreter will tell you exactly what line the error is on. This is good to remember in the future, because when you’re writing longer, more complex scripts, you may have errors, and the error message you get on the web (500 Server Error) is not at all useful for debugging.

Now let’s call this CGI from your browser. You don’t need to call it from a web page. Just move it into your public_html or CGI-bin directory, and type the direct URL for the CGI. For example:

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

Try it in your own web directory. It should return a web page with the “Hello, world!” phrase on it. (If it doesn’t, see “Debugging a Script,” below.)

Another way to write the above CGI, without using multiple print statements, is as follows:

```
#!/usr/bin/perl

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

print <<EndOfHTML;
<html><head><title>Test Page</title></head>
<body>
<h2>Hello, world!</h2>
</body></html>

EndOfHTML
```

🗗 Source code: http://www.cgi101.com/class/ch1/firstcgi2.txt
➪ Working example: http://www.cgi101.com/class/ch1/firstcgi2.cgi

This is the “here-doc” syntax. Note that there are no spaces between the << and the EndOfHTML, in this statement:

```
print <<EndOfHTML;
```

Also, despite the fact that the script appears indented on this page, each line should start in column 1 in your CGI - *especially* the “EndOfHTML” line that’s by itself. If there’s even one space before the EndOfHTML, you’ll get an error, and your script won’t run.

This manner of displaying HTML will become more useful with future CGIs, because it doesn’t require you to escape embedded quotes, like you would with a normal print statement:

```
print "<a href=\"http://lightsphere.com/\">foo</a>";
```

Note that the quotes around the URL have to be escaped with a backslash in front of them for this to work, since Perl strings are enclosed in “quotes” - the only way to embed another quote inside a quoted string is to escape it, like so: `"John \"Q.\" Public"`.

Debugging a Script

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

If you see the code for the actual Perl script instead of the desired output page from your CGI, this means one of two things: either you didn’t rename the file with the .cgi extension (perhaps you left it named “first.pl”), or your web server isn’t configured to run CGIs (at least not in your directory). You’ll need to ask your webmaster how to run CGIs on your server. And if you ARE the webmaster, check your server’s documentation to see how to enable CGIs in user directories.

If you get an Internal Server Error, there’s a bug in your script. First, try running the CGI from the command line in the Unix shell. The following will check the syntax of your script without actually running it:

```
perl -c scriptname.cgi
```

You might also try the -w flag (for “warnings”), to report any unsafe Perl constructs:

```
perl -cw scriptname.cgi
```

This will report any syntax errors in your script, and warn you of improper usage. For example:

```
% perl -cw urllist2.cgi
syntax error at urllist2.cgi line 9, near "print"
urllist2.cgi had compilation errors.
```

This tells you there’s a problem at or around line 9; make sure you didn’t forget a closing semicolon on the previous line, and check for any other typos. Also be sure you saved and uploaded the file as text - hidden control characters or smartquotes can cause syntax errors, too.

If the `perl -cw` command indicates that your syntax is ok, debugging will take a little more work. This means your script is breaking as it runs; possibly the input data is causing a problem. One way to get more info is to look at the server log files. First you’ll have to find out where the logs are... some usual locations are /usr/local/etc/httpd/logs/error_log, or /var/log/httpd/error_log. Then to view the end of the log, do:

```
tail /var/log/httpd/logs/error_log
```

The last line of the file should be your error message. Here are some example errors from the error log:

```
[Fri Mar 26 02:06:10 1999] access to /home/class/ch1/testy.cgi
failed for 205.188.198.46, reason: malformed header from
script.
In string, @yahoo now must be written as \@yahoo at
/home/class/ch1/testy.cgi line 331, near "@yahoo"
Execution of /home/class/ch1/testy.cgi aborted due to compila-
tion errors.
[Fri Mar 26 10:04:31 1999] access to /home/class/ch1/testy.cgi
failed for 204.87.75.235, reason: Premature end of script head-
ers
```

A "malformed header" or "premature end of script headers" can either mean that you printed something before printing the "Content-type:text/html" line, or your script died. An error usually appears in the log indicating where the script died, as well; in the above example the @-sign in an email address ("@yahoo") wasn't escaped with a backslash. Of course, such an error would also appear if you ran `perl -cw` on the script; this example just shows how it would look in the server log.

Resources

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

Perl Variables

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

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

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

You do not need to declare a variable before using it; just drop it into your code. A scalar can hold data of any type, be it a string, a number, or whatnot. You can also drop scalars into double-quoted strings:

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

Now if you print `$blee`, you will get "The magic number is 23."

Let's edit first.pl again and add some scalars to it:

```
#!/usr/bin/perl
$classname = "CGI Programming 101";
print "Hello there.  What is your name?\n";
$you = <STDIN>;
chomp($you);
print "Hello, $you.  Welcome to $classname.\n";
```

🗗 Source code: http://www.cgi101.com/class/ch2/first.txt

Save, and run the script in the Unix shell. (This program will not work as a CGI.) This time, the program will prompt you for your name, and read your name using the following line:

```
$you = <STDIN>;
```

STDIN is *standard input*. This is the default input channel for your script; if you're running your script in the shell, STDIN is whatever you type as the script runs.

The program will print “Hello there. What is your name?”, then pause and wait for you to type something in. (Be sure to hit return when you’re through typing your name.) Whatever you typed is stored in the scalar variable `$you`. Since `$you` also contains the carriage return itself, we use

```
chomp($you);
```

to remove the carriage return from the end of the string you typed in. The following print statement:

```
print "Hello, $you.  Welcome to $classname.\n";
```

substitutes the value of `$you` that you entered. The “\n” at the end if the line is the perl syntax for a carriage return.

Arrays

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

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

In Perl, array indices start with 0, so to refer to the first element of the array @colors, you use $colors[0]. Note that when you’re referring to a single element of an array, you prefix the name with a $ instead of the @. The $-sign again indicates that it’s a single (scalar) value; the @-sign means you’re talking about the entire array.

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

```
#!/usr/bin/perl
# this is a comment
# any line that starts with a "#" is a comment.

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

print "$colors[0]\n";
print "$colors[1]\n";
print "$colors[2]\n";
```

Or, a much easier way to do this is to use the *foreach* construct:

```
#!/usr/bin/perl
# this is a comment
# any line that starts with a "#" is a comment.

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

foreach $i (@colors) {
    print "$i\n";
}
```

For each iteration of the foreach loop, Perl sets $i to an element of the @colors array - the first iteration, $i is “red”. The braces {} define where the loop begins and end, so for any code appearing between the braces, $i is set to the current loop iterator.

Array Functions

Since an array is an ordered list of elements, there are a number of functions you can use to get data out of (or put data into) the list:

```
@colors = ("red","green","blue","cyan","magenta","black",
"yellow");

$elt = pop(@colors);        # returns "yellow", the last value
                            # of the array.
$elt = shift(@colors);      # returns "red", the first value
                            # of the array.
```

In these examples we’ve set $elt to the value returned, but you don’t have to do that - if you just wanted to get rid of the first value in an array, for example, you’d just shift(@arrayname). Both shift and pop affect the array itself, by removing an element;

in the above example, after you pop "yellow" off the end of @colors, the array is then equal to ("red", "green", "blue", "cyan", "magenta", "black"). And after you shift "red" off the front, the array becomes ("green", "blue", "cyan", "magenta", "black").

You can also add data to an array:

```
@colors = ("green", "blue", "cyan", "magenta", "black");
push(@colors,"orange");      # adds "orange" to the end of the
                             # @colors array
```

@colors now becomes ("green", "blue", "cyan", "magenta", "black", "orange").

```
@morecolors = ("purple","teal","azure");
push(@colors,@morecolors);   # appends the values in @morecolors
                             # to the end of @colors
```

@colors now becomes ("green", "blue", "cyan", "magenta", "black", "orange", "purple", "teal","azure").

Here are a few other useful functions for array manipulation:

```
@colors = ("green", "blue", "cyan", "magenta", "black");
sort(@colors)         # sorts the values of @colors
                      # alphabetically
```

@colors now becomes ("black", "blue", "cyan", "green", "magenta"). Note that sort does not change the actual values of the array itself, so if you want to save the sorted array, you have to do something like this:

```
@sortedlist = sort(@colors);
```

The same thing is true for the `reverse` function:

```
@colors = ("green", "blue", "cyan", "magenta", "black");
reverse(@colors)      # inverts the @colors array
```

@colors now becomes ("black", "magenta", "cyan", "blue", "green"). Again, if you want to save the inverted list, you must assign it to another array.

```
$#colors              # length-1 of the @colors array, or
                      # the last index of the array
```

In this example, `$#colors` is 4. The actual length of the array is 5, but since Perl lists count from 0, the index of the last element is length - 1. If you want the actual length of

the array (the number of elements), you'd use the `scalar` function:

```
scalar(@colors);      # the actual length of the array
```

In this case, `scalar(@colors)` is equal to 5.

```
join(", ",@colors)    # joins @colors into a single
                      # string separated by the
                      # expression ", "
```

@colors becomes a single string: "black, magenta, cyan, blue, green".

Hashes

A hash is a special kind of array - an associative array, or paired group of elements. Perl hash names are prefixed with a percent sign (%), and consist of pairs of elements - a key and a data value. Here's how to define a hash:

```
Hash Name        key              value

%pages   = ("fred",   "http://www.cgi101.com/~fred/",
            "beth",   "http://www.cgi101.com/~beth/",
            "john",   "http://www.cgi101.com/~john/"    );
```

Another way to define a hash would be as follows:

```
%pages = ( fred => "http://www.cgi101.com/~fred/",
           beth => "http://www.cgi101.com/~beth/",
           john => "http://www.cgi101.com/~john/" );
```

The => operator is a synonym for ", ". It also automatically quotes the left side of the argument, so enclosing quotes are not needed.

This hash consists of a person's name for the key, and their URL as the data element. You refer to the individual elements of the hash with a $ sign (just like you did with arrays), like so:

```
$pages{'fred'}
```

In this case, "fred" is the key, and `$pages{'fred'}` is the value associated with that key - in this case, it would be "http://www.cgi101.com/~fred/".

If you want to print out all the values in a hash, you'll need a foreach loop, like follows:

```
foreach $key (keys %pages) {
    print "$key's page: $pages{$key}\n";
}
```

This example uses the `keys` function, which returns an array consisting only of the keys of the named hash. One drawback is that `keys %hashname` will return the keys in random order - in this example, `keys %pages` could return ("fred", "beth", "john") or ("beth", "fred", "john") or any combination of the three. If you want to print out the hash in exact order, you have to specify the keys in the foreach loop:

```
foreach $key ("fred","beth","john") {
    print "$key's page: $pages{$key}\n";
}
```

Hashes will be especially useful when you use CGIs that parse form data, because you'll be able to do things like `$FORM{'lastname'}` to refer to the "lastname" input field of your form.

Let's write a simple CGI using the above hash, to create a page of links:

```
#!/usr/bin/perl
#
# the # sign is a comment in Perl
%pages = ( "fred" => "http://www.cgi101.com/~fred/",
       "beth" => "http://www.cgi101.com/~beth/",
       "john" => "http://www.cgi101.com/~john/" );

print "Content-type:text/html\n\n";
print <<EndHdr;
<html><head><title>URL List</title></head>
<body bgcolor="#ffffff">
<p>
<h2>URL List</h2>
<ul>
EndHdr

foreach $key (keys %pages) {
    print "<li><a href=\"$pages{$key}\">$key</a>\n";
}

print <<EndFooter;
```

```
</ul>
<p>
</body>
</html>
EndFooter
```

🗗 Source code: http://www.cgi101.com/class/ch2/urllist.txt
➪ Working example: http://www.cgi101.com/class/ch2/urllist.cgi

Call it urllist.cgi, save it, and change the permissions so it's executable. Then call it up in your web browser. You should get a page listing each person's name, hotlinked to their actual URL.

Chapter 3 will expand this concept as we look at environment variables and the GET method of posting forms.

Hash Functions

Here is a quick overview of the Perl functions you can use when working with hashes.

```
delete $hash{$key}    # deletes the specified key/value pair,
                      # and returns the deleted value

exists $hash{$key}    # returns true if the specified key exists
                      # in the hash.

keys %hash            # returns a list of keys for that hash

values %hash          # returns a list of values for that hash

scalar %hash          # returns true if the hash has elements
                      # defined (e.g. it's not an empty hash)
```

Resources

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

CGI Environment Variables

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

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

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

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

The %ENV hash is automatically set for every CGI, and you can use any or all of it as needed. For example, if you wanted to print out the URL of the page that called your CGI, you'd do:

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

It's very simple to print out all of the environment variables, and some of the values in the ENV array will be useful to you later, so let's try it. Create a new file, and name it env.cgi. Edit it as follows:

```
#!/usr/bin/perl

print "Content-type:text/html\n\n";
print <<EndOfHTML;
<html><head><title>Print Environment</title></head>
<body>
EndOfHTML

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

print "</body></html>";
```

🗗 Source code: http://www.cgi101.com/class/ch3/env.txt
➪ Working example: http://www.cgi101.com/class/ch3/env.cgi

Save the above CGI, chmod it, and call it up in your web browser. Remember, if you get a server error, you'll want to go back and try running the script at the command line in the Unix shell, to see just where the problem is. (But note, if you run env.cgi in the

shell, you'll get an entirely different set of environment variables.)

In this example we've sorted the keys for the ENV hash so they'll print out alphabetically, using the sort function. Perl's sort function, by default, compares the string value of each element of an array - which means it doesn't work properly for sorting numbers. Fortunately, sorting can be customized. We'll cover numeric and custom sorting in Chapter 8.

A Simple Query Form

There are two ways to send data from an HTML form to a CGI: GET and POST. These *methods* determine how the form data is sent to the server. In the GET method, the input values from the form are sent as part of the URL, and saved in the QUERY_STRING environment variable. With POST, data is sent as an input stream to the program. We'll cover POST in the next chapter, but for now, let's look at the GET method.

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

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

Try opening each of these in your web browser. Notice that the QUERY_STRING is set to whatever appears after the question mark in the URL itself. In the above examples, it's set to "test1", "test2", and "test3", respectively. This can be carried one step further, by setting up a simple form, using the GET method. Here's the HTML for such an example:

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

Create the above form (call it form.html), and call it up in your browser. Type something into the field and hit return. You'll get the same env.cgi output, but this time you'll notice that the query string has two parts. It should look something like:

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

The value on the left is the actual name of the form field. The value on the right is

whatever you typed into the input box, BUT you may notice if you had any spaces in the string you typed, they've been replaced with +. Similarly, various punctuation and other special non-alphanumeric characters are escaped out with a %-code. This is called *URL-encoding*, and it happens with data submitted through either GET or POST methods.

Your Perl script can convert this information back, but it's often easier to use the POST method when sending long or complex data. GET is mainly useful for short, one-field queries, especially for things like database searches.

You can also send multiple input data values with GET:

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

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

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

The values are separated by a &-sign. To parse this, you'll want to split the query string with Perl's `split` function:

```
@values = split(/&/,$ENV{'QUERY_STRING'});
foreach $i (@values) {
    ($varname, $mydata) = split(/=/,$i);
    print "$varname = $mydata\n";
}
```

`split` lets you break up a string into an array of different strings, breaking on a specific character. In the first case, we've split on the &-sign. This gives us two values: "fname=joe" and "lname=smith", which are stored in the array named @values. Then, with a foreach loop, we further split each string on the = sign, and print out the field name and the data that was entered into that field in the form.

Some warnings about GET: it is not at all a secure method of sending data, so don't use it for sending password info, credit card data or other sensitive information. Since the data is passed through as part of the URL, it'll show up in the web server's logfile (complete with all the data), and if that logfile is readable by any user (as most are), you're giving the info away to anyone who might happen to be looking. Private information should always be sent with the POST method, which we'll cover in the next

chapter. (Of course, if you're asking visitors to send sensitive information like credit card numbers, you should also use a secure server, in addition to the POST method.)

GETs are most useful because they can be embedded in a link without needing a form element. This is often used in conjunction with databases, or instances where you want a single CGI to handle a clearly defined set of options. For example, you might have a database of articles, each with a unique article ID. You could write a single article.cgi to serve up the article, and the CGI would simply look at the query string to figure out which article to display. For example, clicking on

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

would display article #22.

Remote Host ID

You've probably seen web pages that greet you with a message like "Hello, visitor from (yourhost)!", where (yourhost) is your actual hostname or IP address. Here is an example of how to do that:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
print <<EndHTML
<html><head><title>Hello!</title></head>
<body>
<h2>Hello!</h2>
Welcome, visitor from $ENV{'REMOTE_HOST'}!<p>
</body></html>
EndHTML
```

🗗 Source code: http://www.cgi101.com/class/ch3/rhost.txt
➪ Working example: http://www.cgi101.com/class/ch3/rhost.cgi

This particular CGI creates a new page, but you'll probably want to use a server-side include (SSI), instead, to embed the information in another page. See Chapter 9 for more on SSIs.

One caveat: this won't work if your server isn't configured to do host name lookups. An alternative would be to display the visitor's IP address:

```
Welcome, visitor from $ENV{'REMOTE_ADDR'}<p>;
```

➪ Working example: http://www.cgi101.com/class/ch3/rhostip.cgi

Last Page Visited

This is a variation on the remote host ID script - only here, we show the last page you visited.

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
print <<EndHTML
<html><head><title>Hello!</title></head>
<body>
<h2>Hello!</h2>
I see you've just come from $ENV{'HTTP_REFERER'}!<p>
</body>
</html>
EndHTML
```

🗗 Source code: http://www.cgi101.com/class/ch3/refer.txt
➪ Working example: http://www.cgi101.com/class/ch3/refer.cgi

The HTTP_REFERER value only gets set when a visitor actually clicks on a link to your page - if they type the URL directly, then HTTP_REFERER is blank.

Checking Browser Type

This script does some pattern-checking to see what browser the visitor is using, and displays a different message depending on browser type.

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
print "<html><head><title>Welcome</title></head>\n";
print "<body>\n";
print "Browser: $ENV{'HTTP_USER_AGENT'}<p>\n";

if ($ENV{'HTTP_USER_AGENT'} =~ /MSIE/) {
   print "You seem to be using <b>Internet Explorer!</b><p>\n";
} elsif ($ENV{'HTTP_USER_AGENT'} =~ /Mozilla/) {
   print "You seem to be using <b>Netscape!</b><p>\n";
} else {
   print "You seem to be using a browser other than Netscape
```

```
or IE.<p>\n";
}

print "</body></html>\n";
```

🗗 Source code: http://www.cgi101.com/class/ch3/browser.txt
➪ Working example: http://www.cgi101.com/class/ch3/browser.cgi

This is a tricky example because IE actually includes “Mozilla” in the browser type line, so we have to try matching “MSIE” first, before matching “Mozilla”. The =~ is a pattern matching operator; it checks to see if `/pattern/` is contained somewhere in the string. You can also use the =~ operator to replace patterns; we’ll see an example of that in the next chapter.

Resources

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

Processing Forms

Most forms you create will send their data using the POST method. POST is more secure than GET, since the data isn't sent as part of the URL, and you can send more data with POST. Also, your browser, web server, or proxy server may cache GET queries, but POSTed data is sent each time. However, since data posted via most forms is often more complex than a single word or two, decoding posted data is a little more work.

Your web server, when sending form data to your CGI, encodes the data being sent. Alphanumeric characters are sent as themselves; spaces are converted to plus signs (+); other characters - like tabs, quotes, etc. - are converted to "%HH" - a percent sign and two hexadecimal digits representing the ASCII code of the character. This is called *URL encoding*. Here's a table of some commonly encoded characters:

Normal Character	URL Encoded String
\t (tab)	%09
\n (return)	%0A
/	%2F
~	%7E
:	%3A
;	%3B
@	%40
&	%26

In order to do anything useful with the data, your CGI must decode these. Fortunately, this is pretty easy to do in Perl, using the *substitute* and *translate* commands. Perl has powerful pattern matching and replacement capabilities; it can match the most complex patterns in a string, using *regular expressions* (see Chapter 14). But it's also quite capable of the most simple replacements. The basic syntax for substitutions is:

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

This command substitutes “pattern” for “replacement” in the scalar variable “$mystring”. Notice the operator is a =~ (an equal sign followed by a tilde) - this is a special operator for Perl, telling it that it’s about to do a pattern match or replacement. Here’s an example of how it works:

```
$greetings = "Hello. My name is xnamex.\n";
$greetings =~ s/xnamex/Bob/;
print $greetings;
```

The above code will print out “Hello. My name is Bob.” Notice the substitution has replaced “xnamex” with “Bob” in the $greetings string.

A similar but slightly different command is the translate command:

```
$mystring =~ tr/searchlist/replacementlist/;
```

This command translates every character in “searchlist” to its corresponding character in “replacementlist”, for the entire value of $mystring. One common use of this is to change the case of all characters in a string:

```
$lowerc =~ tr/[A-Z]/[a-z]/;
```

This results in $lowerc being translated to all lowercase letters. The brackets around [A-Z] denote a class of characters to match.

Decoding Form Data

With the POST method, form data is sent in an input stream from the server to your CGI. To get this data, store it, and decode it, we’ll use the following block of code:

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

Let’s look at each part of this. First, we read the input stream using this line:

```
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
```

The input stream is coming in over STDIN (standard input), and we're using Perl's `read` function to store the data into the scalar variable `$buffer`. You'll also notice the third argument to the `read` function, which specifies the length of data to be read; we want to read to the end of the `CONTENT_LENGTH`, which is set as an environment variable by the server.

Next we split the buffer into an array of pairs:

```
@pairs = split(/&/, $buffer);
```

As with the GET method, form data pairs are separated by & signs when they are transmitted, such as `fname=joe&lname=smith`. Now we'll use a foreach loop to further splits each pair on the equal signs:

```
foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
```

The next line translates every "+" sign back to a space:

```
    $value =~ tr/+/ /;
```

Next is a rather complicated regular expression that substitutes every %HH hex pair back to its equivalent ASCII character, using the pack() function. We'll learn exactly how this works in Chapter 14, but for now we'll just use it to parse the form data:

```
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
```

Finally, we store the values into a hash called %FORM:

```
    $FORM{$name} = $value;
}
```

The keys of %FORM are the form input names themselves. So, for example, if you have three text fields in the form - called name, email-address, and age - you could refer to them in your script by using $FORM{'name'}, $FORM{'email-address'}, and $FORM{'age'}.

Let's try it. Start a new CGI, and name it post.cgi. Enter the following, save it, and chmod it:

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

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

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

print "<html><head><title>Form Output</title></head><body>";
print "<h2>Results from FORM post</h2>\n";

foreach $key (keys(%FORM)) {
    print "$key = $FORM{$key}<br>";
}

print "</body></html>";
```

Source code: http://www.cgi101.com/class/ch4/post.txt

This code can be used to handle almost any form, from a simple guestbook form to a more complex order form. Whatever variables you have in your form, this CGI will print them out, along with the data that was entered.

Let's test the script. Create an HTML form with the fields listed below:

```
<form action="post.cgi" method="POST">
        Your Name:   <input type="text" name="name">
    Email Address:   <input type="text" name="email">
              Age:   <input type="text" name="age">
   Favorite Color:   <input type="text" name="favorite_color">
<input type="submit" value="Send">
<input type="reset" value="Clear Form">
</form>
```

Source code: http://www.cgi101.com/class/ch4/post.html

Enter some data into the fields, and press "send" when finished. The output will be the variable names of these text boxes, plus the actual data you typed into each field.

> Tip: If you've had trouble getting the boxes to align on your form, try putting <pre> tags around the input fields. Then you can line them up with your text editor, and the result is a much neater looking form. The reason for this is that most web browsers use a fixed-width font (like Monaco or Courier) for preformatted text, so aligning forms and other data is much easier in a preformatted text block than in regular HTML. This will only work if your text editor is also using a fixed-width font!
>
> Another way to align input boxes is to put them all into a table, with the input name in the left column, and the input box in the right column.

A Form-to-Email CGI

Most people using forms want the data emailed back to them, so, let's write a form-to-mail CGI. First you'll need to figure out where the *sendmail* program lives on the Unix system you're on. (For cgi101.com, it's in /usr/sbin/sendmail. If you're not sure where yours is, try doing "which sendmail" or "whereis sendmail"; usually one of these two commands will yield the location of the sendmail program.)

Copy your post.cgi to a new file named mail.cgi. Now the only change will be to the foreach loop. Instead of printing to standard output (the HTML page the person sees after clicking submit), you want to print the values of the variables to a mail message. So, first, we must open a *pipe* to the sendmail program:

```
$mailprog = '/usr/sbin/sendmail';
open (MAIL, "|$mailprog -t")
```

The pipe causes all of the ouput we print to that filehandle (MAIL) to be fed directly to the mail program as if it were standard input to that program.

You also need to specify the recipient of the email, with either:

```
$recipient = 'nullbox@cgi101.com';
$recipient = "nullbox\@cgi101.com";
```

Perl will complain if you use an "@" sign inside a double-quoted string or a print <<EndHTML block. You can safely put an @-sign inside a single-quoted string, like `'nullbox@cgi101.com'`, or you can escape the @-sign in other strings by using a backslash. For example, `"nullbox\@cgi101.com"`.

You don’t need to include the comments in the following code; they are just there to show you what’s happening.

```
#!/usr/bin/perl

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

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

# where is the mail program?
$mailprog = '/usr/sbin/sendmail';

# change this to your own email address

$recipient = 'nullbox@cgi101.com';

# this opens an output stream and pipes it directly to the
# sendmail program.  If sendmail can't be found, abort nicely
# by calling the dienice subroutine (see below)

open (MAIL, "|$mailprog -t") or dienice("Can't access
$mailprog!\n");

# here we're printing out the header info for the mail
# message. You must specify who it's to, or it won't be
# delivered:

print MAIL "To: $recipient\n";

# Reply-to can be set to the email address of the sender,
# assuming you have actually defined a field in your form
# called 'email'.

print MAIL "Reply-to: $FORM{'email'} ($FORM{'name'})\n";
```

```
# print out a subject line so you know it's from your form cgi.
# The two \n\n's end the header section of the message.
# anything you print after this point will be part of the
# body of the mail.

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

# here you're just printing out all the variables and values,
# just like before in the previous script, only the output
# is to the mail message rather than the followup HTML page.

foreach $key (keys(%FORM)) {
    print MAIL "$key = $FORM{$key}\n";
}

# when you finish writing to the mail message, be sure to
# close the input stream so it actually gets mailed.

close(MAIL);

# now print something to the HTML page, usually thanking
# the person for filling out the form, and giving them a
# link back to your homepage

print <<EndHTML;
<h2>Thank You</h2>
Thank you for writing.  Your mail has been delivered.<p>
Return to our <a href="index.html">home page</a>.
</body></html>
EndHTML

# The dienice subroutine, for handling errors.
sub dienice {
    my($errmsg) = @_;
    print "<h2>Error</h2>\n";
    print "$errmsg<p>\n";
    print "</body></html>\n";
    exit;
}
```

Source code: http://www.cgi101.com/class/ch4/mail.txt

Now let's test the new script. Here's the form again, only the action this time points to mail.cgi:

```
<form action="mail.cgi" method="POST">
        Your Name: <input type="text" name="name">
   Email Address: <input type="text" name="email">
             Age: <input type="text" name="age">
  Favorite Color: <input type="text" name="favorite_color">
<input type="submit" value="Send">
<input type="reset" value="Clear Form">
</form>
```

➪ Working example: http://www.cgi101.com/class/ch4/mail.html

Save it, enter some data into the form, and press "send". If the script runs successfully, you'll get email in a few moments with the results of your post. (Remember to change the $recipient in the form to your email address!)

Sending Mail to More Than One Recipient

What if you want to send the output of the form to more than one email address? Simple: just add the desired addresses to the $recipients line:

```
$recipient = 'kira@cgi101.com, kira@io.com,
webmaster@cgi101.com';
```

Subroutines

In the above script we used a new structure: a subroutine called "dienice." A subroutine is a block of code, separate from the main program, that only gets run if it's directly called. In the above example, dienice only runs if the main program can't open sendmail. Rather than aborting and giving you a server error (or worse, NO error), you want your script to give you some useful data about what went wrong; dienice does that, by printing the error message and closing html tags, and exiting from Perl. There are several ways to call a subroutine:

```
&subname;
&subname(args);
subname;
subname(args);
```

The &-sign before the subroutine name is optional. `args` are values to pass into the subroutine.

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

```
sub subname {
    ...code to execute...
}
```

A subroutine can be placed anywhere in your CGI, though for readability it's usually best to put them at the end, after your main code. You can also include and use subroutines from different files and modules; we'll cover that more in Chapter 17.

You can pass data into your subroutines. For example:

```
mysub($a,$b,$c);
```

This passes the scalar variables $a, $b, and $c to the `mysub` subroutine. The data being passed (called *arguments*) is sent as a list. The subroutine accesses the list of arguments via the special array "@_". You can then assign the elements of that array to special temporary variables, like so:

```
sub mysub {
    my($tmpa, $tmpb, $tmpc) = @_;
    ...code to execute...
}
```

Notice the *my* in front of the variable list? `my` is a Perl function that limits the scope of a variable or list of variables to the enclosing subroutine. This keeps your temporary variables visible only to the subroutine itself (where they're actually needed and used), rather than to the entire script (where they're not needed).

We'll be using the dienice subroutine throughout the rest of the book, as a generic catch-all error-handler.

Resources

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

Advanced Forms

In the last chapter you learned how to decode form data, and mail it to yourself. However, one problem with the previous script is that it doesn't have any error-checking or specialized processing. You might not want to get blank forms, or you may want to require certain fields to be filled out. You might also want to write a quiz or questionnaire, and have your script take different actions depending on the answers. All of these things require some more advanced processing of the form data.

All that's required here is to know how to test conditions in Perl. Probably the main one you'll use in a form-handling script is the if-elsif condition:

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

The `elsif` and `else` blocks are optional; if you are only testing whether a particular variable is true or not, you can just use a single if block:

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

In Perl there are different conditional test operators, depending on whether the variable you want to test is a string or a number:

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

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

Let's try it. Copy your mail.cgi to a new script called mail2.cgi, and insert this conditional test before the open(MAIL) statement:

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

This condition takes advantage of the dienice subroutine we wrote before. Now, if you submit a blank form, you'll get the error message.

You can extend this to test for multiple fields at the same time:

```
if ($FORM{'name'} eq "" or $FORM{'email'} eq "" or
$FORM{'age'} eq "") {
    dienice("Please fill out the fields for your name, age,
and email.");
}
```

The above code will return an error if any of the name, email, or age fields are left blank. The conditions are separated by the or operator (which may also be written as ||) - if any of (test1 or test2 or test3) is true, then the condition is met.

Handling Checkboxes

You may want to include checkboxes in your form, to allow the viewer to select one or more options. But how do you decode these inside your CGI?

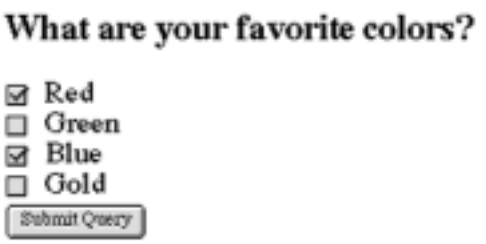
If you just want to display them in your email message, you can just print them like you

would any text field; each checkbox has a different name. Open a new HTML file, and call it colors.html. Enter the following form:

```
<html><head><title>colors</title></head>
<body>
<form action="colors.cgi" method="POST">

<h3>What are your favorite colors?</h3>
<input type="checkbox" name="red" value=1> Red<br>
<input type="checkbox" name="green" value=1> Green<br>
<input type="checkbox" name="blue" value=1> Blue<br>
<input type="checkbox" name="gold" value=1> Gold<br>

<input type="submit">

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

Source code: http://www.cgi101.com/class/ch5/colors.html

Here's how it looks on the screen:



This example lets the visitor pick as many options as they want - or none, if they prefer. While you can set the value=″whatever” part of the checkbox field to any value you want, if you use integers, it will mean less code inside the CGI.

Now let’s write the CGI to process the above form. Call it colors.cgi:

```
#!/usr/bin/perl

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

read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
```

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

print "<html><head><title>Results</title></head>\n";
print "<body>\n";
print "<h2>Results</h2>\n";

@colors = ("red","green","blue","gold");
foreach $x (@colors) {
    if ($FORM{$x} == 1) {
        print "You picked $x.\n";
    }
}

print "</body></html>\n";
```

Source code: http://www.cgi101.com/class/ch5/colors.txt

NOTE: if you used value=1 in your form, then your CGI can test it with “if ($FORM{$x} == 1)”. But if you put quotes around the 1, such as value=“1”, then your CGI will not work unless you change the == operator to an eq:

```
if ($FORM{$x} eq "1") {
    ...do whatever
}
```

Handling Radio Buttons

Radio buttons differ from checkboxes in that you can have several buttons that share the same field name in the form itself - thus allowing the viewer to only select one of a series of options. To distinguish each option, the buttons themselves must have different values.

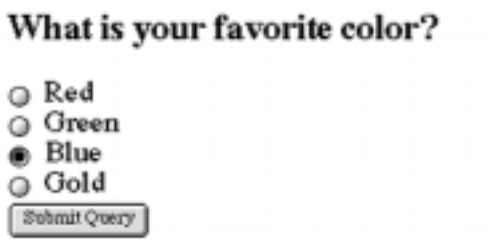
Let’s try it. Take your colors.html file, and copy it to colors2.html. Then edit it and replace the checkbox fields with the following:

```
<h3>What is your favorite color?</h3>
```

```
<input type="radio" name="color" value="red"> Red<br>
<input type="radio" name="color" value="green"> Green<br>
<input type="radio" name="color" value="blue"> Blue<br>
<input type="radio" name="color" value="gold"> Gold<br>
```

Source code: http://www.cgi101.com/class/ch5/colors2.html

Here's how it looks on the screen:



This is similar to the checkboxes form. However, in this case, each radio button has the same field name, but a different value. It's easiest to set the value to be relevant to the name of the thing being picked - in this case the values are set to the name of the colors themselves. Radio buttons can be handled in Perl fairly simply. Copy your colors.cgi to colors2.cgi, and replace the foreach loop with the following line:

```
print "Your favorite color is: $FORM{'color'}<br>\n";
```

Source code: http://www.cgi101.com/class/ch5/colors2.txt

You see here why it is best to set the value to something meaningful - this lets you just print out the radio button and its value, without having to also store another list inside your CGI to show what each button means.

Let's take this one step further. Say you not only want to tell the viewer what color they picked, but you also want to show it to them. Edit your colors2.cgi and replace the existing print statements with the following code:

```
%colors = ( "red"   => "#ff0000",
            "green" => "#00ff00",
            "blue"  => "#0000ff",
            "gold"  => "#ffcc00");

print "<html><head><title>Colors</title></head>\n";
```

```
print "<body bgcolor=\"$colors{$FORM{'color'}}\">\n";
print "<h2>Your favorite color is: $FORM{'color'}</h2><br>\n";
print "</body></html>";
```

Source code: http://www.cgi101.com/class/ch5/colors2a.txt

The above actually sets the background color to whatever color you picked. It's using a hash called %colors, whose keys are the same as the data value of the radio buttons in the form itself. The values of the hash are hex codes for those colors.

Handling SELECT Fields

Select fields may be handled almost exactly the same as radio buttons. A select field, in your HTML page, is a pull-down menu like this:

The HTML to generate the above is as follows:

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

As with radio buttons, you can print out the selection just as it was made in the form:

```
print "Your favorite color is: $FORM{'color'}<br>\n";
```

This will only work if your select statement is a single-value select (that is, your visitor can't select more than one item from the selection list). For handling multi-value selects, see below.

Multiple-choice Selects

The reason our above form-handling code doesn't handle multi-choice selects is that we're assigning a single $value to each field name from the form:

```
$FORM{$name} = $value;
```

If you have a multiple select, like this:

```
<select name="cities" multiple size=3>
<option>Dallas
<option>Houston
<option>Seattle
<option>Portland
<option>Denver
</select>
```

This code allows the user to pick more than one city from the list:

Then when your form data is passed on to the CGI, you have several instances of cities="cityname". So if $FORM{'cities'} already exists and has a value in it, and you send another value, the old value just gets overwritten by the new one.

The solution to this is to handle the multi-selects differently, by storing them in their own array, rather than in the $FORM hash. Here's how you might do it:

```
@cities = ();

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

```
            $FORM{$name} = $value;
        }
    }
```

🗗 Source code: http://www.cgi101.com/class/ch5/mform.txt
➪ Working example: http://www.cgi101.com/class/ch5/mform.html

Now you can use the @cities array for any special processing of the multi-field city data. Everything else is stored in %FORM.

Survey Form and CGI

Let's take what you've learned so far and put it to practical use: a survey form and its corresponding CGI. This type of form adds interactivity to your site, whether you're doing a one question poll-of-the-day, or a lengthy survey of your site's readers.

Create a new HTML form, and name it survey.html. Enter the following form:

```
<html><head><title>Survey</title></head>
<body>
<h2>Survey</h2>
<form action="survey.cgi" method="POST">

Enter your name: <input type="text" name="name" size=30><p>

Your email address: <input type="text" name="email" size=30><p>

How did you reach this site? <select name="howreach">
<option value=0 selected>Choose one...
<option value=1>Typed the URL directly
<option value=2>Site is bookmarked
<option value=3>A search engine
<option value=4>A link from another site
<option value=5>From a book
<option value=6>Other
</select><p>

How would you rate the content of this site?<br>
Poor <input type="radio" name="rating" value=1> 1
<input type="radio" name="rating" value=2> 2
<input type="radio" name="rating" value=3> 3
```

```
<input type="radio" name="rating" value=4> 4
<input type="radio" name="rating" value=5> 5 Excellent<p>

Are you involved in any of the following? (check all that
apply):<br>
<input type="checkbox" name="des" value=1> Website Design<br>
<input type="checkbox" name="svr" value=1> Web Server
Administration<br>
<input type="checkbox" name="com" value=1> Electronic
Commerce<br>
<input type="checkbox" name="mkt" value=1> Web
Marketing/Advertising<br>
<input type="checkbox" name="edu" value=1> Web-related
Education<br>
<p>

Any other comments?<br>
<textarea name="comments" rows=5 cols=70 wrap="VIRTUAL">
</textarea>
<p>

<input type="submit">
</form>
</body></html>
```

Save it. Now create survey.cgi:

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

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

# Since the "how you reached this site" list was saved as
# a number, we need a hash to translate it back to English:
%howreach = ( 0 => "",
              1 => "Typed the URL directly",
```

```
                2 => "Site is bookmarked",
                3 => "A search engine",
                4 => "A link from another site",
                5 => "From a book",
                6 => "Other" );

print <<EndHTML;
<html><head><title>Results</title></head>
<body>
<h2>Results</h2>

Here's what you entered:<p>
Your name: $FORM{'name'}<p>
Email: $FORM{'email'}<p>
How you reached this site: $howreach{$FORM{'howreach'}}<p>
How you'd rate this site (1=poor,5=excellent):
$FORM{'rating'}<p>
EndHTML

%boxes = ( "des" => "Website Design",
       "svr" => "Web Server Administration",
       "com" => "Electronic Commerce",
       "mkt" => "Web Marketing/Advertising",
       "edu" => "Web-Related Education" );

print "You're also involved in the following:<br>\n";
foreach $key (keys %boxes) {
    if ($FORM{$key} == 1) {
        print "$boxes{$key}<br>\n";
    }
}

print <<EndFoot;
<p>
Your comments:<br>
$FORM{'comments'}<p>
</body></html>
EndFoot
```

🗗 Source code: http://www.cgi101.com/class/ch5/survey.txt
➪ Working example: http://www.cgi101.com/class/ch5/survey.html

Save and chmod it, and try filling out the survey in your browser. This example posts

the results to the page, but you can just as easily email yourself a copy of the results, using the mail code in chapter 4. In this case all you need to change, after the MAIL handle is open, is the print <<EndHTML statements, to:

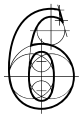
```
print MAIL <<EndHTML;
```

This will redirect the subsequent text to the mail message, rather than standard output.

It's more likely you'll want to write the data to a file, however, so you can analyze the results later. We'll cover reading and writing files next.

Resources

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



Reading and Writing Data Files

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

Most web servers run with very limited permissions; this protects the server (and the system it's running on) from malicious attacks by users or web visitors. Unfortunately this means it's harder to write to files; when your CGI is run, it's run with the server's permissions, and it's likely the server doesn't have permission to create files in your directory. In order to write to a data file, you must usually make it *world-writable*, via the chmod command:

```
chmod 666 myfile.dat
```

This sets the permissions so that all users can read from and write to the file. (If you want the file to be readable by you, but write-only by all other users, do "chmod 622 filename".) See Appendix A for a chart of the various chmod permissions.

The bad part about this is, it means that anyone on your system can go in and change your data file, or even delete it, and there's not much you can do about it.

Some alternatives are CGIwrap and Apache's suEXEC; both of these force CGIs on the web server to run under the CGI owner's userid and permissions. Apache also allows the webmaster to define what user and group the server - including virtual hosts - runs under. If your site is a virtual host, ask your webmaster to set your server up under a group that only you are a member of. Then you can chmod your files one of these ways:

```
chmod 664 myfile.dat  # group read-write, world read
```

```
chmod 660 myfile.dat # group read-write, world none
```

This is safer than having a world-writable file. Ask your webmaster how you can secure your data files.

Permissions are less of a problem if you only want to read a file; just set the file group- and world-readable, and your CGIs can safely read from that file.

Opening Files

Reading and writing files is done by opening a filehandle, with the statement:

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

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

```
open(INF,"mydata.txt");      # opens mydata.txt for reading
open(OUTF,">outdata.txt");   # opens outdata.txt for overwriting
open(OUTF,">>outdata.txt");  # opens outdata.txt for appending
```

The filehandles in these cases are INF and OUTF. You can use just about any name for the filehandle, but for readability, it's always good to name it something relevant.

Also, a warning: your web server might do strange things with the path your CGI runs under, so it's possible you'll have to use the full path to the file (such as "/home/you/public_html/somedata.txt"), rather than just the filename. This is generally not the case with the Apache web server, but some other servers behave differently. Try opening files with just the filename first (provided the file is in the same directory as your CGI), and if it doesn't work, then use the full path.

One problem with the above code is that it doesn't check to ensure the file was really opened. The safe way to open a file is as follows:

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

This uses the "dienice" subroutine we wrote in chapter 4 to display an error message and exit the CGI if the file can't be opened. You should do this for all file opens, because if you don't, the CGI will continue running even if the file isn't open, and you could end up losing data. It can be quite frustrating to realize you've had a survey run-

ning for several weeks while no data was being saved to the output file.

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

Let's test it out, by modifying our survey.cgi from chapter 5 to write to a data file. Edit survey.cgi as follows:

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

read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
    $value =~ s/\n/ /g;      # replace newlines with spaces
    $value =~ s/\r//g;       # remove hard returns
    $value =~ s/\cM//g;      # delete ^M's
    $FORM{$name} = $value;
}

open(OUTF,">>survey.out") or dienice("Couldn't open survey.out
for writing: $!");

# This locks the file so no other CGI can write to it at the
# same time...
flock(OUTF,2);
# Reset the file pointer to the end of the file, in case
# someone wrote to it while we waited for the lock...
seek(OUTF,0,2);

print OUTF "$FORM{'name'}|$FORM{'email'}|";
print OUTF "$FORM{'howreach'}|$FORM{'rating'}|";

%boxes = ( "des" => "Website Design",
           "svr" => "Web Server Administration",
           "com" => "Electronic Commerce",
           "mkt" => "Web Marketing/Advertising",
           "edu" => "Web-Related Education" );
```

```
foreach $key (keys %boxes) {
    if ($FORM{$key} == 1) {
        print OUTF "$key,";
    }
}

print OUTF "|$FORM{'comments'}\n";
close(OUTF);

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

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

Source code: http://www.cgi101.com/class/ch6/survey.txt

Next you'll need to create the output file and make it writable, because your CGI probably can't create new files in your directory (unless you made the entire directory writable by the server - but that's usually a bad idea, since it means anyone can delete any file in that directory, or add new files). Go into the Unix shell, change to the directory where your CGI is located, and type the following:

```
touch survey.out
chmod a+w survey.out
```

The Unix `touch` command, in this case, creates a new, empty file called survey.out. Then the chmod makes it writable by everyone.

Now go back to your browser and fill out the survey form again. If your CGI runs without any errors, you'll get data added to the survey.out file. The resulting file should look something like this:

```
You|you@any.com|A link from this site|5|des,svr,com|This site
rocks!
```

This is what's called a flat-file database - a text file containing data, with each line of the file being a new record (or one set of results from a form) in the database. In this example, we've separated the fields with the pipe symbol (vertical bar "|"), though you could use any character that will not appear in the data itself.

Notice a few new things in the above code. First, the following lines:

```
$value =~ s/\n/ /g;  # replace newlines with spaces
$value =~ s/\r//g;   # remove hard returns
$value =~ s/\cM//g;  # delete ^M's
```

strip out all carriage returns and other strange end-of-line characters from the incoming data. Since we want each line of data in the output file to represent one record, we don't want extraneous carriage returns messing things up. Also you'll notice we've done several "print" statements to the output file, but it only resulted in a single line of data printed to the output file. This is because a line of output isn't really ended until you print out the "\n" character. So you can do:

```
print "Foo ";
print "Bar ";
print "Blee\n";
```

and the resulting output will be

```
Foo Bar Blee
```

Since the \n only appears after "Blee".

File Locking

CGI processes on a Unix web server can run simultaneously, and if two scripts try to open and write the same file at the same time, the file may be erased, and you'll lose all of your data. To prevent this, we've used `flock(OUTF,2)` in the survey.cgi to exclusively lock the survey file while we are writing to it. (The 2 means exclusive lock.) The lock will be released when your script finishes running, allowing the next CGI to access the file. This is only effective if all of the CGIs that read and write to that file also use `flock`; without `flock`, the CGI will ignore the locks of any other process and open/write/erase the file.

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

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

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

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

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

Note that `flock` is not supported on all systems (definitely not on Windows), so if you get an error in your script due to the `flock`, just comment it out. Of course, without the lock, you risk losing data; you can either accept that risk, or look at Chapter 18 and find out how to write your data to a database instead of a file.

Closing Files

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

```
close(filehandle);
```

Files are automatically closed when your script ends, as well.

Reading Files

After you've run a survey or poll like our previous example, you'll want to summarize the data. All that's involved is opening your data file, reading every record, and doing whatever calculations or summarizations you want to do on it.

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

```
open(INF,"survey.out") or dienice("Can't open survey.out: $!");

$a = <INF>;       # reads one line from the file into
```

```
                    # the scalar $a
@b = <INF>;         # reads the ENTIRE FILE into array @b

close(INF);
```

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

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

```
open(INF,"survey.out") or dienice("Can't open survey.out: $!");
@ary = <INF>;
close(INF);

foreach $line (@ary) {
    chomp($line);
    print "$line\n";
}
```

This code minimizes the amount of time the file is actually open. Throughout the rest of the book, we'll be using this method of reading files.

Back to our survey. Say we'd like to summarize the data: how many people took the survey; how most people reached the site; the average rating for the site; counts on how many people are involved in the various areas of webmastering; and a list of user comments.

Let's try it. Create a new file and name it surveysumm.cgi. This script will read the file into an array, then loop through each element of the array, incrementing several counters. At the end, it prints a web page that summarizes the data:

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

open(INF,"survey.out") or dienice("Couldn't open survey.out for
reading: $! \n");
@data = <INF>;
close(INF);

# First we initialize some counters and hashes for storing the
```

```
# summarized data.
$count = 0;
$ratings = 0;
$commentary = "";
%howreach_counts = ();
%involved = ();

%howreach = ( 0 => "",
              1 => "Typed the URL directly",
              2 => "Site is bookmarked",
              3 => "A search engine",
              4 => "A link from another site",
              5 => "From a book",
              6 => "Other" );

foreach $i (@data) {
    chomp($i);
    ($name,$email,$how,$rating,$boxes,$comments) =
split(/\|/,$i);

# this is the same as $count = $count + 1;
    $count++;

    $ratings = $ratings + $rating;

# the following appends "$comments\n" to the end of the
# $commentary string.  The .= construct is just a way to
# concatenate strings.
   $commentary .= "$comments\n";

    $howreach_counts{$how}++;

    @invlist = split(/,/,$boxes);
    foreach $j (@invlist) {
        $involved{$j}++;
    }
}

$avg_rating = int($ratings / $count);

# Now we can print out a web page summarizing the data.
print <<EndHTML;
```

```
<html><head><title>Survey Results</title></head>
<body>
<h2 align="CENTER">Survey Results</h2>

Total visitors: $count<p>

Average rating for this site: $avg_rating<p>

How people reached this site:<br>
<ul>
   <li>(did not answer) - $howreach_counts{0}
   <li>$howreach{1} - $howreach_counts{1}
   <li>$howreach{2} - $howreach_counts{2}
   <li>$howreach{3} - $howreach_counts{3}
   <li>$howreach{4} - $howreach_counts{4}
   <li>$howreach{5} - $howreach_counts{5}
   <li>$howreach{6} - $howreach_counts{6}
</ul>

Involvement in <br>
<ul>
   <li>Website Design: $involved{'des'}
   <li>Web Server Administration: $involved{'svr'}
   <li>Electronic Commerce: $involved{'com'}
   <li>Web Marketing/Advertising: $involved{'mkt'}
   <li>Web-Related Education: $involved{'edu'}
</ul><p>

Comments:<p>
$commentary
EndHTML

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

🗗 Source code: http://www.cgi101.com/class/ch6/surveysumm.txt
➪ Working example: http://www.cgi101.com/class/ch6/surveysumm.cgi.

You'll notice that this summary CGI is actually longer than the script that handled the

survey form; summarizing data from polls can often be a lengthy and complicated process. Also, our summary CGI doesn't do anything with the names or e-mail addresses of the people taking the survey; you may want to write a second CGI to dump those to another file, which you could then use for sending followup mail to your survey respondents.

A survey is just one use for data files. You can use this same code to hold contests, solicit suggestions, populate a mailing list, or for other interactive applications. Flat-file databases can also be used for generating online catalogs; we'll cover that in the next chapter, along with multi-CGI interaction.

Resources

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

Order *CGI Programming 101* Today!

The rest of the book is packed with information and examples of more advanced CGI use:

Chapter 7: Multi-Script Forms - passing data between scripts; a flat-file catalog database and multi-part order form
Chapter 8: Searching & Sorting
Chapter 9: Using Server-Side Includes - SSI Syntax, list of SSI elemenets, how to include files and execute CGIs; SSI page counter; SSI error logger
Chapter 10: Randomness - a random image picker; random password generator; random rotating ad banners
Chapter 11: Redirects and Refreshes - how to send visitors to another page
Chapter 12: Working With Strings - comparing, finding, joining, and formatting strings
Chapter 13: Date and Time in Perl - how to generate the date; countdown/up clocks
Chapter 14: Regular Expressions - pattern matching with Perl
Chapter 15: HTTP Cookies - how to set and read cookies
Chapter 16: Writing Secure Scripts - taint checking; protecting your data
Chapter 17: Beyond Scripts: Perl Modules - CGI.pm, GD.pm; where to find modules
Chapter 18: Database Programming - an introduction to MySQL and DBI; an SQL-based catalog script; SQL page counter
Chapter 19: Writing Your Own Modules
Chapter 20: Working With Unix
Appendix A: Online Resources
Appendix B: Unix Tutorial and Command Reference
Appendix C: Finding CGI Jobs
Appendix D: Password Tutorial

You can order online at http://www.cgi101.com/class/order.html,
or print out the form on the following page and mail it.

CGI PROGRAMMING 101

If you'd like to pay by check or money order, just print this page and mail it to:

CGI Programming 101
PO Box 891174
Houston TX 77289-1174

Please make checks payable to "CGI101.COM".

If you prefer to pay by credit card, please use our secure order form at https://secure.cgi101.com/class/order2.html.

Name: _____

Address: _____

_____

Phone: _____

Email: _____

Please send _____ copies of CGI Programming 101 at $24.95 ea.

Shipping:

U.S. First Class Mail - $3 for first book, $0.50 each additional book

All foreign orders (including Canada/Mexico), airmail - $8 first book, $3 each additional book.

Texas residents add 8.25% sales tax.

Total Enclosed: $_____