CHAPTER 4

Using Libraries

OVERVIEW

After digesting the previous chapters, you should be comfortable finding scripts, downloading them to your local CGI scripts directory, and configuring them to run on your intranet or Internet server. This is a great start, but there is only so much you can do with other people's scripts without being able to modify them.

Before you know it, your boss or clients will be clamoring for various customizations. Perhaps they will want different graphics, perhaps some new features, or maybe an entirely new look and feel for the graphical user interface (GUI). In other words, most likely the scripts you download will not be configured the way you want them to be set up. In fact, any well-designed script should be as generic (read "boring") as possible so that it can be ported to as many different environments as possible. It is up to the local programmer to add the bells and whistles that turn a generic Perl script into a dynamic Web application.

The next step in your evolution as a CGI programmer is to learn how to customize the scripts for your own needs. The first task is to understand how CGI programs are structured. Each of the scripting chapters in this book includes a design discussion, and many of our programs also use a common toolbox of routines—called *libraries*—that are included in every Web application. Although subsequent chapters will discuss each of the major libraries that we use in our Web programming, it is helpful to understand how libraries work.

Web applications are not ferocious. Certainly, in the beginning, a thousand lines of code can look daunting when seen as a whole—seething with what seems like endlessly intertwined loops and variables and arrays and subroutines. However, you should not let that stop you from studying the program. After a bit of delving into the lines of code, you'll often find that a well-written program can be tame. It's a good idea to view programs as a group of associated algorithms (or routines), all of which have small, well-defined functions. Algorithms are like ants in an ant colony, each doing one small job well. To decipher a program, you need only focus on understanding how these simple packets of code, called *subroutines*, interact. With practice, they will call out to you: "I add numbers," "I gather form input," "I parse that input," "I say hello world when asked." If you understand the program one routine at a time, the application's design will emerge and you will be more comfortable modifying and adding to the original program.

Design Discussion

There are three types of algorithms that you will be faced with in most Perl CGI applications: individual algorithms, application-specific subroutines, and interapplication libraries. Let's look at each of these types.

Individual Algorithms

Algorithms are merely pieces of code that perform an action in a specific way. An algorithm that adds two numbers can be expressed as x + y.

However, its usage is very specific. To add 31,289 and 23,990, you would use the following Perl code:

```
$sum = 31289 + 23990;
```

This is a simple routine, but it is also specific to just one case. Programs typically consist of many such routines put together. However, there comes a time when you will want to make the routine generic so that you can call it over and over again in other parts of the program without having to rewrite it. This is where application-specific subroutines come in.

Application-Specific Subroutines

Routines that are general enough that they are used several times in the same application should usually be placed in a subroutine. A subroutine encapsulates a routine so that other parts of the program can refer to it by its subroutine name. Consider the addition algorithm; what if we needed to add various numbers several times in our program but did not want to create a separate piece of code for each instance of addition? If the algorithm were four or five lines long, it would be annoying to type the similar lines over and over again. Even worse, if you ended up changing the logic of the algorithm you would have to hunt down every occurrence and change each one. Maintaining such a program could become a nightmare, because many errors could arise if you forgot to change one of the lines of code in any of the duplicate routines or changed one of them incorrectly.

When faced with such a circumstance, a programmer can create a subroutine that can be used again and again by other parts of the program. Figure 4.1 depicts how one subroutine could be used by several algorithms.

To create and use subroutines in Perl, we need three things: a subroutine reference, a subroutine identifier, and a parameter list of variables to pass to the subroutine. The `&` symbol precedes the name of the routine, telling Perl to look for the subroutine and call it. For example, `&AddNumbers` would direct Perl to execute the `AddNumbers` subroutine.

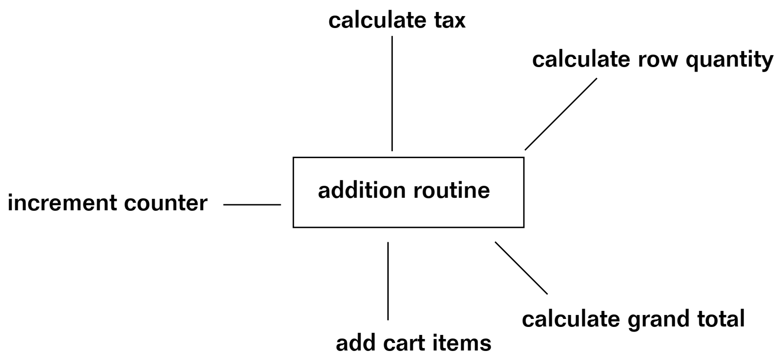

Figure 4.1 *Using a subroutine.*

We also need to be able to send the subroutine some information. Specifically, we need to send the subroutine parameters that it will use to customize its output. If we want to add 2 and 3, for example, we pass 2 and 3 to the subroutine using the following format:

```
&AddNumbers(2,3);
```

The `&` marker tells Perl to look for the subroutine in the program in order to call it. However, the *definition* of the subroutine is marked off in the program using a `sub` marker. The code belonging to the routine is delimited with curly brackets (`{}`). The following example shows what the `AddNumber` subroutine definition would look like:

```
sub AddNumbers
  {
   local($first_number, $second_number) = @_;
   print $first_number + $second_number;
  }
```

Note the third line above. We use the `local` keyword to make sure that the `$first_number` and `$second_number` variables will be considered local to only that subroutine. The subroutine will not affect any other variables that may be called `$first_number` or `$second_number` in other subroutines within the program. In Perl, the `@_` is a list of parameters that have been passed to the function. `$first_number` is set equal to the first parameter, and `$second_number` is set equal to the second parameter in the `@_` list of parameters. If the routine is called by `&AddNumbers(2,3)`, 2 and 3 will be assigned to `$first_number` and `$second_number`, respectively.

NOTE

It is important to use local variables within subroutines so that they do not overwrite variables used by the main script. In complex scripts that use dozens of variables, you may easily forget which variables you are using. Using local variables ensures that if you end up using the same name for a variable, you keep them separated.

Whenever you want to add numbers, you can simply use the subroutine call `&AddNumbers(x,y)` instead of writing each addition individually. As a bonus, if you need to change the logic of the addition algorithm, you need only change it in the subroutine.

Interapplication Libraries

Good design does not stop with the use of subroutines. Often, several different scripts will be designed to incorporate the use of similar routines. In this case, it makes sense to remove the common routines from the programs and place them in a separate file of routines. This file can then be loaded as a library of subroutines into each program as needed. For example, in CGI, most applications will need a form gathering and parsing routine, a template for sending out the HTTP header, and perhaps one to generate template HTML code, such as the following:

```
<HTML><HEAD><TITLE>My Script Title</TITLE></HEAD><BODY>
```

In this case, we use library files and `require` them from the main script. A library file in Perl is simply a text file containing subroutines that are shared by several different Perl scripts. For these library files to be usable by the program, they must be readable by the script and must be in the Perl library path (or its location must be explicitly referenced). For example, if we wanted to load Steven Brenner's **cgi-lib.pl** library into our script, we would use the following:

```
require "cgi-lib.pl";
```

When this is done, every subroutine in **cgi-lib.pl** becomes accessible to the main script as if it were actually written into the script's code. We simply reference a subroutine contained in **cgi-lib.pl** as we would any other subroutine in the main program.

NOTE

Library files need to be readable by the script that requires them. If your server is running as "Nobody" (see Chapter 1 on setting up script permissions), you may need to make the library files readable by the world.

CONCLUSION

As a preparation for the chapters to come, the concepts of subroutines and libraries are crucial to understanding the programs in this book. Many of our CGI scripts, both large and small, take strategic advantage of these libraries. The subsequent chapters in this section explain the internal workings of the Perl libraries we use and discuss how to use the routines in your own programs. The libraries discussed in this book are extremely useful in gaining an awareness of how our CGI scripts were built.