1998

Creating an interactive web-based model course using the Common Gateway Interface standard

Theodore James O'Neill
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

O'Neill, Theodore James, "Creating an interactive web-based model course using the Common Gateway Interface standard" (1998). *Master's Theses*. 1659.
DOI: <https://doi.org/10.31979/etd.sh4c-qkng>
https://scholarworks.sjsu.edu/etd_theses/1659

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

CREATING AN INTERACTIVE WEB-BASED MODEL COURSE
USING THE COMMON GATEWAY INTERFACE STANDARD

A Project Report

Presented to

The Faculty of the School of Library and Information Science
San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Library and Information Science

by

Theodore James O'Neill

May 1998

UMI Number: 1389670

UMI Microform 1389670

Copyright 1998, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI

**300 North Zeeb Road
Ann Arbor, MI 48103**

© 1998

Theodore James O'Neill

ALL RIGHTS RESERVED

APPROVED FOR THE SCHOOL OF LIBRARY AND INFORMATION SCIENCE

Dr. Robert Wagers

Dr. Judy Tessier

Donald L. Tanner, Director of Digital Media Projects

APPROVED FOR THE UNIVERSITY

ABSTRACT

CREATING AN INTERACTIVE WEB-BASED MODEL COURSE USING THE COMMON GATEWAY INTERFACE STANDARD

by Theodore James O'Neill

This project report describes the design and implementation of an interactive, Web-based service called "NetLearn!" created using Practical Extraction and Report Language (PERL) and the World Wide Web (WWW) Common Gateway Interface (CGI) standard. It was developed to demonstrate the advanced capabilities of the Web as an instructional medium and to encourage its use for such applications. NetLearn! is a model mini-course in computer networking for librarians and information professionals consisting of four subject modules and two user modes. Student mode allows for the presentation of module subject matter, online testing with scoring and feedback, selective topic review, and storage of student information and exam responses. Administrator mode allows the instructor to review average and individual student performance and to manage student accounts. Secure access is provided by a password-based authentication system. The NetLearn! project indicates that WWW CGI is indeed of value in the development of instructional resources.

TABLE OF CONTENTS

Introduction.....	1
Literature Review: The WWW and CGI	6
References.....	14
Design and Implementation:	
Background	16
NetLearn! Walk-Through	18
Programming.....	29
Conclusion	35
Appendix A: C Form Parsing Source Code.....	38
Appendix B: 299login.cgi PERL Source Code.....	43
Appendix C: 299select.cgi PERL Source Code.....	53
Appendix D: 299modout.cgi PERL Source Code	58
Appendix E: 299admin.cgi PERL Source Code.....	66

LIST OF FIGURES

Figure 1. Log-In Page	22
Figure 2. Welcome Page	22
Figure 3. Status Summary Page	23
Figure 4. Module Content Page (Top)	23
Figure 5. Module Content Page (Bottom).....	24
Figure 6. Module Quiz Page (Top)	24
Figure 7. Module Quiz Page (Bottom)	25
Figure 8. Quiz Page Confirmation Dialog Box	25
Figure 9. Score/Topic Selection Page	26
Figure 10. Selected Topics Module Content Page.....	26
Figure 11. Administrative Summary Page.....	27
Figure 12. Administrative Delete Confirmation Dialog Box.....	27
Figure 13. NetLearn! Page Map.....	28
Figure 14. CGI Program and Page Map.....	28

INTRODUCTION

The popularity of the World Wide Web (WWW) has grown tremendously in recent years, and the sophistication and variety of its resources and services have likewise increased. One of the most significant changes is that the Web is no longer limited to a “one-size-fits-all” model in which each site offers the same static pool of information to every visitor. Information about visitors and their preferences can now be solicited, recorded, and used to customize the variety of options available and the actual content displayed, which enables Web sites to be more entertaining, useful, and interactive in the true sense of the word. For example, users can be greeted by name and presented with information directly relevant to them, such as news stories, local events, or even private information associated with personal accounts.

Web users can also now log in to many premium information services, such as DIALOG (<http://www.dialogweb.com>), where charges are accrued based on their online activities and the information they retrieve. And even though charging users for Web-based services may seem contrary to the free-for-all spirit of the Internet, fewer such services would be developed if it were not possible for companies to profit from them. Other examples of Web-based services that are designed for secure individual access include Wells Fargo Online Banking (<http://wellsfargo.com>) and Netcom Communications (<http://www.netcom.com>), an Internet service provider. These allow users to log in and access personal information, and, unlike DIALOG, these services are

free in that they are included as a standard component of each company's customer service benefits.

When it comes to developing and implementing such individualized Web-based services, several approaches using various technologies are currently available. One is to use Web browser scripting languages, such as Netscape's JavaScript, Microsoft's J-Script, or the built-in scripting elements of the latest HyperText Markup Language (HTML) standard (version 4.0). In addition to letting users interact with Web services in real-time through the use of dialog boxes, forms, and other interface elements, these scripting languages also allow the storage of user information for later reference. Such information can be used to greet returning visitors by name and even to remember where they left off. For example, the contents of a "virtual shopping cart" can be retained to permit future resumption of an interrupted shopping spree.

Such a scripting approach is said to be "client-side" because the script is downloaded along with the page and runs in the user's browser. Also, any information recorded for future use is stored on the user's local hard disk. The advantages of this approach are that the script runs quickly without the need for additional interaction with the server. In addition, the number of users that can be accommodated is unlimited (since the server is not burdened with the storage of user information). Disadvantages are that users can only take advantage of such features when using the particular computer on which their information is stored and that an administrative user cannot gain access to

other users' information. Also, the range of functions that can be performed by client-side scripting is somewhat limited when compared to other approaches.

A second approach to enhancing service interactivity is to use the Web's Common Gateway Interface, or CGI. This standard enables the Web server software to run other programs located on the server computer, passing information from the user's browser as input and returning program output back to the browser for display. These CGI server programs can perform functions beyond the scope of client-side browser scripting, such as accessing databases, sending e-mail, processing form responses, and outputting customized HTML documents created in real-time. They can also collect personal user information, which is processed and stored by the server rather than the user's local machine. In other words, CGI is a "server-side" approach. In addition to the functions mentioned above, this lets CGI programs perform advanced tasks, such as password authentication, allowing users to log in and access private or sensitive information.

Other advantages that stem from the server-side nature of CGI include enhanced information security—since servers tend to be more secure than desktop systems—and the centralized storage of data, which simplifies backups and allows users to interact with the service from any Web-capable computer. Moreover, it becomes possible for an administrative user to monitor visitors' individual and aggregate use of the service. This information could be used to evaluate user performance or the usefulness and accessibility of particular service features.

This project report describes the design and implementation of an interactive, Web-based service called “NetLearn!,” an online model mini-course in computer networking for librarians and information professionals. Briefly, NetLearn! works like this: On their first visit, students establish a personal account by choosing a username and password. Once registered, they can log in from any Web-capable computer. After entering some personal information (name and e-mail address) students can access course content modules built around various topic areas. Students work through the modules at their own pace, taking an online multiple-choice quiz at the end of each. This quiz is scored instantly and feedback is generated to help students focus on areas of difficulty. All progress is tracked and recorded, allowing students to continue on subsequent visits from where they left off. An administrator, such as the course instructor, can log in and view summary statistics on individual and average student performance. This information could then be used to tailor supplemental instruction, modify module content, or identify students who are having trouble with the course material. The administrator can also delete student obsolete user accounts.

NetLearn! is powered by a series of server programs written in Practical Extraction and Report Language (PERL) that are invoked through CGI. The technical objective in developing NetLearn! was to build a sophisticated model Web service using some of the technologies discussed above, especially server-side CGI. Some client-side techniques are used as well, including browser scripting using JavaScript. This dual client/server approach was chosen in order to balance the specific strengths and

weaknesses of each. For example, browser scripting is ideal for detecting invalid input and quickly bringing it to the user's attention. Although this is possible using CGI, it takes longer and requires additional network and server resources. On the other hand, CGI is capable of more advanced operations than browser scripting, as discussed earlier.

The motivation for creating an interactive Web-based course instead of a different type of service was based on the observation that Web sites developed for educational purposes rarely incorporate the more advanced capabilities of the medium. Such sites are usually limited to disseminating course information and rely on standard HTML documents, which are static by nature. Also, within the San Jose State University School of Library and Information Science, it seems likely that interactive Web-based course resources and instructional tools will find expanding application in the future as more students and faculty members become interested in exploiting the potential utility of the Internet. Many courses within the School already make use of standard Web pages, so interactive services and features represent the next logical step. It seemed timely to develop a prototype that would illustrate some of the possibilities.

LITERATURE REVIEW: CGI AND THE WWW

The World Wide Web (also known as “W3,” “WWW,” or “the Web”) is a vast collection of documents and data files stored on Internet computers located all over the world. The word “web” refers to the fact that many of these documents, or “pages,” are linked to each other in nonlinear fashion, allowing users to “browse” from one to another quickly and easily, often by selecting (“following”) a hyperlink based on its context in order to view related information. For example, technical terms on one page can be linked to definitions provided by some other document, such as a glossary. It is important to note that such links can logically connect documents even if they are located on geographically distant computers, as long as there is an available network pathway between the user’s computer and the two systems storing the linked documents [5, 18]. The standard computer language used for creating such hypertext documents is called HyperText Markup Language, or HTML. As of this writing, the current version of the HTML standard is 4.0 [13], as defined by the World Wide Web Consortium (W3C), the official body responsible for maintaining Web-related standards.

For the most part, Web documents are made publicly available and can be viewed using software called a Web browser on a PC (personal computer) or workstation, providing the system has access to the Internet. The browser, also known as the “client,” sends a request to the computer storing the desired information, known as the “server.” This request includes the address of the document or resource, called a Uniform Resource Locator, or URL. The server computer runs Web server software that “listens” for such

requests and responds by sending back the information or an error message if there is a problem. These client-server communications use a standard known as Hypertext Transfer Protocol, or HTTP, which is also maintained by the W3C. HTTP uses a system of headers that can specify additional information, such as the type of documents the browser can accept and the date the requested document was last modified [15]. The client-server nature of HTTP means that both computers share the total work involved in providing the user with the final product [2]. Namely, the server computer provides the desired information and the client computer formats it for display.

Originally, HTTP and the Web were developed as a means for research physicists to collaborate and share information by the European Laboratory for Particle Physics (CERN) in Geneva [5], but it was soon adopted by the Internet community at large and has since become incredibly popular, eclipsing most other Internet services with the possible exception of electronic mail (e-mail) [13, 18]. But as early as 1994, when the Web was beginning to rapidly gain in popularity over other then-state-of-the-art Internet services such as Gopher [5], it became apparent to Web authors and developers that something beyond HTML was needed if Web sites and pages were to truly live up to their billing as being “interactive” [3]. HTML is designed only to specify a document’s formatting, similar in function to other formatting languages like Adobe Systems’ PostScript [16]. This formatting may be considered dynamic in that the document’s layout will adjust when viewed under different circumstances. For example, changing the size of the browser window in which an HTML document is displayed might cause the

text near an image to “wrap” in order to accommodate the new window dimensions.

However, HTML has no capabilities for changing the content of a document. For instance, there is no HTML code that can display the current time. As often pointed out in the literature [1, 8, 10, 14, 15, 16], the content of an HTML document is fixed at the time it is created and remains static thereafter. Also, HTML cannot respond to user input [16], although it can collect information through the use of forms [19, 21].

These limitations led to the development of the Common Gateway Interface, or CGI, by the National Center for Supercomputing Applications at the University of Illinois as part of their Web server software [10]. CGI is a standard, or set of rules, that enables a Web server to communicate with other programs, often called CGI programs, located on the server machine [8]. Such programs are started by the Web server based on requests received from a user’s browser using HTTP. The Web server then provides the CGI program with any additional information received from the browser, such as HTTP headers or form input from the user. Typically, CGI programs process the information received and generate HTML or other output, which is directed back to the Web server. The Web server adds any HTTP headers and then sends the information back to the browser for display, as though it were any other document [10, 15]. The whole process can take anywhere from a few seconds to a couple minutes, depending mainly on the complexity of the CGI program.

The processing performed by CGI programs may include storing information in a database file, e-mailing it to someone for follow-up, performing numerical calculations,

generating graphics, or formulating and running a database query [1, 3, 14, 19]. Typical Web site applications for CGI programs include submitting order or feedback forms, signing guest books, submitting searches, verifying user information, validating credit-cards, counting page visits, displaying graphs or charts, and tracking link usage [1, 8, 11, 14]. Because CGI programs terminate after handling each request, subsequent requests typically include some sort of information to indicate that they are related to earlier CGI output [15]. In this way, a single CGI program might process two different HTML forms, the second of which it also generates based on the information obtained from the first.

CGI programs can be written in any programming language supported by Web server machine's operating system. Commonly, these include C, C++, Fortran, PERL, TCL, UNIX shell scripting languages such as SED and AWK, Visual Basic, and AppleScript. By far, the most popular of these is PERL (Practical Extraction and Report Language) [8]. One reason for this is that PERL is an interpreted programming language [17], also called a scripting language; for this reason PERL programs are sometimes called "scripts." In general, programs written in scripting languages are easier to debug, modify, and maintain than those written in compiled languages [10] because there are fewer steps involved. However, compiled language programs tend to be faster and may use less memory. C is a popular compiled language often used for CGI programming[1], but it is considerably more complex than PERL, requiring the programmer to carefully manage character strings and memory [9]. In PERL, these are managed automatically by

the interpreter [17, 20] which runs the script. Also, many pre-written PERL “library” functions are available to handle common CGI related tasks, such as outputting HTML forms [16].

Because CGI lets anyone with a Web browser run, or execute, programs on the server there is an inherent security risk [10, 16] associated with its use. To solve this problem, Web servers usually designate a special “cgi” or “cgi-bin” directory where all CGI programs must be kept. Only programs in this directory can be executed by the Web server, thereby limiting the user’s access to the server’s contents. Of course, even these designated CGI programs may contain security holes that can be exploited for malicious purposes [10], so it is important for programmers to code defensively.

There are two basic types of requests, called “methods,” that most browsers can send to a Web server. These are called “GET” and “POST.” Both methods pass HTTP headers and other information to the Web server, which in turn makes it available to the CGI program, but they use different mechanisms for doing so. The GET method, which is also the method used to send a request when a hyperlink is followed, is the simpler and more limited of the two. As part of the requested URL, a GET request can include a limited amount of arbitrary text. This text is “URL-encoded” (or “hexadecimal encoded”) [1, 15] and appended to the filename portion of the URL using a question mark. The encoding ensures that no illegal characters or characters with special meanings are transferred as part of the URL, which could confuse the Web server. Examples of such characters are the space (illegal) and slash (special meaning)

characters, encoded as “%20” and “%2F,” respectively. The POST method can handle a much larger amount of information than GET because that information is not incorporated into the URL—though it is URL-encoded. POST is the preferred method for handling form submissions, since they can be quite lengthy [1, 18].

The mechanism by which the Web server provides the information received to the CGI program involves the creation of special “environment variables,” which are units of information to which programs have access concerning the conditions under which they are running. The information in these variables can be read by the program but not modified. For example, the URL-encoded text gleaned from a GET request is stored in the environment variable “QUERY-STRING.” For POSTed information, the amount of information (number of characters) is stored as “CONTENT_LENGTH,” though the information itself is handled differently. Besides user input, environment variables also provide a wealth of information of which the user may be unaware. For example, the environment variable “HTTP_USER_AGENT” contains information about what kind of browser was used to send the request, its version, and the operating system under which it is running. Such browser information can be used by the CGI program to determine the special features the user can take advantage of and to vary its output accordingly. Other important environment variables can tell a program what types of documents a browser will accept (“HTTP_ACCEPT”) and the Internet (IP) address (“REMOTE_ADDR”) and name (“REMOTE_HOST”) of the user’s machine.

Even as Web technologies have continued to rapidly progress over the last four years, CGI has remained in widespread use. One reason for this is that CGI doesn't require any special compatibility on the part of the client [1]. All but the most antiquated browsers handle standard HTTP and HTML, and this is all CGI needs to be fully functional. As far as the browser is concerned, CGI is completely invisible, since all of the details are handled by the server. In fact, it is impossible to distinguish CGI output from that of a static HTML file, except of course, by the content, which is the reason for using CGI in the first place. However, if desired, CGI programs can take advantage of the latest browser technologies, such as browser scripting, which can directly control the behavior of the browser and react to user input in real-time. For example, a CGI program can output an HTML form that contains an embedded JavaScript that validates the form before submitting it back to the server [6]. Thus, CGI is compatible with new Web technologies that rely on basic HTTP.

Another reason for the ubiquity of CGI on the Web is very few limitations are imposed on the functionality of CGI programs. For example, a CGI program can act as a second gateway, which interacts with another program using a specialized API (application program interface). It has been pointed out that this method is inefficient and can be streamlined by including API functions for the desired target application program within the Web server itself [7]. But the design and implementation of a custom Web server is no small task. Though sometimes less efficient, the widespread compatibility of CGI with existing Web server software makes it an attractive approach. And, as interest

in interactive, customizable Web services continues to grow [12], there is little doubt that CGI will continue to play a major role. For example, in the growing Web-based distance education field, CGI applications have been used to create online tests that students can take using any browser [4]. This project will attempt to demonstrate how such applications of CGI can be developed.

REFERENCES

1. Boutell, Thomas. (1996). *CGI Programming in C & PERL*. New York: Addison-Wesley Developers Press.
2. Casad, Joe, & Newland, Dan. (1997). *MCSE Training Guide: Networking Essentials*. Indianapolis: New Riders Publishing.
3. The Common Gateway Interface for Enhancing Access to Database Servers via the WWW. (1995). *Online Libraries and Microcomputers*, v.13(10):1-3.
4. Dickinson, Ken. (1997). Distance Learning on the Internet: Testing Students Using Web Forms and the Computer [sic] Gateway Interface. *Tech Trends*, v.42(2):43-6.
5. Falk, Bennett. (1994). *The Internet Roadmap*. Alameda: Sybex.
6. Gesing, Ted, & Schneider, Jeremy. (1997). *JavaScript for the World Wide Web*. Berkeley: Peachpit Press.
7. Greenstein, D. (1996). API Trumps CGI. *Internet World*, v.7(5):32-3.
8. Gundavaram, Shishir. (1996). *CGI Programming on the World Wide Web*. Sebastopol: O'Reilly & Associates.
9. Horton, Ivor. (1995). *Instant C Programming*. Chicago: Wrox Press.
10. HTTPd Development Team. (1998). *The CGI Specification*. [Web page] National Center for Supercomputing Applications, University of Illinois: <http://hoohoo.ncsa.uiuc.edu/cgi/>.
11. Lee, Gary. (1996). CGI Programming Made Overly Simple. *Web Techniques*, v.1(1).
12. Machlis, Sharon. (1998). Pay-per-View for Biz News Taking Off. *ComputerWorld*, v.32(19):1.
13. Mack, E. Stephen, & Platt, Janet. (1997). *HTML 4.0, No Experience Required*. Alameda: Sybex.
14. Patchett, Craig, Wright, Matthew, & Holfelder, Peter. (1997). *The CGI/PERL Cookbook*. New York: John Wiley & Sons.

15. Spainhour, Stephen, & Quercia, Valerie. (1996). *Webmaster in a Nutshell*. Sebastopol: O'Reilly & Associates.
16. Stark, Thom. (1995). CGI: Gateway to the World Wide Web. *LAN Times*, v.12(25):125.
17. Till, David. (1996). *Teach Yourself Perl 5 in 21 Days, 2nd ed.* Indianapolis: Sams Publishing.
18. Tittle, Ed, & James, Steve. (1995). *HTML for Dummies*. Foster City: IDG Books Worldwide.
19. Tittel, Ed, Gaither, Mark, & Hassinger, Sebastian (Eds.). (1996). *CGI Bible*. Foster City: IDG Books Worldwide.
20. Vromans, Johan. (1996). *Perl 5 Desktop Reference*. Sebastopol: O'Reilly & Associates.
21. Wang, J., & Reid, E. (1996). Developing WWW Information Systems on the Internet. *Microcomputers for Information Management*, v.13(3-4):237-52.

DESIGN AND IMPLEMENTATION

Background

As a learning course in computer networking, NetLearn! consists of four content modules. Each covers a different networking subject area and is further divided into several sections, each discussing a distinct topic. For example, Module 2 is concerned with the World Wide Web and is divided into four sections, the third of which discusses Uniform Resource Locators. Each module also includes a short multiple-choice quiz covering all module sections. More modules could be easily added without requiring any programming since NetLearn! uses a system of data files to store information.

Functionally, NetLearn! consists of two user modes. The first is for student use and is designed to present course content, administer and score quizzes, give students feedback, and allow for selective review of module topics based on quiz scores or student preference. Student mode also maintains student information and a detailed record of all student quiz responses. This information is stored on the server (in a separate data file for each user), enabling students to log in from any Web-capable computer and continue from where they left off. The second user mode is for administrative users, such as the course instructor. It presents average and individual quiz scores for all students and each module. Administrator mode can also be used to delete obsolete user information.

The NetLearn! interface was designed to be simple to use while offering sophisticated features. All functions and information are accessed using seven main Web pages. These are produced by four interrelated PERL CGI programs—the NetLearn!

“engine.” Each page consists of content (system or course information), HTML formatting, and one or more HTML forms. These forms are used to communicate user responses and system information back to CGI for processing, often through the use of hidden fields. A few of these pages include some embedded JavaScript used for simple functions such as displaying dialog boxes. Also several auxiliary pages report errors and problems. The decision to use PERL instead of C or some other programming language was based on its ease of use and wide range of built-in text processing capabilities.

NetLearn! was designed and programmed in modular fashion to simplify system maintenance and maximize performance. Although it would be easy to write one long CGI program to handle all NetLearn! functions, this approach is less efficient than creating separate programs to handle related functions because it requires more time and server resources to run a larger program. Also, smaller programs are easier to modify and debug. Besides the CGI programs, NetLearn! uses a series of data files to store user information and course content. These are also modularized, with data for each content module and each student stored in a separate text file. Again, this is more efficient since the server deals with a smaller amount of information—only that which is relevant to the current task.

NetLearn! Walk-through

NetLearn! recognizes three types of users: new, returning, and administrative. New and returning users are assumed to be students. All three initiate a NetLearn! session from the Log-In page (Figure 1.). A new user fills out the New Users form, which requires choosing a username and password, both of which must be from five to fifteen alpha-numeric characters. The password must be entered twice in order to guard against typing errors. Problems that prevent registration and cause error messages to be displayed include invalid username or password, username already in use, and failed password confirmation (two different passwords entered). Returning and administrative users fill out the Returning Users form, supplying the previously registered information. Possible errors here include invalid username or password, username not registered (not entered correctly), and incorrect password. Administrative users are identified by the special reserved username "Administrator."

If appropriate information is submitted, one of three variations of the Welcome page is displayed, depending on the type of user (Figure 2.). Here, new users are prompted to enter a full name and e-mail address, while returning and administrative users are given information about when they last visited and prompted to confirm or correct their name and e-mail address currently on record. Administrative users are reminded that they are in administrative mode.

Upon clicking the “Continue” button, new and returning users (students) enter student mode and are presented with the personal Status Summary page showing information about each content module and indicating past quiz scores (Figure 3.).

After reviewing the information, students choose a content module for study by selecting its radio button and clicking “Start,” which causes the Module Content page for the selected module to appear (Figure 4.). This page displays all module topic sections. At the bottom of the Module Content page, students can choose to take a module quiz or return to the Status Summary page by clicking the appropriate button (Figure 5.).

If the student chooses “Test Your Knowledge,” the corresponding Module Quiz page appears (Figure 6.). If a returning student has previously answered any quiz questions, those choices will remain selected so that the student doesn’t have to start from scratch. Otherwise, all choices will be blank. After working on the quiz to whatever degree desired, the student has two choices: return to the Status Summary page or submit the quiz for scoring (Figure 7.). Attempting to return to the Status Summary page causes a confirmation dialog box to appear warning the student that no quiz progress information has yet been saved (Figure 8.). Selecting “OK” continues without saving whereas “Cancel” returns to the quiz.

Selecting “Check Answers” at the end of the quiz page will score the quiz and present the Score/Topic Selection page (Figure 9.). In addition to presenting detailed quiz results, this page also generates feedback by automatically suggesting topics for review. At this point, all progress has been recorded and the user can return to the Status

Summary page without receiving a warning. Or, the user can select desired module topics for further study—or de-select the topics automatically suggested—and return to a customized version of the content page, the Selected Topics Module Content page (Figure 10.), by clicking “Review Selected Topics.”

This completes the cycle. From here, the user can retake the quiz and continue selecting topics for review or return to the Status Summary page at any time, where the process can be started over with a different content module if desired.

In the case of an administrative user, clicking the “Continue” button on the administrative Welcome page displays the Administrative Summary page (Figure 11.). This page presents information about all currently registered students and their quiz scores for each module and total scores for all modules. It also displays information about maximum possible and average scores. Registered students who have not yet used the service are not included in the calculation of averages. This information on individual and aggregate quiz performance can be used by the instructor to identify particular subject areas in which students are having difficulty or to identify students who are struggling with the course material. The instructor might then choose to modify the module content or devote additional attention to that subject area or individual student in question. Finally, the instructor can permanently delete student accounts as desired, for instance at the end of the course or if a student drops the class. This action requires confirmation using a JavaScript dialog box (Figure 12.)

An overview map showing the structure of NetLearn! and the various pathways between the different pages is provided in Figure 13.

NetLearn! Login Netscape

<http://www.grid.com/cgi/239login.cgi>

NetLearn!

Returning Users

Username:

Password:

New Users

Username: Confirm pw:

Password:

All field entries must be 5 to 15 alphanumeric characters (no symbols or spaces).

Figure 1. Log-In Page

NetLearn! Login Netscape

<http://www.grid.com/cgi/239login.cgi>

Welcome back to NetLearn!

It has been less than a day since your last visit!
Please verify your personal information below. Thank you.

Personal Information

Name:

Email address:

Figure 2. Welcome Page

Figure 3. Status Summary Page

Figure 4. Module Content Page (Top)

Figure 5. Module Content Page (Bottom)

Figure 6. Module Quiz Page (Top)

Figure 7. Module Quiz Page (Bottom)

Figure 8. Quiz Page Confirmation Dialog Box

Figure 9. Score/Topic Selection Page

Figure 10. Selected Topics Module Content Page

NetLearn! Administrative Summary - Netscape

http://www.grid.com/cgi/293admin.cgi

NetLearn! Administrative Summary

Student Account	Module Number & Points Possible				Total Pts. 17	Delete Account
	1. 5 pts.	2. 5 pts.	3. 4 pts.	4. 3 pts.		
Ted O'Neill "toncill"	5	5	4	3	17	C
John Doe "jdoe456"	4	2	2	1	9	C
Don Tanner "dtanner"	3	0	0	0	3	C
No data for account "josephp"	NA					C
Averages	4.00	2.33	2.00	1.33	9.67	<input type="button" value="Delete"/>

Figure 11. Administrative Summary Page

NetLearn! Administrative Summary - Netscape

http://www.grid.com/cgi/293admin.cgi

NetLearn! Administrative Summary

Student Account	Module Number & Points Possible				Total Pts. 17	Delete Account
	1.	2.	3.	4.		
Ted O'Neill "toncill"						C
John Doe "jdoe456"						C
Don Tanner "dtanner"	3	0	0	0	3	C
No data for account "josephp"	NA					C
Averages	4.00	2.33	2.00	1.33	9.67	<input type="button" value="Delete"/>

www.grid.com [JavaScript Application]

?

Are you sure you want to delete this account?

Figure 12. Administrative Delete Confirmation Dialog Box

Figure 13. NetLearn! Page Map

Figure 14. CGI Program and Page Map

Programming

The four PERL CGI programs that comprise NetLearn! are designed to handle specific groups of related tasks: 1. log-in tasks (creation of new accounts, welcome pages); 2. summary tasks (module information and selection); 3. module display tasks (topics and quizzes, scoring); and 4. administrative tasks (performance summaries, account management). These four programs have several features in common. For example, each processes input received from HTML forms and must therefore be able to decode URL-encoded data and parse it into appropriate field variables. This is easily accomplished by the following PERL program lines:

```
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'}); # read URL-encoded data
@pairs = split(/&/, $buffer);                  # store name/value pairs
foreach $pair (@pairs) {                       # process each pair
    $pair =~ tr/+//;                            # replace encoded spaces
    ($name, $value) = split(/=/, $pair);        # split name from value
    $value =~ s/%([a-f0-9][a-f0-9])/pack("C", hex($1))/ieg; # decode hex
    $data{$name} = $value; # store name/value pairs in associative array
}
```

This code produces an associative array containing all submitted form information.

Associative arrays are a convenient way to store labeled pieces of information in PERL.

This code appears in all four NetLearn! programs as the subroutine “parseForm.” It is interesting to note that it requires many more lines of code to accomplish the same thing using C (see Appendix A. for the C source code).

The first program is called 299login.cgi. The full PERL source code is listed in Appendix B. Briefly, this program produces and processes the Log-In page (Figure 1.).

It also produces the three variations of the Welcome page (Figure 2.), depending on the type of user logging in. Finally, it creates new user accounts when required.

The first thing 299login.cgi does is check for the presence of POSTed information from an HTML form. If none is present, it outputs the Log-In page and ends. This can be thought of as the default behavior and the entry point to the NetLean! service. If POSTed information is present, it is parsed as described above. Because the program stores various user information in a file called users.dat, it next checks to make sure this file exists and creates it if it does not. This would only occur when the very first student or administrator logged in for the first time. The file is backed up (copied) before proceeding in case it gets corrupted by an error. Next, the program verifies that all form fields contain valid (though not necessarily correct) information, such as the proper number and type of characters. If all is in order, 299login.cgi begins the actual task of processing the log-in information.

If the New Users form was submitted (as evidenced by the presence of the second password field), the program attempts to create a new user account. This involves the following steps: 1. making sure the submitted username is not already in use and that the two passwords are consistent; 2. adding the new user information to the users.dat file, as well as the current time and date; and 3. creating a personal data file for the user. If the Returning Users form was submitted, the program verifies that the username exists in the users.dat file and that the password is correct. It then calculates the time elapsed since the user's last visit and updates the user's information with the current time and date for

future reference. After producing one of three variations of the Welcome page as previously described, the program is done and execution terminates.

The second program is called 299select.cgi. Its source code is listed in Appendix C. This program processes the Welcome form and outputs the Status Summary page used to select content modules (Figure 3.). It begins by checking that the expected pieces of information have indeed been POSTed and comparing the submitted name and e-mail address to those currently stored in the user's personal data file. If no information was POSTed, the program returns an error message; there is no useful default output as in the case of the log-in program. If the name and e-mail address are different or none is currently recorded, the user's file is updated. The program then outputs the Status Summary page, which involves reading the user's personal data file and parsing it into the values to be displayed in the Your Score column. A module summary data file (modules.dat) is also consulted to similarly obtain the Max Score values for each module.

The third program is called 299modout.cgi. Its source code is listed in Appendix D. This program is responsible for all module content output, including quizzes and selected topics. It also scores module quizzes. Unlike the first two programs, 299modout.cgi both processes and outputs multiple forms and pages as follows: 1. reads the module selection form from the Status Summary page and outputs the appropriate Module Content page (Figure 4.); 2. receives the quiz request form (a form that contains only hidden fields), resulting in the display of the corresponding Module Quiz page (Figure 6.); 3. scores the multiple-choice quiz form, displaying feedback and results as

the Score/Topic Selection page (Figure 9.); and 4. processes the user's topic selections and displays the selected topics as the Selected Module Content Page (Figure 10.), which can be used to start the cycle all over again. The user can choose to return to the Status Summary page at several points along the way.

The program begins by checking for the presence of POSTed form information and parsing it or returning an error message if none is detected. Each of the four forms processed by 299modout.cgi includes a hidden field identifying itself. Although the forms could have been identified exclusively by their content, using a form identifier instead makes the program code less confusing and easier to follow. For producing both variations of the Module Content page (all topics and selected topics) in processes 1. and 4., the identifier is the same because similar processing is involved. Specifically, the program reads the data file for the appropriate module and outputs the topics one at a time. If information about topic selection is absent, all topics are displayed by default.

Producing the quiz (process 2.) requires 299modout.cgi to read the questions and answer choices from the appropriate module data file and assemble the information into an HTML form employing groups of radio buttons. This is fairly straightforward, but there is one complication: If the user has previously taken the quiz, the recorded answers must be read from the user's personal data file and indicated on the quiz form. This prevents users from having to repeatedly answer the same questions. Instead, they can simply edit their previous responses.

Scoring the quiz (process 3.) requires that the topic names and quiz answers be read from the module data file. The correct answers are then compared to the user's submitted choices. In addition to computing a score, the program saves the submitted data to the user's personal data file for future reference. It also compares the current score to previously recorded scores and provides feedback concerning any decline or improvement in performance. Finally, the program automatically selects topics for review if the corresponding question was answered incorrectly. This is accomplished using the CHECKED attribute of the HTML form checkbox.

The fourth and last program in the NetLearn! engine is called 299admin.cgi. Its source code is listed in Appendix E. This program is responsible for processing the administrative version of the Welcome form, as well as producing and processing the Administrative Summary page (Figure 11.) Form identifiers are again used to simplify the code. After checking for the presence of POSTed information, parsing it, and making sure that all required information has been provided, the program checks the submitted form identifier and proceeds as indicated.

If the Welcome form identifier is present, the program compares the submitted name and e-mail address to those currently on record, updating the administrator data file if necessary. Next, the program gathers and formats module and user information from the users log-in data file (containing information on all users), the module data files, and the individual student data files. Maximum possible and user scores for each module are then presented as table data, as are total maximum and user scores. Average scores for

each module are calculated by adding all user scores and dividing by the number of users with recorded data.

If the administrator selects an account and then clicks the “Delete” button, a warning dialog box will be displayed (Figure 12.) using JavaScript. If the action is confirmed, the delete form identifier will be submitted along with information about the user account to be removed. An updated version of the Administrative Summary page is displayed and the program ends.

Figure 14. indicates graphically which pages are processed and produced by the various NetLearn! programs.

CONCLUSION

The NetLearn! project demonstrates that the advanced capabilities of the World Wide Web can be applied successfully to the design of instructional tools for use in traditional classroom-based and distance education. However, several improvements to the NetLearn! service could be made to increase its usefulness, especially with regard to administrative capabilities.

For example, the current method for adding, modifying, or deleting course content (both topic material and quiz questions) requires the careful editing of precisely formatted text files. Because the inclusion of even an extra tab character can render the data file unusable, testing of the modified file is also necessary. While this system is better than one requiring modification of the program files, it is still rather complicated and therefore inconvenient. Many instructors wouldn't have the time or patience for such maintenance.

A better solution, and one which could be easily implemented, is to include Web-based content editing functions within the administrative user mode. Using HTML forms similar to those used throughout NetLearn!, instructors would have the option to modify or create new content modules. Data file formatting and other details would be handled by the system.

Another limitation of the current NetLearn! system is that only one course can be accommodated at a time. It would be more useful if several instructors could maintain different courses simultaneously. This could be accomplished by implementing a directory system on the server that would contain the various module and user data files

for each course. Students would then choose the desired course when registering. A system of add codes could even be set up to regulate registration privileges.

As far as student mode capabilities, NetLearn! is limited in that a strict correspondence must be maintained between module topics and quiz questions: each topic is associated with exactly one question. It would be useful to have the option of associating multiple questions with a topic, or none at all. Also, no hypertext capabilities are supported by the content display pages. These limitations are more difficult to overcome because doing so requires an extensive redesign of fundamental NetLearn! algorithms. However, such changes may well be justified because the current system doesn't make full use of traditional Web capabilities familiar to students and instructors, although it demonstrates new ones.

Finally, once the desired capabilities and design have been developed and finalized, it would be advantageous to port (convert) the entire NetLearn! system over to a compiled programming language such as C. The complexities involved in doing so are considerable, but the increase in performance would be significant, reducing user waiting time during processing and decreasing the burden on server processing resources.

In conclusion, although NetLearn! could be improved in many ways, the project still shows very clearly that the World Wide Web (WWW) Common Gateway Interface (CGI) is a useful and valuable means for producing advanced, interactive Web-based services and resources. Furthermore, this approach can be easily and successfully applied to educational applications, such as the design and implementation of Web-based

instructional tools for students and instructors. And because CGI is built in to almost all Web server software currently in use and compatible with almost all Web browsers, it remains an excellent and viable choice, even when compared to newer technologies. In fact, CGI can easily be integrated with newer technologies, such as browser scripting, in complementary fashion. For these reasons, it is anticipated that CGI will remain an important and influential factor in the continuing evolution of the World Wide Web.

APPENDIX A

```
/*
*****

CGI Form Parsing POST Version in C, by Ted O'Neill, 1998

*****
*/

#ifndef _webcgi_h_
#define _webcgi_h_ 1

#include <stdio.h> // include standard input/output functions
#include <stdlib.h> // include standard library functions
#include <string.h> // include standart string functions

#define debug 1 // debug switch; 1 is 'on'

/*
*****
Function to be used by main():
*/
char * valueof(char * field_name); // return field contents
/*
*****
*/

/****** global variables (in this file) *****/
/* char array that will hold all decoded form data as substrings */
char * data = NULL; // memory will be allocated later, just before use
/* index array of pointers to field names in data[] */
char ** names = NULL;
/* index array of pointers to field values in data[] */
char ** values = NULL;
/* number of fields in form, as counted in decodeData() */
int numFields = 0;

/* helper functions for valueof() */
int parseForm(); // process form, ret num fields found
int checkForm(); // check that form was POSTed correctly, ret 0 or 1
int decodeData(int content_length); // decode data, ret num fields total
int tokData(char * str, int numFields); // tokenize data, ret num field

/****** form functions *****/

```



```

char * valueof(char * name){
    int i; //counter
    if(!(numFields)){
        if(!(parseForm())){ // error parsing form
            numFields = -1; // to prevent additional calls to parseForm()
            return NULL;
        }
    }
    for(i=0;i<numFields;i++){ // loop through all fields if necessary
        if( strcmp(names[i], name) == 0 ) return values[i]; //ret corresponding value
    }
    if(numFields > 0){
        printf("<P>Error: Field \"%s\" not found\n", name); // no field match
    }
    return NULL;
}

int parseForm(){ // no arguments, returns number of fields in form if successful
    if(debug)printf("<P>Calling checkForm()\n");
    if(!(checkForm())){
        if(debug){
            printf("<P>Error: checkForm() returned 0\n");
            printf("<P>Error: parseForm() returning 0 - FAILED\n");
        }
        return 0; //not a properly posted form
    }
    if(debug)printf("<P>Calling decodeData()\n");
    /* decode urlencoded data and count number of fields in form */
    if(!(decodeData(atoi(getenv("CONTENT_LENGTH"))))){
        if(debug)printf("<P>Error: decodeData() returned 0\n");
        printf("<P>Error: parseForm() returning 0 - FAILED\n");
        return 0; //error decoding
    }
    if(debug)printf("<P>Calling tokData(): ");
    /* tokenize data and compare number of fields processed to number present */
    if(tokData(data, numFields) == numFields) return 1; // all processed!
    else return 0; // error
}

/*****/

int checkForm(){

```

```

/* check for non-zero content length */
if(!(getenv("CONTENT_LENGTH"))){
    if(debug)printf("<P>Error: CONTENT_LENGTH is
%s\n",getenv("CONTENT_LENGTH"));
    return 0; //empty submission!
}

/* check for correct content type */
if( strcmp(getenv("CONTENT_TYPE"),"application/x-www-form-urlencoded") != 0 ){
    if(debug)printf("<P>Error: CONTENT_TYPE is %s\n",getenv("CONTENT_TYPE"));
    return 0; // wrong or null content
}

/* check for correct submission method */
else if( strcmp(getenv("REQUEST_METHOD"),"POST") != 0 ){
    if(debug)printf("<P>Error: REQUEST_METHOD is
%s\n",getenv("REQUEST_METHOD"));
    return 0; // wrong form method
}
else {
    if(debug)printf("<P>checkForm() successful, returning 1\n");
    return 1; // looks good!
}
}

int decodeData(int length){
    int i = 0; // counter
    char * pdata = NULL; // pointer to current data character
    char chars[3] = {'\0'}; // to hold hex string or regular char
    if(debug)printf("with length %d\n", length);
    if((pdata=data=(char *)calloc((length+1), sizeof(char)))==NULL ){
        printf("Error: memory allocation for buffer failed\n");
        return 0; // memory allocatio failed
    }
    else if(debug){
        printf("<P>Buffer memory allocated: %d bytes<P>\n", length+1);
        printf("<TABLE
BORDER=1\n"><TR><TD>Position:<TD>Char:<TD>Action:\n");
    }
    for(i=0; i<length; i++){
        chars[0] = getchar();
        if(debug)printf("<TR><TD>%d<TD>%c", i, chars[0]);
        if(chars[0] == '=') numFields++; // increment field count
        if(chars[0] == '+'){

```

```

        if(debug)printf("&lt;TD>replaced with space\n");
        *pdata = ' ';          // replace plus sign with space
    }
    else if(chars[0] != '%'){
        if(debug)printf("&lt;TD>copied as is\n");
        *pdata = chars[0]; // copy regular characters directly
    }
    else { // it is an escaped character. catch it!
        if(debug)printf("&lt;TD>decode next two chars\n");
        if((i+2)>=length){
            if(debug)printf("&lt;/TABLE>&lt;LI>Error: missing escape values\n");
            return 0; // error: 2 chars needed after %
        }
        chars[0] = getchar(); // read 2 digit hex code
        i++; // keep track of number of chars read
        chars[1] = getchar();
        i++;
        if(debug)printf("&lt;TR>&lt;TD>...");
        if(debug)printf("&lt;TD COLSPAN=\"2\">decoding hex string \"%s\"\n", chars);
        if(!sscanf(chars, "%x", pdata)) == EOF) return 0; // read hex string and store
        if(debug)printf("&lt;TR>&lt;TD>...&lt;TD COLSPAN=\"2\">char \"%c\"
copied\n", *pdata);
    } // end of else block
    pdata++; // advance data array pointer
} // end of for loop
*pdata = '\0'; // add terminating null
if(debug)printf("&lt;/TABLE>&lt;P>Decoded string is:\n&lt;P>%s\n", data);
return 1; // success!
}

int tokData(char * str, int numFields){
    int i = 0; //counter
    if((names = (char **)calloc(numFields, sizeof(char *))) == NULL ){
        printf("Error: memory allocation for names index failed\n");
        return 0; // memory allocation failed
    }
    if((values = (char **)calloc(numFields, sizeof(char *))) == NULL ){
        printf("Error: memory allocation for values index failed\n");
        return 0; // memory allocation failed
    }
    if(debug)printf("&lt;P>tokData() called\n");
    for(i=0; i<numFields; i++){

```

```

    names[i] = str; // store start address of name
    while (*str != '=') str++; // skip to equal sign at end of name
    *str = '\0'; // convert equal sign to terminating null
    if(debug)printf("<P>Name token %d found: %s\n", i, names[i]);
    str++; // advance to value start (or term. null of data string)
    values[i] = str; // store start/term. null address
    while( (*str != '&') && (*str != '\0') ) str++; // skip to end of value
    if(*str == '\0'){
        if(debug)printf("<P>Value token %d found: %s<P>END DATA\n", i,
values[i]);
        i++; // since loop won't get to final increment
        break; // if end of data, stop looping
    }
    *str = '\0'; // if &, there is more data, so terminate value string
    if(debug)printf("<P>Value token %d found: %s\n", i, values[i]);
    str++; // advance to start of next name
}
if(debug)printf("<P>tokData() finished: %d fields found\n", i);
return i; // actual number of fields processed
}

#endif // _webcgi_h_

int main(){
    int i; //counter
    printf("Content-type: text/html\n\n");
    printf("<HTML><HEAD>\n<TITLE>C Post Test Results</TITLE>\n");
    printf("</HEAD><BODY BGCOLOR=\"#EEEEEE\">\n");
    printf("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000ff\">C Post Test
Results</FONT></H2>\n");
    if(!parseForm())printf("<P>Error parsing form!\n"); // parse the form
    printf("<H2>Final Output:</H2>\n");
    for(i=0;i<numFields;i++){
        printf("<LI>Value of field \"%s\" is \"%s\"\n",names[i],values[i]);
    }
    printf("</BODY></HTML>\n");
    return 0;
}

```

APPENDIX B

```
#!/usr/local/bin/perl5
#####
#
#           NetLearn! Log-in Page, by Ted O'Neill, 1998
#
#   This program creates new user accounts and information files
#   and authenticates existing users, presenting a status page to both.
#
#####

# set to 1 for verbose debugging output
$debug = 0;

# check for presence of POSTed form info
if($ENV{'CONTENT_LENGTH'}){
    &parseForm;    # parse form information into associative array
}else{ # nothing posted, so display default page
    &loginPage;    # display login form
    exit;         # end program
}

&startHTML; # start HTML output page

# check debugging option
if($data{'debug'}){$debug = 1;} # turn on debugging if requested
print("<P>Debugging info will be displayed...\n")if($debug);

# create users data file if not already present
if(!( -e "../tone/net/users.dat" )){&createFile;}

# backup the users data file in case it gets corrupted
print("<P>Calling sub backupFile...\n")if($debug);
&backupFile;

# check fields for valid info
print("<P>Calling sub checkInfo...\n")if($debug);
&checkInfo;

# handle user depending on new/existing status
if($data{'password2'}){ # this must be a new user!
```

```

print("<P>Calling sub createUser for new user...\n")if($debug);
&createUser; # add new user info to users data file and create user file
}else{ # this is an existing user
print("<P>Calling authenticate for existing user...\n")if($debug);
&authenticate; # check password
print("<P>Calling updateTime for existing user...\n")if($debug);
&updateTime; # record current time for future reference
}

# confirm login success and welcome new/existing user
print("<P>Calling sub welcome...\n")if($debug);
&welcome;

print("<P>All done. Exiting...\n")if($debug);
&done; # end program

##### subroutines #####

# start HTML output page #####
sub startHTML(){
print("Content-type: text/html\n\n");
print("<HTML>\n<HEAD>\n<TITLE>NetLearn! Login</TITLE>\n</HEAD>\n");
print("<BODY BGCOLOR=\"#EEEEEE\">\n<BLOCKQUOTE>\n<CENTER>\n");
}

# parse posted form data from any form #####
sub parseForm(){
read(STDIN, $buffer, $ENV['CONTENT_LENGTH']); # read urlencoded data
@pairs = split(/&/, $buffer); # store name/value pairs
foreach $pair (@pairs) { # process each pair
($name, $value) = split(/=/, $pair); # split at equals sign
$name =~ tr/+//; # replace encoded spaces
$value =~ tr/+//;
# the next line replaces hex encoded escape sequences with ascii characters
$value =~ s/%([a-f0-9][a-f0-9])/pack("C", hex($1))/ieg;
$data{$name} = $value; # store name/value pairs in associative array %data
} # done processing pairs
} # end

# create/reset users data file #####
sub createFile(){
open(FILE, ">../tone/net/users.dat") # create and close file

```

```

    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> # or print error msg
        Error: could not create user info file</H2></FONT>\n") && &done;# and quit
$time = time(); # get current time stamp
print FILE ("Administrator\tBenjamin\t$time\n"); # write admin acct info
close(FILE); # close file
`chmod 000 ../tone/net/users.dat`; # restrict file access for security
if($debug){`chmod 666 ../tone/net/users.dat`;} # allow debugging access
# create administrator file
open(FILE, ">../tone/net/Administrator.dat") && close(FILE)
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
        Error: could not creat Admin's info file</H2></FONT>\n") && &done;
`chmod 000 ../tone/net/Administrator.dat`; # restrict file access
if($debug){`chmod 666 ../tone/net/Administrator.dat`;} # debugging access
} # end

# backup user data file before any other read/write activities #####
sub backupFile(){
    `chmod 400 ../tone/net/users.dat`; # allow file read access
    open(FILE, "../tone/net/users.dat") # open file for reading
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> # or print error msg
            Error: could not open user info file</H2></FONT>\n") && &done; # and quit
        @users = <FILE>; # copy file contents
    close(FILE) # close file
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> # or print error msg
            Error: could not close user info file</H2></FONT>\n") && &done; # and quit
    `chmod 000 ../tone/net/users.dat`; # restrict file access for security
    if($debug){`chmod 666 ../tone/net/users.dat`;} # allow debugging access
    if(-e "../tone/net/users.bak"){`chmod 600 ../tone/net/users.bak`;} # access
    open(FILE, ">../tone/net/users.bak") # open file for writing
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> # or print error msg
            Error: could not open user backup file</H2></FONT>\n") && &done;# and quit
    print FILE (@users); # write data to backup file
    close(FILE) # close file
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> # or print error msg
            Error: could not open user backup file</H2></FONT>\n") && &done;# and quit
    `chmod 000 ../tone/net/users.bak`; # restrict file access for security
    if($debug){`chmod 666 ../tone/net/users.dat`;} # allow debugging access
}

# check for valid form information #####
sub checkInfo(){
    if( !($data{'username'} =~ /^[a-zA-Z0-9]{5,15}$/) ){

```

```

        print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
        print("Missing or invalid username!</H2></FONT><P>Please try again.\n");
        &done; # username too short or long, or contains illegal characters
    }
    if( $data{'password2'} && {-e "../tone/net/$data{'username'}.dat" } ){
        print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
        print("Username already taken!</H2></FONT><P>Please try again.\n");
        &done; # filename already in use
    }
    if ( !($data{'password'} =~ /^[a-zA-Z0-9]{5,15}$/) ){
        print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
        print("Missing or invalid password!</H2></FONT><P>Please try again.\n");
        &done; # missing information
    }
    if ( $data{'password2'} && ($data{'password'} ne $data{'password2'}) ){
        print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">Passwords
inconsistent!");
        print("</H2></FONT><P>Please try again.\n");
        &done; # password confirmation failed
    }
}

# for new users, create entry in users file and individual data file #####
sub createUser(){
    `chmod 400 ../tone/net/users.dat`; # allow file read access
    open(FILE, "../tone/net/users.dat") # open file
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> # or print error msg
        Error: could not read user info file</H2></FONT>\n") && &done; # and quit
    @info = <FILE>; # read user data into array @info
    close(FILE) # and close the file
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> # or print error msg
        Error: could not close user info file</H2></FONT>\n") && &done; # and quit
    chop(@info); # remove newline character from each line
    print("<P>Processing individual lines of info file...\n") if($debug);
    foreach $line (@info){ # examine info for each existing user
        ($username, $password, $lasttime) = split(/\t/, $line);
        if ($data{'username'} eq $username){
            print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
            print("Username already taken!</H2></FONT><P>Please try again.\n");
            &done; # username already taken, so quitting
        }
    }
} # done examining all users, username not already in use so continue

```



```

print("<P>Username not in use. OK to create. Getting time...\n")if($debug);
$time = time(); # get current time in seconds
print("<P>Current time in seconds: $time\n")if($debug);
`chmod 600 ../tone/net/users.dat`;
open(FILE, ">>../tone/net/users.dat") # open info file in append mode
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
        Error: could not open file</H2></FONT>\n") && &done;
print("<P>User info file opened for appending...\n")if($debug);
print FILE ("<$data{'username'}>\t<$data{'password'}>\t<$time>\n");
print("<P>New user info appended to file successfully!\n")if($debug);
close(FILE)
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
        Error: could not close file</H2></FONT>\n") && &done;
`chmod 000 ../tone/net/users.dat`; # restrict file access
if($debug){`chmod 666 ../tone/net/users.dat`;} # allow debugging access
# create user's individual data file
print("<P>Creating user's individual data file...\n")if($debug);
open(FILE, ">../tone/net/$data{'username'}.dat") && close(FILE)
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
        Error: could not creat user's info file</H2></FONT>\n") && &done;
`chmod 000 ../tone/net/$data{'username'}.dat`; # restrict file access
if($debug){`chmod 666 ../tone/net/$data{'username'}.dat`;} # debugging access
} # end

# check user name and password for purported existing user #####
sub authenticate(){
    `chmod 400 ../tone/net/users.dat`; # allow file access
    open(FILE, "../tone/net/users.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not parse user info file</H2></FONT>\n") && &done;
    @info = <FILE>;
    close(FILE)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not close user info file</H2></FONT>\n") && &done;
    `chmod 000 ../tone/net/users.dat`; # restrict file access
    if($debug){`chmod 666 ../tone/net/users.dat`;} # allow debugging access
    chop(@info);
    foreach $line (@info){ # process each user's info
        ($username, $password, $lasttime) = split(/\t/, $line);
        if ($data{'username'} eq $username){ # check username
            if ($data{'password'} eq $password){ # check password
                $time = time(); # get current time

```

```

        $elapsed = int( ($time - $lasttime) / 86400 ); # days since last login
        return(1); # user authenticated successfully, exit subroutine
    }else{
        print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">Password
incorrect!</H2>");
        print("</FONT><P>Please try again.\n");
        &done; # found user, but pw doesn't match
    }
}

print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">Username not found!</H2>");
print("</FONT><P>Please try again.\n");
&done; # username not found
}

# update last login time for existing users #####
sub updateTime(){
    `chmod 600 ../tone/net/users.dat`; # allow file access
    open(FILE, "../tone/net/users.dat") # open for reading
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
        Error: could not read user info file</H2></FONT>\n") && &done;
    @timeInfo = <FILE>; # read all user info
    close(FILE);
    open(FILE, ">../tone/net/users.dat") # open for writing
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
        Error: could not update user info file</H2></FONT>\n") && &done;
    chop(@timeInfo); # remove newline characters
    foreach $line (@timeInfo){ # find user's info
        ($username, $password, $lasttime) = split(/\t/, $line);
        if ($data{'username'} eq $username){ # update this line
            print FILE ("{$username}\t$password\t$time\n");
        }else{
            print FILE ("{$line}\n"); # write unchanged line for non-matches
        }
    }
    close(FILE)
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
        Error: could not close updated user info file</H2></FONT>\n") && &done;
    `chmod 000 ../tone/net/users.dat`; # restrict file access
    if($debug){`chmod 666 ../tone/net/users.dat`;} # allow debugging access
}

```

```

# display welcome page for new and existing users #####
sub welcome(){
  print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">Welcome ");
  print("back ")if(!$data('password2'));
  print("to NetLearn!</H2></FONT></FONT>\n");
  if(!$data('password2')){ ##### processing for existing users
    print("<P>It has been less than a day since your last
visit!\n")if(!$elapsed);
    print("<P>Your last visit was about a day ago!\n")if($elapsed == 1);
    print("<P>Your last visit was about $elapsed days ago!\n")if($elapsed > 1);
    # get user's personal info, if present
    if(-s "../tone/net/$data('username').dat"){ # get user info if file not empty
      &getUserInfo;
      chop($userInfo[0]);
      ($name, $email) = split(/\t/, $userInfo[0]); # store name and email
    }else{ # file is empty, so set to nothing to keep random values out
      $name="";
      $email="";
    }
  }else{ ##### processing for new users (password2 present)
    print("<P>Thank you for registering!\n");
    $name="";
    $email="";
  }
  if($name && $email){
    print("<BR>Please verify your personal information below. Thank you.\n");
  }else{
    print("<BR>Please enter your personal information below. Thank you.\n");
  }
  if($data('username') eq "Administrator"){ # administrator login
    print("<BR><B>You are logged in as Administrator!</B>\n");
    print("<FORM METHOD=\"POST\" ACTION=\"../../cgi/299admin.cgi\">\n");
  }else{ # everyone else
    print("<FORM METHOD=\"POST\" ACTION=\"../../cgi/299select.cgi\">\n");
  }
  print("<TABLE BORDER=\"1\" CELLPADDING=\"15\">\n");
  print("<TR><TD><CENTER><B>Personal Information</B></CENTER><BR>\n");
  print("<FONT SIZE=\"-1\">\n");
  print("Name:<BR><INPUT TYPE=\"text\" NAME=\"name\" VALUE=\"$name\" ");
  print("SIZE=\"25\"><BR>\n");
  print("Email address:<BR><INPUT TYPE=\"text\" NAME=\"email\" ");
  print("VALUE=\"$email\" SIZE=\"25\"><BR><BR>\n");
}

```

```

print("<CENTER><INPUT TYPE=\"submit\" VALUE=\"Continue\"></CENTER>");
print("</TD></TR></TABLE>\n");
print("<INPUT TYPE=\"hidden\" NAME=\"username\"
VALUE=\"\$data('username')\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"form\" VALUE=\"welcome\"></FORM>\n");
}

# get single user's personal info and store in array @userInfo
sub getUserInfo(){
`chmod 400 ../tone/net/$data('username').dat`; # allow file access
open(FILE, "../tone/net/$data('username').dat")
|| print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
Error: could not open user's file</H2></FONT>\n") && &done;
@userInfo = <FILE>; # read file
close(FILE)
|| print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
Error: could not close user's file</H2></FONT>\n") && &done;
`chmod 000 ../tone/net/$data('username').dat`; # restrict file access
`chmod 666 ../tone/net/$data('username').dat` if($debug); # for manual access
}

# close HTML document and exit program #####
sub done(){
print("</CENTER>\n</BLOCKQUOTE>\n</BODY>\n</HTML>\n");
exit;
}

# display default login page when nothing is posted ##### (last
subroutine)
sub loginPage(){
print <<LOGINPAGE;
Content-type: text/html

<HTML>
<HEAD>
<TITLE>NetLearn! Login</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--HIDE
function validate(form){
for(var i = 0; i < form.elements.length; i++){
if (form.elements[i].value == ""){

```

```

        alert("\\nPlease be sure to fill out the \"" + form.elements[i].name +
        "\\\" field.")
        form.elements[i].focus()
        return(false)
    }
}
return(true)
}
//END-->
</SCRIPT>
</HEAD>
<BODY BGCOLOR="#EEEEEE">
<CENTER>
<TABLE WIDTH="100% HEIGHT="100%" BORDER="0" CELLPADDING="0" CELLSPACING="0">
<TR>
<TD COLSPAN="2" ALIGN="center">
<IMG SRC="../../../tone/net/netlearn.gif" ALT="NetLearn!" BORDER="0" VSPACE="0">
</TD>
</TR>
<TR>
<FORM METHOD="POST" ACTION="../../../cgi/299login.cgi" onSubmit="return
validate(this)">
<TD ALIGN="center" WIDTH="50%">
<TABLE BORDER="1" CELLPADDING="6" WIDTH="75%"><TR><TD ALIGN="center"
VALIGN="middle">
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0">
<TR>
<TD COLSPAN="2" ALIGN="center"><B>Returning Users</B></TD>
</TR>
<TR>
<TD VALIGN="bottom"><FONT SIZE="-1">
Username:<BR><INPUT TYPE="text" NAME="username" SIZE="11"><BR>
Password:<BR><INPUT TYPE="password" NAME="password" SIZE="11">
</FONT></TD>
<TD VALIGN="bottom"><FONT SIZE="-1">
<INPUT TYPE="submit" VALUE="Log-In">
</FONT></TD>
</TR>
</TABLE>
</TD></TR></TABLE>
</TD>
</FORM>

```

```

<FORM METHOD="POST" ACTION="../../cgi/299login.cgi" onSubmit="return
validate(this)">
<TD ALIGN="center" WIDTH="50%">
<TABLE BORDER="1" CELLPADDING="6" WIDTH="75%"><TR><TD ALIGN="center"
VALIGN="middle">
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0">
<TR>
<TD COLSPAN="2" ALIGN="center"><B>New Users</B></TD>
</TR>
<TR>
<TD VALIGN="bottom"><FONT SIZE="-1">
Username:<BR><INPUT TYPE="text" NAME="username" SIZE="11"><BR>
Password:<BR><INPUT TYPE="password" NAME="password" SIZE="11">
</FONT></TD>
<TD VALIGN="bottom"><FONT SIZE="-1">
Confirm pw:<BR><INPUT TYPE="password" NAME="password2" SIZE="11"><BR>
<BR><INPUT TYPE="submit" VALUE="Register">
</FONT></TD>
</TR>
</TABLE>
</TD></TR></TABLE>
</TD>
</FORM>
</TR>
<TR>
<TD COLSPAN="2" ALIGN="center" VALIGN="bottom">
<BR>All field entries must be 5 to 15 alphanumeric characters (no symbols or
spaces) .
</TD>
</TR>
</TABLE>
</FORM>
</CENTER>
</BODY>
</HTML>

LOGINPAGE
}

```

APPENDIX C

```
#!/usr/local/bin/perl5
#####
#
#           Perl NetLearn! Select Page, by Ted O'Neill
#
#   This program outputs the module status summary page showing all modules.
#
#####

# set to 1 for verbose debugging output
$debug = 0;

# start HTML output page
&startPage;

# start processing
if($ENV{'CONTENT_LENGTH'}){
    print("<P>Calling sub parseForm...\n")if($debug);
    &parseForm;    # parse form information into associative array
}else{
    print("<P>Nothing posted! Quitting.\n")if($debug);
    &done;        # nothing posted, so quitting!
}

# check debugging option
if($data{'debug'}){$debug = 1;} # turn on debugging if requested
print("<P>Debugging info will be displayed...\n")if($debug);

# username and form name must always be present
if(!$data{'username'} || !$data{'form'}){
    print("<P>Username or form name not present. Quitting...\n")if($debug);
    print("<H2 ALIGN='center'><FONT COLOR='\"#0000FF\">");
    print("Error: Invalid submission!</H2></FONT>\n");
    &done;
}

# process form (from successful login or a module screen)
if($data{'form'} eq "welcome"){
    print("<P>Calling sub welcome...\n")if($debug);
    &welcome;
}
```

```

}elseif($data{'form'} ne "summary"){
    print("<P>Unknown form! Quitting...\n")if($debug);
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">");
    print("Error: Invalid submission!</H2></FONT>\n");
    &done;
}

# display module status summary table
print("<P>Calling sub summary...\n")if($debug);
&summary;

# close HTML document and exit program
print("<P>All done. Exiting...\n")if($debug);
&done;          # end program

##### subroutines #####

# start HTML output
sub startPage{
    print("Content-type: text/html\n\n");
    print("<HTML>\n<HEAD>\n<TITLE>");
    print("NetLearn! Module Status Summary</TITLE>\n</HEAD>\n");
    print("<BODY BGCOLOR=\"\#EEEEEE\">\n<CENTER>\n");
}

# parse posted form data from any form #####
sub parseForm{
    read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'}); # read urlencoded data from stdin
    @pairs = split(/&/, $buffer); # store name/value pairs
    foreach $pair (@pairs) {      # process each pair
        ($name, $value) = split(/=/, $pair);
        $name =~ tr/+//;          # replace encoded spaces
        $value =~ tr/+//;
        $value =~ s/%([a-f0-9][a-f0-9])/pack("C", hex($1))/ieg; # replace hex encoded
        $data{$name} = $value;    # store name/value pairs in associative array
    }
}

# check for valid name and email and update user's file #####
sub welcome{
    if( !($data{'name'} =~ /^[a-zA-Z -'\.]{5,20}$/) ){
        print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">");
    }
}

```



```

    print("Missing or invalid name!</H2></FONT><P>Please try again.\n");
    &done; # name too short or long, or contains illegal characters
}
if ( !($data{'email'} =~ /\@.+./) ){
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
    print("Missing or invalid email address!</H2></FONT><P>Please try again.\n");
    &done; # email missing or doesn't contain at sign followed by a period
}
if(-s "../tone/net/$data{'username'}.dat"){&getUserInfo;} # get user's info
if( $info[0] ne "$data{'name'}\t$data{'email'}\n" ){ # check for changes
    $info[0] = "$data{'name'}\t$data{'email'}\n"; # store new info
    @newInfo = @info; # copy updated info
    print("<P>New user info: @newInfo\n")if($debug);
    &saveUserInfo; # update user's file
}
}

# get user's personal info and store in array @info
sub getUserInfo{
    `chmod 400 ../tone/net/$data{'username'}.dat`; # allow file access
    open(USER, "../tone/net/$data{'username'}.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not open user's file</H2></FONT>\n") && &done;
    @info = <USER>;
    close(USER)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not close user's file</H2></FONT>\n") && &done;
    `chmod 000 ../tone/net/$data{'username'}.dat`; # restrict file access
    `chmod 666 ../tone/net/$data{'username'}.dat` if($debug); # for manual access
}

# save user's personal info
sub saveUserInfo{
    `chmod 600 ../tone/net/$data{'username'}.dat`; # allow file access
    open(USER, ">../tone/net/$data{'username'}.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not open user's file</H2></FONT>\n") && &done;
    print USER (@newInfo);
    close(USER)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not close user's file</H2></FONT>\n") && &done;
    `chmod 000 ../tone/net/$data{'username'}.dat`; # restrict file access
}

```

```

`chmod 666 ../tone/net/$data{'username'}.dat` if($debug); # for manual access
}

# display summary page #####
sub summary{
    &getModInfo; # get module info from modules.dat and store in @modules
    &getUserInfo; # get user info from user's file and store in @info
    print("<P> info: ", @modules)if($debug);
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
    print("NetLearn! Module Status Summary</FONT></H2>\n");
    print("<P>Start of showAll form...\n")if($debug);
    # start of form "showAll"
    print("<FORM METHOD=\"POST\" ACTION=\"../cgi/299modout.cgi\">\n");
    print("<TABLE BORDER=\"1\" CELLPADDING=\"5\">\n");
    print("<TR VALIGN=\"bottom\">");
    print("<TD ALIGN=\"center\"><B>Number</B>");
    print("<TD ALIGN=\"center\"><B>Module Title</B>");
    print("<TD ALIGN=\"center\"><B>Max Score</B>");
    print("<TD ALIGN=\"center\"><B>Your Score</B>\n");
    print("<TD ALIGN=\"center\"><B>Select</B>\n");
    print("<TR><TD COLSPAN=\"5\">\n");
    print("<TR><TD>Starting loop...\n")if($debug);
    for($i=1; $i<(@modules); $i++){
        chop($modules[$i]); # remove newline character
        ($modTitle, $modMax) = split(/\t/, $modules[$i]); # parse module info
        $maxTotal += $modMax; # add current max score to total max
        print("<TR><TD ALIGN=\"center\">$i<TD>$modTitle");
        print("<TD ALIGN=\"center\">$modMax"); # print module info
        if(!$info[$i]){ # for first time, add newline chars to file
            ($info[$i] = "0\n") && ($newInfo[$i] = "0\n") && ($infoChange = 1);
        }
        chop($info[$i]); # remove newline
        ($oldScore, @answers) = split(/\t/, $info[$i]);
        $userTotal += $oldScore; # add current module score to user total
        print("<TD ALIGN=\"center\">$oldScore");
        print("<TD ALIGN=\"center\"><INPUT TYPE=\"radio\" NAME=\"module\"");
        print("VALUE=\"$i\">");
    }
    print("<TR><TD COLSPAN=\"5\">\n");
    print("<TR><TD COLSPAN=\"2\" ALIGN=\"right\"><B>Point Totals: </B>");
    print("<TD ALIGN=\"center\"><B>$maxTotal</B>");
    print("<TD ALIGN=\"center\"><B>$userTotal</B>\n");
}

```

```

print("<TD ALIGN=\"center\"><INPUT TYPE=\"submit\" VALUE=\"Start\">\n");
print("</TABLE>\n");
print("<INPUT TYPE=\"hidden\" NAME=\"form\" VALUE=\"topics\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"username\"");
print(" VALUE=\"\$data{'username'}\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"topics\"");
print(" VALUE=\"all\">\n</FORM>\n");
if($infoChange){&saveUserInfo;} # update user's file if info has changed
}

# get module summary info and store in @modules
sub getModInfo{
print("<P>Called sub getModInfo...\n")if($debug);
open(MOD, "../tone/net/modules.dat")
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
        Error: could not modules data file</H2></FONT>\n") && &done;
@modules = <MOD>;
close(MOD)
    || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
        Error: could not modules data file</H2></FONT>\n") && &done;
}

# close HTML document and exit program #####
sub done{
print("</CENTER>\n</BODY>\n</HTML>\n");
exit;
}

```

APPENDIX D

```
#!/usr/local/bin/perl5
#####
#
#           Perl NetLearn! Module Output Page, by Ted O'Neill
#
#   This program outputs the module detail page containing course content.
#
#####

# set to 1 for verbose debugging output
$debug = 0;

# start HTML output page
&startPage;

# start processing
if($ENV{'CONTENT_LENGTH'}){
    print("<P>Calling sub parseForm...\n")if($debug);
    &parseForm;    # parse form information into associative array
}else{
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
    print("Error: Nothing posted!</H2></FONT>\n");
    &done;        # nothing posted, so quitting!
}

# check debugging option
if($data{'debug'}){$debug = 1;} # turn on debugging if requested
print("<P>Debugging info will be displayed...\n")if($debug);

if($debug){
    foreach $key (keys(%data)){
        print("<P>$key: $data{$key}\n");
    }
}

# username, module number, and form name must always be present
if(!$data{'username'} || !$data{'module'} || !$data{'form'}){
    print("<P>Username, module number, or form name not present!")if($debug);
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
}
```

```

    print("Error: Invalid submission!</H2></FONT>\n");
    print("<CENTER>Make sure you've selected a module to begin...</CENTER>\n");
    &done;
}

# process forms
if($data{'form'} eq "topics"){ # display module topics and text
    print("<P>Calling sub showTopics...\n")if($debug);
    &showTopics;
}elseif($data{'form'} eq "test"){ # display module test questions
    print("<P>Calling sub showTest...\n")if($debug);
    &showTest;
}elseif($data{'form'} eq "score"){ # score quiz
    print("<P>Calling sub scoreTest...\n")if($debug);
    &scoreTest
}else{
    print("<P>Unknown form! Quitting...\n")if($debug);
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
    print("Error: Invalid submission!</H2></FONT>\n");
    &done;
}

# close HTML document and exit program
print("<P>All done. Exiting...\n")if($debug);
&done;          # end program

##### subroutines #####

# start HTML output
sub startPage{
    print("Content-type: text/html\n\n");
    print("<HTML>\n<HEAD>\n<TITLE>");
    print("NetLearn! Module Display</TITLE>\n</HEAD>\n");
    print("<BODY BGCOLOR=\"#EEEEEE\">\n");
}

# parse posted form data from any form #####
sub parseForm{
    read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'}); # read urlencoded data from stdin
    @pairs = split(/&/, $buffer); # store name/value pairs
    foreach $pair (@pairs) {      # process each pair
        ($name, $value) = split(/=/, $pair);
    }
}

```

```

$name =~ tr/+// ; # replace encoded spaces
$value =~ tr/+// ;
$value =~ s/%{[a-f0-9]{a-f0-9}}/pack("C", hex($1))/ieg; # replace hex encoded
$data{$name} = $value; # store name/value pairs in associative array
}
}

# display module content for all/selected topics
sub showTopics{
    &getModContent; # load all module info into @sections
    chop(@sections); # remove newline chars
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> NetLearn! Module
        $data{'module'}: $sections[0]</H2></FONT>\n"); # display module title
    $content = 0; # in case no topics are selected
    for($i=1; $i<@sections; $i++){ # process topic lines of module
        $topicNum = "topic$i"; # reference to current topic
        # display requested (or all) topics:
        if( ($data{'topics'} eq "all") || ($data{$topicNum} eq "on") ){
            ($topic, $text, @otherStuff) = split(/\t/, $sections[$i]);
            print("\n<HR>\n<H3>$topic</H3>\n<BLOCKQUOTE>\n$text\n</BLOCKQUOTE>\n");
            $content = 1; # at least one topic was selected for display
        }
    }
    if(!$content){
        print("\n<HR>\n<H3>Nothing Selected...</H3>\n<BLOCKQUOTE>\n");
        print("You didn't select any topics to display!\n</BLOCKQUOTE>\n");
    }
    print("<HR>\n<CENTER>\n");
    print("<TABLE BORDER=\"0\" CELLPADDING=\"10\">\n<TR>\n");
    print("<TD ALIGN=\"center\">\n");
    # form to request module test
    print("\n<FORM METHOD=\"POST\" ACTION=\"../cgi/299modout.cgi\">\n");
    print("<INPUT TYPE=\"submit\" VALUE=\"Test Your Knowledge\">\n");
    print("<INPUT TYPE=\"hidden\" NAME=\"form\" VALUE=\"test\">\n");
    print("<INPUT TYPE=\"hidden\" NAME=\"username\"
VALUE=\"$data{'username'}\">\n");
    print("<INPUT TYPE=\"hidden\" NAME=\"module\" VALUE=\"$data{'module'}\">\n");
    print("</FORM>\n\n");
    # form to request summary page
    print("<TD ALIGN=\"center\">\n");
    print("\n<FORM METHOD=\"POST\" ACTION=\"../cgi/299select.cgi\">\n");
    print("<INPUT TYPE=\"submit\" VALUE=\"Return to Status Summary\">\n");

```

```

print("<INPUT TYPE=\"hidden\" NAME=\"form\" VALUE=\"summary\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"username\"
VALUE=\"$data{'username'}\">\n");
print("</FORM>\n\n");
print("</TABLE>\n");
}

# display test questions
sub showTest{
    &getModContent; # load all module info into @sections
    &getUserInfo;   # get user's last answers if taken before
    chop(@sections); # remove newline chars
    chop(@info);
    ($oldScore, @prev) = split(/\t/, $info[$data{'module'}]); # get prev answers
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> NetLearn! Quiz
        $data{'module'}: $sections[0]</H2></FONT>\n");
    print("\n<FORM METHOD=\"POST\" ACTION=\"../../cgi/299modout.cgi\">\n");
    for($i=1; $i<@sections; $i++){ # process module question lines
        # display questions and choices
        ($stop, $txt, $ques, $ans1, $ans2, $ans3, $ans4, $corr)=split(/\t/, $sections[$i]);
        print("\n<P>$i. $ques\n<UL>\n");
        print("<INPUT TYPE=\"radio\" VALUE=\"1\" NAME=\"topic$i\"");
        if($prev[$i-1] == 1){print(" CHECKED");}
        print("> $ans1<BR>\n");
        print("<INPUT TYPE=\"radio\" VALUE=\"2\" NAME=\"topic$i\"");
        if($prev[$i-1] == 2){print(" CHECKED");}
        print("> $ans2<BR>\n");
        print("<INPUT TYPE=\"radio\" VALUE=\"3\" NAME=\"topic$i\"");
        if($prev[$i-1] == 3){print(" CHECKED");}
        print("> $ans3<BR>\n");
        print("<INPUT TYPE=\"radio\" VALUE=\"4\" NAME=\"topic$i\"");
        if($prev[$i-1] == 4){print(" CHECKED");}
        print("> $ans4\n");
        print("</UL>\n");
    }
    print("<HR>\n<CENTER>\n");
    print("<TABLE BORDER=\"0\" CELLPADDING=\"10\">\n<TR>\n");
    print("<TD ALIGN=\"center\">\n");
    print("<INPUT TYPE=\"submit\" VALUE=\"Check Answers\">\n");
    print("<INPUT TYPE=\"hidden\" NAME=\"form\" VALUE=\"score\">\n");
    print("<INPUT TYPE=\"hidden\" NAME=\"username\"
VALUE=\"$data{'username'}\">\n");

```

```

print("<INPUT TYPE=\"hidden\" NAME=\"module\" VALUE=\"$data('module')\">\n");
print("</FORM>\n\n");
# button for summary page
print("<TD ALIGN=\"center\">\n");
print("\n<FORM METHOD=\"POST\" ACTION=\"../cgi/299select.cgi\" ");
print("onSubmit=\"return confirm('Are you sure? Your progress ");
print("has not been saved!')\">\n");
print("<INPUT TYPE=\"submit\" VALUE=\"Return to Status Summary\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"form\" VALUE=\"summary\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"username\"
VALUE=\"$data('username')\">\n");
print("</FORM>\n\n");
print("</TABLE>\n");
}

```

```

# function to score test, save and display results
sub scoreTest(){
    &getModContent; # load all module info into @sections
    &getUserInfo; # get user's last answers if taken before
    chop(@sections); # remove all newline chars
    chop($info[$data('module')]); # remove newline char from this modules line
    ($oldScore, @prev) = split(/\t/, $info[$data('module')]); # get prev answers
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\"> NetLearn! Quiz
        $data('module'): $sections[0]</H2></FONT>\n");
    print("<CENTER><TABLE BORDER=\"0\" CELLPADDING=\"5\">\n");
    print("<TR><TD COLSPAN=\"2\" ALIGN=\"center\">\n");
    print("\n<FORM METHOD=\"POST\" ACTION=\"../cgi/299modout.cgi\">\n");
    print("<CENTER><TABLE BORDER=\"1\" WIDTH=\"100%\">\n");
    print("<TR ALIGN=\"center\">");
    print("<TD><B>Question</B></TD><TD><B>Result</B></TD><TD><B>Points</B></TD>");
    print("<TD><B>Topic</B></TD><TD><B>Review?</B></TD></TR>\n");
    $score = 0; # start total score count
    for($i=1; $i<@sections; $i++){ # process module question lines
        $topicNum = "topic$i"; # reference to current question topic
        $answers .= "$data($topicNum)\t"; # keep track of user's responses
        ($stop, $txt, $ques, $ans1, $ans2, $ans3, $ans4, $corr)=split(/\t/, $sections[$i]);
        print("<TR ALIGN=\"center\"><TD>$i</TD>");
        if($data($topicNum) == $corr){ # check for correct answer
            $score++; # increment total score
            print("<TD>correct</TD><TD>1</TD><TD>$stop</TD>");
            print("<TD><INPUT TYPE=\"checkbox\" NAME=\"$topicNum\">");
        }else{

```



```

        print("<TD>incorrect</TD><TD>0</TD><TD>$stop</TD>");
        print("<TD><INPUT TYPE=\"checkbox\" NAME=\"\"$topicNum\" CHECKED\"");
    }
    print("></TD></TR>\n");
}
print("</TABLE></TD></TR><TR><TD COLSPAN=\"2\" ALIGN=\"center\">\n");
if($score == (@sections - 1)){
    print("<P>Perfect score! ($score out of $score) You are a true guru.\n");
    print("<BR>Time to try the next module...\n");
}elseif($oldScore && ($score > $oldScore)){
    printf("<P>You got $score out of %d correct -- an ", (@sections - 1));
    printf("improvement over your last score of %d!\n", $oldScore);
    print("<BR>Keep up the good work!\n");
}elseif($oldScore && ($score < $oldScore)){
    printf("<P>You got $score out of %d correct -- a ", (@sections - 1));
    printf("decline from your last score of %d!\n", $oldScore);
    print("<BR>But don't give up. Please keep trying!\n");
}elseif(!$score){
    print("<P>You didn't choose any correct answers...");
    print("<BR>But don't give up. Please keep trying!\n");
}elseif($score == $oldScore){
    printf("<P>You got $score out of %d correct ", (@sections - 1));
    print("(same score as last time).<BR>Keep trying!\n");
}else{
    printf("<P>You got $score out of %d correct.", (@sections - 1));
    print("<BR>Keep up the good work!\n");
}
print("<P></TD></TR><TR ALIGN=\"center\"><TD>\n");
print("<P><INPUT TYPE=\"submit\" VALUE=\"Review Selected Topics\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"form\" VALUE=\"topics\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"username\"
VALUE=\"\"$data{'username'}\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"module\" VALUE=\"\"$data{'module'}\">\n");
print("</FORM></TD>\n\n");
# return to summary button
print("\n<FORM METHOD=\"POST\" ACTION=\"../cgi/299select.cgi\">\n");
print("<TD>\n");
print("<INPUT TYPE=\"submit\" VALUE=\"Return to Status Summary\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"form\" VALUE=\"summary\">\n");
print("<INPUT TYPE=\"hidden\" NAME=\"username\"
VALUE=\"\"$data{'username'}\">\n");
print("</FORM>\n\n");

```

```

print("</TD></TR></TABLE></CENTER>\n");
$info[$data{'module'}] = "$score\t$answers\n"; # refresh user info
&saveUserInfo; # save to user's record file
}

# get modules sections (both topics and questions) and store in @sections
sub getModContent{
    print("<P>Called sub getModContent...\n") if($debug);
    open(FILE, "../tone/net/mod$data{'module'}.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
            Error: could not open module detail file</H2></FONT>\n") && &done;
    @sections = <FILE>;
    close(FILE)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
            Error: could not close module detail file</H2></FONT>\n") && &done;
}

# get user's personal info and store in array @info
sub getUserInfo{
    `chmod 400 ../tone/net/$data{'username'}.dat`; # allow file access
    open(USER, "../tone/net/$data{'username'}.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
            Error: could not open user's file</H2></FONT>\n") && &done;
    @info = <USER>;
    close(USER)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
            Error: could not close user's file</H2></FONT>\n") && &done;
    `chmod 000 ../tone/net/$data{'username'}.dat`; # restrict file access
    `chmod 666 ../tone/net/$data{'username'}.dat` if($debug); # for manual access
}

# save user's personal info from array @newInfo
sub saveUserInfo{
    `chmod 600 ../tone/net/$data{'username'}.dat`; # allow file access
    open(USER, ">../tone/net/$data{'username'}.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
            Error: could not open user's file</H2></FONT>\n") && &done;
    print USER (@info);
    close(USER)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
            Error: could not close user's file</H2></FONT>\n") && &done;
    `chmod 000 ../tone/net/$data{'username'}.dat`; # restrict file access
}

```

```
`chmod 666 ../tone/net/$data{'username'}.dat` if($debug); # for manual access
}

# close HTML document and exit program #####
sub done{
    print("</BODY>\n</HTML>\n");
    exit;
}
```

APPENDIX E

```
#!/usr/local/bin/perl5
#####
#
#       Perl NetLearn! Administration Page, by Ted O'Neill, 1998
#
#   This program presents administrative options. Access is strictly limited!
#
#####

# set to 1 for verbose debugging output
$debug = 0;

# start HTML output page
&startPage;

# start processing
if($ENV{'CONTENT_LENGTH'}){
    print("<P>Calling sub parseForm...\n")if($debug);
    &parseForm;    # parse form information into associative array
}else{
    print("<P>Nothing posted! Quitting.\n")if(debug);
    &done;        # nothing posted, so quitting!
}

# check debugging option
if($data{'debug'}){$debug = 1;} # turn on debugging if requested
print("<P>Debugging info will be displayed...\n")if($debug);

# update Admin personal info
if(($data{'username'} eq "Administrator") && ($data{'form'} eq "welcome")){
    &updateAdminInfo;
}elseif($data{'username'} && ($data{'form'} eq "delete")){
    &deleteUser;
}else{
    print("<P>Unknown form. Quitting...\n")if($debug);
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">");
    print("Error: Invalid submission!</FONT></H2>\n");
    &done;
}
```

```

# display table of users and scores
&showUsers;

# close HTML document and exit program
print("<P>All done. Exiting...\n")if($debug);
&done;          # end program

##### subroutines #####

# start HTML output
sub startPage(){
    print("Content-type: text/html\n\n");
    print("<HTML>\n<HEAD>\n<TITLE>");
    print("NetLearn! Administrative Summary</TITLE>\n</HEAD>\n");
    print("<BODY BGCOLOR=\"#EEEEEE\">\n");
}

# parse posted form data from any form #####
sub parseForm(){
    read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'}); # read urlencoded data from stdin
    @pairs = split(/&/, $buffer); # store name/value pairs
    foreach $pair (@pairs) {      # process each pair
        ($name, $value) = split(/=/, $pair);
        $name =~ tr/+//;          # replace encoded spaces
        $value =~ tr/+//;
        $value =~ s/%([a-f0-9][a-f0-9])/pack("C", hex($1))/ieg; # replace hex encoded
        $data{$name} = $value;     # store name/value pairs in associative array
    }
}

# check for valid name and email and update Administrator's file #####
sub updateAdminInfo(){
    if( !($data{'name'} =~ /^[a-zA-Z -'\.]{5,20}$/) ){ # confirm valid name
        print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
        print("Missing or invalid name!</H2></FONT><P>Please try again.\n");
        &done; # name too short or long, or contains illegal characters
    }
    if ( !($data{'email'} =~ /\@.+./) ){ # confirm valid email
        print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">");
        print("Missing or invalid email address!</H2></FONT><P>Please try again.\n");
        &done; # email missing or doesn't contain at sign followed by a period
    }
}

```

```

if(-s "../tone/net/$data{'username'}.dat"){&getUserInfo;} # get Admin info
if( $userInfo[0] ne "$data{'name'}\t$data{'email'}\n" ){ # check for changes
    $userInfo[0] = "$data{'name'}\t$data{'email'}\n"; # store new info if changed
    @newUserInfo = @userInfo; # copy updated info
    print("<P>New user info: @newUserInfo\n")if($debug);
    &saveUserInfo; # update Admin's file
}
}

# delete specified user
sub deleteUser(){
    &getAllUserInfo; # store current info in @allUserInfo
    chop(@allUserInfo);
    foreach $line (@allUserInfo){ # discard user's line in user info file
        ($username, @misc) = split(/\t/, $line);
        if($username ne $data{'username'}){$allUserNew .= "$line\n";}
    }
    `chmod 600 ../tone/net/users.dat`; # allow file access
    open(FILE, ">../tone/net/users.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
            Error: could not open user info file</FONT></H2>\n") && &done;
    print FILE ($allUserNew); # write modified info to user info file
    close(FILE)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">
            Error: could not close user info file</H2></FONT>\n") && &done;
    `chmod 000 ../tone/net/users.dat`; # restrict file access
    `chmod 666 ../tone/net/users.dat` if($debug); # for manual access
    `rm ../tone/net/$data{'username'}.dat`; # remove user's file
}

# show table of users and scores
sub showUsers(){
    &getAllUserInfo; # get admin info for all users
    if(@allUserInfo < 2){ # make sure some user has registered
        print("<H3>No students registered! Nothing to display.</H3>\n");
        &done; # no users, so quitting
    }
    &getModInfo; # get modules titles and max scores possible
    chop(@modInfo); # remove newline characters
    # start HTML body content
    print("<H2 ALIGN=\"center\"><FONT COLOR=\"\#0000FF\">");
    print("NetLearn! Administrative Summary</FONT></H2>\n");

```

```

print("<FORM METHOD=\"POST\" ACTION=\"../cgi/299admin.cgi\"\\n");#start form
print("    onSubmit=\"return confirm('Are you sure you wish to ');
print("irreversibly delete this account?')\">\\n");
print("<TABLE BORDER=\"1\" WIDTH=\"100%\">\\n"); # start table
print("<TR ALIGN=\"center\">\\n"); # start header row
print("    <TD ROWSPAN=\"2\">Student<BR>Account\\n"); # column 1
printf("    <TD COLSPAN=\"%d\">", (@modInfo-1));
print("Module Number & Points Possible\\n");
print("    <TD>Total Pts.\\n"); # 2nd to last column
print("    <TD ROWSPAN=\"2\">Delete Account\\n</TR>\\n"); # last column
print("<TR ALIGN=\"center\">\\n    ");
for($i=1; $i<@modInfo; $i++){
    ($title, $maxP) = split(/\\t/, $modInfo[$i]); # get title and pts for this mod
    print("<TD>$i.<BR>$maxP pts.");# mod info
    $totalMaxP += $maxP;
}
print("<TD>$totalMaxP\\n</TR>\\n");
$numUsers = 0; # keep track of how many users with data
for($i=1; $i<@allUserInfo; $i++){
    $totalScore = 0; # reset total score for each user
    print("<TR ALIGN=\"center\">\\n"); # start row for each user
    ($data{'username'}, @misc) = split(/\\t/, $allUserInfo[$i]); # get username
    if(-s "../tone/net/$data{'username'}.dat"){
        &getUserInfo;
        chop(@userInfo); # remove newline characters
    }else{ # output default info if this user's file is empty
        print("    <TD>No data for<BR>account \"$data{'username'}\"\\n");
        printf("    <TD COLSPAN=\"%d\">NA\\n", (@modInfo/1)); # module and total
columns
        print("    <TD><INPUT TYPE=\"radio\" NAME=\"username\"");
        print(" VALUE=\"$data{'username'}\">\\n</TR>\\n");
        next; # skip to next user
    }
    $numUsers++; # increment user count
    ($name,$email)=split(/\\t/, $userInfo[0]);
    print("    <TD>$name<BR>\"$data{'username'}\"\\n    ");
    for($j=1; $j<@userInfo; $j++){ # get this user's info and score for each mod
        ($score, @misc)=split(/\\t/, $userInfo[$j]);
        print("<TD>$score"); # show score for each module
        $totalScore += $score;
        $allUserTotal += $score;
        $modTotal{$j} += $score;

```

```

    }
    print("<TD>$totalScore\n"); # show total score for all modules
    print(" <TD><INPUT TYPE=\"radio\" NAME=\"username\"");
    print(" VALUE=\"${data('username')}\">\n</TR>\n");
}
print("<TR ALIGN=\"center\">\n <TD>Averages");
for($i=1; $i<@modInfo; $i++){
    printf("<TD>%.2f", ($modTotal{$i}/$numUsers) );
}
printf("<TD>%.2f\n", ($allUserTotal/$numUsers) );
print(" <TD><INPUT TYPE=\"submit\" VALUE=\"Delete\">\n</TR>\n</TABLE>\n");
print("<INPUT TYPE=\"hidden\" NAME=\"form\" VALUE=\"delete\">\n");
print("</FORM>");
}

# get personal info and store in array @userInfo (for single user)
sub getUserInfo{
    `chmod 400 ../tone/net/${data('username')}.dat`; # allow file access
    open(FILE, "../tone/net/${data('username')}.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not open Admin's file</H2></FONT>\n") && &done;
    @userInfo = <FILE>;
    close(FILE)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not close user's file</H2></FONT>\n") && &done;
    `chmod 000 ../tone/net/${data('username')}.dat`; # restrict file access
    `chmod 666 ../tone/net/${data('username')}.dat` if($debug); # for manual access
}

# save personal info to file (for single user)
sub saveUserInfo{
    `chmod 600 ../tone/net/${data('username')}.dat`; # allow file access
    open(FILE, ">../tone/net/${data('username')}.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not open user's file</H2></FONT>\n") && &done;
    print FILE (@newUserInfo);
    close(FILE)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not close user's file</H2></FONT>\n") && &done;
    `chmod 000 ../tone/net/${data('username')}.dat`; # restrict file access
    `chmod 666 ../tone/net/${data('username')}.dat` if($debug); # for manual access
}

```



```

# get all user info and store in array @allUserInfo
sub getAllUserInfo{
    `chmod 400 ../tone/net/users.dat`; # allow file access
    open(FILE, "../tone/net/users.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not open user info file</H2></FONT>\n") && $done;
    @allUserInfo = <FILE>;
    close(FILE)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not close user info file</H2></FONT>\n") && $done;
    `chmod 000 ../tone/net/users.dat`; # restrict file access
    `chmod 666 ../tone/net/users.dat` if($debug); # for manual access
}

# get module summary info and store in @modules
sub getModInfo{
    print("<P>Called sub getModInfo...\n")if($debug);
    open(FILE, "../tone/net/modules.dat")
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not modules data file</H2></FONT>\n") && $done;
    @modInfo = <FILE>;
    close(FILE)
        || print("<H2 ALIGN=\"center\"><FONT COLOR=\"#0000FF\">
            Error: could not modules data file</H2></FONT>\n") && $done;
}

# close HTML document and exit program #####
sub done{
    print("</BODY>\n</HTML>\n");
    exit;
}

```

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved