Additional Topics —TOC

The Domain Name Service (DNS)

The Internet doesn't use `/etc/hosts` for hostname-address resolution. It uses DNS, which maintains a huge distributed database of these mappings. The system owes it origin to Paul Mockapetris, who wrote its specification and first implementation. However, Kevin Dunlap later wrote the most popular implementation called BIND (Berkeley Internet Name Domain) for Berkeley UNIX. BIND 9 is now shipped with practically all UNIX machines.

Apart from maintaining the hostname-address database, BIND also specifies the *mail exchangers* (servers that handle mail) for a domain. It's not immediately obvious which host receives a message addressed to *henry@heavens.com*. This is because *heavens.com* is not always a hostname, but even if it is, that host may not receive mail.

BIND specifies one or more mail exchangers in order of preference so that the hosts are tried out in the right order. Further, if BIND is not able to resolve an FQDN, it must be able to refer the problem to some other host running the same service.

How DNS Works

DNS divides the Internet namespace into *zones* with corresponding delegation of authority for that zone. The administrator of every zone has the responsibility of maintaining one or more *name servers*—databases containing the name-address information of that zone. The *master* (or primary) name server contains the latest information. The *slave* (or secondary) name server obtains it from the primary through a *zonal transfer*. It also serves as a backup in the event of failure of the primary. There is a third type of name server—caching-only, which merely diverts queries to a special group of 13 master name servers.

The difference between a domain and a zone is quite subtle but must be understood. Let's consider that a domain *birds.edu* is further divided into two sub-domains, *parrots.birds.edu* and *cuckoo.birds.edu*. The administrator at *birds.edu* has decided to delegate the management of *cuckoo.birds.edu* while retaining it for *parrots.birds.edu*. There are two distinct zones here—*cuckoo.birds.edu* and *birds.edu*. Here, the *birds.edu* zone includes the *parrots.birds.edu* sub-domain, but not *cuckoo.birds.edu* The zones together present a unified picture of the entire domain in the same way multiple file systems combine to form a single file system.

Every zone has its own group of name servers (master and slave), and the answers they provide are *authoritative* (correct) for that zone. A name server is queried by a resolver, which acts on behalf of the application to obtain the IP address of a host. It is not a separate program by itself, but a set of library routines that are linked into the applications like **`telnet`** and **`ftp`**. Usually, multiple name servers have to be queried to obtain an IP address. This is a job best done by a name server, so the resolver depends on the local name server for doing all the work.

When queried, a name server may be able to perform the resolution, but if it can't, it's not designed to give up. It must be able to provide the IP address (a referral) of another name server that would take you one step "closer" to the desired host. If this server can't provide the answer either, it too has to refer the matter to yet another one. The client-server architecture in DNS ensures that the referrals and the linkages between zones are properly exploited till the address is finally resolved.

In the above *birds.edu* domain, if the name server for the *birds.edu* zone is queried for the IP address of a host in the *cuckoo.birds.edu* zone, it obviously wouldn't know the answer. But it must be able to provide the IP address of the name server for the *cuckoo.birds.edu* zone that will know the answer.

The resolution follows a hierarchical pattern. If the local server can't provide the address of *www.cuckoo.birds.edu* (here, *www* is the host), it will check from its records whether it knows the name servers for *cuckoo.birds.edu*, then *birds.edu*, and so forth. At some point this has to stop, and the buck stops at the *root* name servers. These servers know the name servers of all the top-level domains like *com*, *edu*, *ca*, *gb* and so forth. All name servers on the Internet have the IP addresses of the root name servers, and will directly contact them when local resolution efforts fail.

Now consider that a host queries the local name server at *cuckoo.birds.edu* to know the address of *www.heavens.com*. It obviously won't know and will contact the root name servers directly. One of them will return the IP addresses of the *com* name servers. The local name server now queries the *com* name server, which then refers it to the *heavens.com* name server. This server knows the address of *www.heavens.com* and returns the answer. The whole thing happens fast in spite of the fact that each of the root name servers handles several thousand queries every second. (There are just 13 of them, and they all run BSD UNIX.)

Some of the root name servers not only provide the IP addresses of the name servers of the top-level domains, but are actually name servers of these domains.

The Shell: Evaluating Twice with eval

The **eval** statement postpones evaluation of portions of the command line. This feature can be used to create numbered prompts:

```
$ prompt1="User Name: "
$ x=1
$ eval echo \$prompt$x
User Name:
```

The first pass ignores the $ escaped with a \; this evaluation results in \$prompt1. The second pass ignores the \ and evaluates $prompt1 as a variable. This is exactly what we want.

Let's now consider a useful application that uses multiple numbered variables. If a script has to take input from the terminal 10 times, without **eval** you would need to define and use 10 variables to read the input. Sometimes, you won't even know the number of variables that are required at run time. We would prefer to have a more general script where the variable name itself can be generated by the script on-the-fly. We would store the prompts as variables and read the input into "numbered variables."

We have seen how to create a numbered prompt. We now need to use numbered variables like value1, value2, value3, and so on to hold the input:

```
$ { x=1
> eval echo \$prompt$x '\\c'
> read value$x                                    OK—no problems
> eval echo \$value$x ; }
User Name: kleinrock
kleinrock
```

The statement **read value$x** reads the response into the variable value1; **eval** makes two passes to echo this value.

Using eval to Create a User

We'll now develop a script to create a user with the **useradd** command that is discussed in Chapter 19. This script requires root permission and should have the following features:

- Interactively accept all six arguments required by **useradd**.
- Accept these inputs into six fields using a generalized prompt and variable.
- Automatically invoke the -e option with **echo** if the script is run by Bash.

Before we discuss the main script, let's define a function, **anymore**, that seeks user confirmation to perform the next loop iteration. This function was discussed in Section 13.18 and is shown here after a minor modification:

```
anymore() {
     echo $option "\n$1 ?(y/n) : \c" 1>&2
     read response
     case "$response" in
       y|Y) echo  1>&2
            return 0 ;;
         *) return 1 ;;
     esac
}
```

Place the script in the file **mainfunc.sh** which we'll source from the main script using the dot command. Since we'll have to prompt the user six times, we can use the services of **eval** in producing a compact script. The **createuser.sh** script is shown below:

```
# cat createuser.sh
#!/bin/sh
# Program to create user -- uses eval to create prompts and variables
# Will run in the Korn and Bash shells without modification
option= ;
[ $SHELL = "/bin/bash" ] && option=-e
. ./mainfunc.sh                        # Makes anymore() function available

prompt1="User Name:"    ; prompt2="User-id:"
prompt3="Group-id:"     ; prompt4="Home Directory:"
prompt5="Login Shell:"  ; prompt6="GCOS Details:"

while true ; do
    x=1
    while [ $x -le 6 ] ; do
        eval echo $option \$prompt$x '\\c'         # Displays 6 prompts
        read value$x
        x=`expr $x + 1`
    done
    useradd -u $value2 -g $value3 -d $value4 -s $value5 \
            -c "$value6" -m $value1
    anymore "More users to create" || break
done
```

The current directory is generally absent in the super user's PATH, and the dot command (in all shells except Bash) doesn't look for the script in the current directory. That's why we had to use a relative pathname with the dot command.

The six prompts are defined at the beginning of the script as numbered variables. The inner **while** loop issues all of them in turn. The values are read into the variables value1, value2, and so on. We'll now create a user ppp:

```
# ./createuser.sh                          Required as . is not in root's PATH
User Name: ppp
User-id: 520
Group-id: 100
Home Directory: /home/ppp
Login Shell: /etc/ppp/ppplogin
GCOS Details: PPP Server Account
More users to create ?(y/n) : n
```

This should have created a new user account, but to be sure, let's observe the last line of /etc/passwd:

```
# tail -n 1 /etc/passwd
ppp:x:520:100:PPP  Server  Account:/home/ppp:/etc/ppp/login
```

Thanks to **eval**, we have managed to read six user responses into six variables with a minuscule script. As **echo** automatically uses the -e option if Bash is the user's login shell, the script will run without modifications in all shells.

The exec Statement

You have used **exec** to replace the current shell with a new program (Section 13.21). But **exec** has another important property: it can return you a file descriptor by opening a disk file. For example, if a script has several commands whose standard output go to a single file, then instead of using separate redirection symbols for each, you can use **exec** to reassign the default destination like this:

```
exec 1> found.lst                                    Can use >> also
```

Apart from the descriptors 0, 1 and 2, **exec** can create a descriptor 3 for directing all output and associate it with a physical file `foundfile`:

```
exec 3>foundfile
```

Once the file has been opened, you can now replicate the standard output stream to write to file descriptor 3:

```
echo "This goes to foundfile" 1>&3
```

This makes file handling a lot simpler, so let's design a script which reads emp-ids from a file. It then searches `emp.lst` (introduced in Section 11.1) and saves in three separate files the following:

- The lines found.
- The emp-ids not found.
- Badly formed emp-ids.

First, let's examine the file containing the emp-ids. It contains two three-digit empids which should be trapped by the script:

```
$ cat empid.lst
2233
9765
2476
789
1265
9877
5678
245
2954
```

The **countpat.sh** script shown below generates three output streams and redirects them to three separate files:

```
$ cat countpat.sh
#!/bin/sh
# Creates file descriptors 3 and 4
#
[ $# -ne 4 ] && { echo "4 arguments required" ; exit 2 ; }

exec > $2                          # Descriptor 1 stores selected lines
exec 3> $3                         # Descriptor 3 stores patterns not found
exec 4> $4                         # Descriptor 4 stores invalid patterns
exec < $1                          # Descriptor 0 reads from file

while read pattern ; do
      case "$pattern" in
         ????) grep $pattern emp.lst ||
               echo $pattern not found in file 1>&3 ;;
```

```
            *) echo $pattern not a four-character string 1>&4 ;;
     esac
done

exec >/dev/tty                      # Assigns standard output to terminal file
echo Job Over
```

The script requires four arguments: the file containing the patterns and the files associated with the three streams. By using the stream replicating symbols >&, two **echo** commands actually write to descriptors 3 and 4.

Note that we have set `$1` as the source of standard input. This means that the **read** statement in the loop will take input from `$1`, the file containing the patterns. Once all write operations are over, the standard output stream has to be reassigned to the terminal (**exec >/dev/tty**), otherwise the message `Job Over` will also be saved in the filename passed to `$2`.

The script takes four arguments and diverts the output into three files:

```
$ countpat.sh empid.lst foundfile notfoundfile invalidfile
Job Over
```

The message appears on the terminal instead of going to any of these files. Now, just have a look at the three files and see for yourself what has actually happened:

```
$ cat foundfile
2233:charles harris   :g.m.      :sales      :12/12/52: 90000
2476:jackie wodehouse:manager   :sales      :05/01/59:110000
1265:p.j. woodhouse   :manager   :sales      :09/12/63: 90000
5678:robert dylan     :d.g.m.    :marketing  :04/19/43: 85000

$ cat notfoundfile
9765 not found in file
9877 not found in file
2954 not found in file

$ cat invalidfile
789 not a four-character string
245 not a four-character string
```

This then is the power of **exec**. It opened several files and accessed each one separately in the same way **perl** uses its own filehandles.

Using perl with CGI

perl is ideally suited for use in CGI programming, so we'll develop an application to do two simple things:

- Filter form data.
- Generate HTML.

The CGI application also uses a similar employee database that was presented in Section 11.1 except this time we'll access this database from our Web browser. Before doing that, you need to know some HTML to understand the significance of its tags. You should then be able to make the CGI **perl** program generate the right tags when it sends back data to the browser. The HTML code presented below specifies a form with the `<form>` tag:

```
$ cat emp_form.html
<html>  <head>
            <title>The Employee Database</title>              Appears on title bar
        </head>
        <body>
             <h1> Employee Form </h1>                         Appears in a large bold font
            <hr>                                              Adds a horizontal rule
          <form action="http://localhost/~sumit/cgi-bin/emp_add.pl" method=get>
             Emp-id: <input type="text" name="empid" size=4> <br>
             Name: <input type="text" name="ename" size=30> <br>
             Designation: <input type="text" name="desig" size=15> <br>
             Department: <input type="text" name="dept" size=15> <br>
             Date of birth: <input type="text" name="dtbirth" size=10> <br>
             Salary: <input type="text" name="salary" size=10> <br> <br>
            <center>
                <input type=submit value="Add">               The Add button is centered
            </center>
        </form>
        </body>
</html>
```

The `action` attribute of this tag specifies a URL pointing to a **perl** program (**emp_add.pl**) on the server. This program adds a line to the database. The form that accepts user input is shown below:

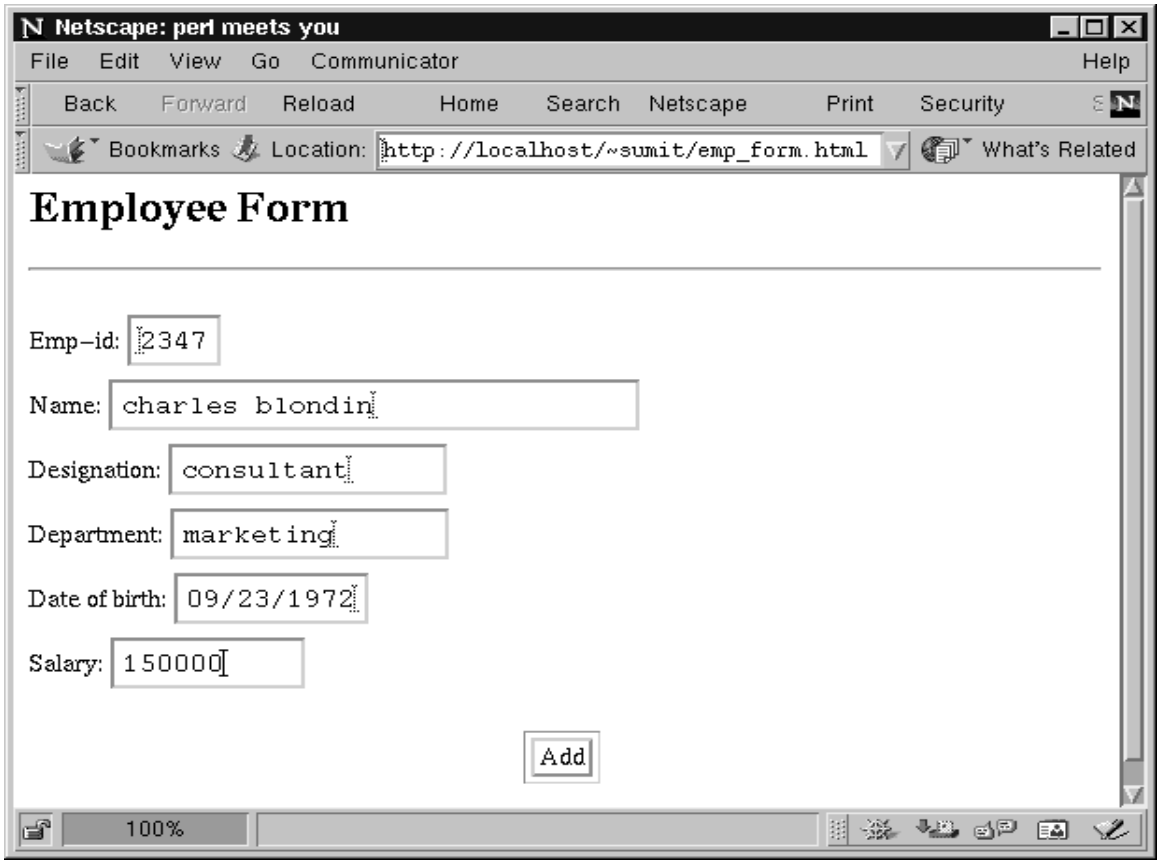
Every HTML document consists of some header and footer code. Since **perl** has to generate these lines in most CGI applications, we'll create two subroutines to be used by our CGI programs.

The body of the HTML document specifies a form enclosed by the `<form>` and `</form>` tags. There are six text boxes here which accept the six fields of the employee database. The values entered into these fields are paired with their corresponding variable names—`empid`, `ename`, `desig`, and so on. The `<br>` tag is required to place each text box in a separate line.

The `action` attribute of the `<form>` tag specifies a URL on the localhost itself. This URL points to a **perl** program **emp_add.pl** in `cgi-bin` (the directory where CGI programs are generally kept). The program is executed the moment the button labeled *Add* and of type `submit` is clicked with the mouse.

The Query String

The browser sends data to the server through its request header. To understand how form data is structured, consider a form that has only three fields with names `empid`, `ename` and `desig` (the `name` attribute of the `<input>` tag). Let's put the values `1234`, `henry higgins` and `actor` into these three fields. On submission, the browser strings together the entire data as *name*=*value* pairs into a query string in this manner:



emp_form.html: An HTML form viewed by Netscape

```
empid=1234&ename=henry+higgins&desig=actor
```

This single string is sent to the server specified in the URL. The `&` here acts as the delimiter of each *name=value* pair. Note that the browser has encoded the space character to a `+`.

To be able to use this data, **`perl`** has to split this string twice—once to extract all *name=value* pairs and then to separate the names from their values. This can be done in two ways depending on the method specified.

GET and POST: The Request Method

The `<form>` tag shows another attribute: `method`. This signifies the way data is transmitted to the server. Generally, the query string shown above is sent in two ways:

- **GET** This method appends the query string to the URL using the `?` as the delimiter. With this string, the URL will now look like this:

  ```
  http://localhost/cgi-bin/emp_add.pl?empid=1234&ename=henry+higgins&desig=actor
  ```

The server parses the `GET` statement in the request header and stores the data following the `?` in its environment variable, `QUERY_STRING`. This variable can be used by any CGI program.

- **POST** With this method, the browser precedes this string with a number signifying the number of characters the string holds. The server stores this number in the `CONTENT_LENGTH` variable. It

supplies the string as standard input to the CGI program. **perl** reads this data with its **read** function, and reads just as much as specified by CONTENT_LENGTH.

The method itself is available as the REQUEST_METHOD variable in the server's environment. Our sample HTML form uses GET as the method. GET has the limitation that the string size is restricted to 1024 characters. If you have a lot of data to transmit, then use POST. However, the structure of the query string is the same in both cases, so the **emp_add.pl** script should be able to handle the data using both methods. There was no compelling reason to choose GET rather than POST for this example.

The CGI program, **emp_add.pl** (shown later), has to parse the data in QUERY_STRING (for GET) or STDIN (for POST). It then has to combine the extracted data into a single line and add it to a text file acting as the database. To better understand what's going on, we'll print the contents of the important CGI environment variables on the browser window. The CGI program must be able to generate the HTML required to display these messages.

Creating the Subroutines for Header and Footer

Since all HTML documents have a common header and footer segment, let's first frame two subroutines for them. **Htmlheader** prints the header:

```
sub Htmlheader {
   local ($title, $h1) = @_;
    print << "MARKER";                          A here document
    <html>
        <head>
            <title> $title </title>             Variable substitution enabled
        </head>
        <body>
        <h1> $h1 </h1>
MARKER                                          Don't indent this!
}
```

Htmlheader accepts two arguments into the placeholders $title and $h1. This means you have the option of specifying the title and first level header when calling the subroutine. Here, we have used a single **print** statement like a here document. The double quotes surrounding the marker tag ensure that variable substitution is enabled (required for $title and $h1).

The subroutine for the footer is simpler still:

```
sub Htmlfooter {
    print "</body></html>\n";
}
```

Place these two subroutines (and another one that we'll be discussing shortly) in a separate file, **web_lib.pl**. They will be required at runtime by the CGI program. Make sure you add the statement 1; at the end of the file so that it always returns a true value.

emp_add.pl: The Main CGI Program

Before we take up the third subroutine, let's have a look at the CGI program **emp_add.pl** specified in the URL. Apart from printing some messages on the browser window, the program appends a line built from form data to a text database:

```
$ cat emp_add.pl
#!/usr/bin/perl
#
require "web_lib.pl" ;

open (OUTFILE, ">>/home/sumit/public_html/emp_out.lst") ;
&Parse(*field) ;
print "Content-type: text/html\n\n";
&Htmlheader("Testing Query String", "The QUERY_STRING Variable") ;

print "The query string is $ENV{'QUERY_STRING'}<br>\n" ;
print "The method of sending data to server is $ENV{'REQUEST_METHOD'}<br>\n" ;
print "THE content length is $ENV{'CONTENT_LENGTH'}<br>\n" ;
print  OUTFILE  "$field{'empid'}|$field{'ename'}|$field{'desig'}|$field{'dept'}|
$field{'dtbirth'}|$field{'salary'}<br>\n"  ;
print "A record has been added <a href=\"http://localhost/cgi-bin/emp_query.pl\"
>Click here to see the records</a><br>\n" ;

&Htmlfooter ;
close (OUTFILE);
```

Since this program generates HTML, it has to explicitly spell out its Content-Type and then leave a blank line (`\n\n`) before sending back data. The HTML header is then printed with the **`Htmlheader`** subroutine. The footer is printed at the end with **`Htmlfooter`**.

Observe that the array `%field` is passed by reference with the `*` prefix to the **`Parse`** subroutine (discussed next). The next three **`print`** statements following `&Htmlheader` display the contents of the server's environment variables on the browser window. The fourth one uses the filehandle `OUTFILE` to add a line to the database (the file `emp_out.lst`), using the `|` as the field delimiter. The final **`print`** statement prints a completion message and offers a hyperlink (with `A HREF`) to the **`emp_query.pl`** program. You should be able to see all lines of `emp_out.lst` when you click on this link.

The child HTTP process that communicates form data to the server runs as an ordinary user. For the process to be able to create the file `emp_out.lst`, the directory `public_html` must be world-writable (with **`chmod 777`**). This is necessary for entering the first detail. Once the file is created, the directory can have its old permissions.

The `Parse` Subroutine

`Parse` is the third subroutine that we need to use here. **`Parse`** makes each *name=value* pair available as separate entries in the associative array, `%field`. To understand how **`perl`** makes the form values available to the main program, we need to study this subroutine:

```
sub Parse {
  local (*in) = @_ ;
  local ($i, $key, $val) ;                          # Local variables

  if ($ENV{'REQUEST_METHOD'} eq "GET") {            # Takes care of both GET
      $in = $ENV{'QUERY_STRING'} ;
  } elsif ($ENV{'REQUEST_METHOD'} eq "POST") {      # ... and POST
      read(STDIN, $in, $ENV{'CONTENT_LENGTH'}) ;
  }                                                  # Query string in $in

  @in = split(/&/, $in) ;                         # Break up into name=value pairs
  foreach $i (0 .. $#in) {
     $in[$i] =~ s/\+/ /g ;                          # Decode a + to a space
     ($key, $val) = split(/=/, $in[$i], 2) ;        # Splits on the first =
```

```
        $key =~ s/%(..)/pack("c",hex($1))/ge ;
        $val =~ s/%(..)/pack("c",hex($1))/ge ;
       $in{$key} = $val ;                    # Name and value in associative array
    }
    return %in ;
}
```

Parse here accepts an array by reference. This array is copied to `%in` inside the subroutine. We used the associative array `%ENV` to evaluate the server's three environment variables we have previously discussed. The query string is assigned to the variable `$in` irrespective of the method used. POSTed data is also read into `$in`, but from standard input (`STDIN` is the filehandle) with the **read** function. The number of characters read is determined by **read**'s third argument (the content length).

The query string uses the `&` as the delimiter of the *name=value* pairs. The first **split** stores every such pair in the scalar array, `@in`. The **s** function decodes every `+` in the string to a space as encoding (space to a `+`) takes place whenever there are spaces in the data. Many characters are encoded into hexadecimal strings because they have special significance in the URL string. For instance, the `/` that separates the elements of the date field is used to delimit directories in the URL string. As you can see in the figure below, it is encoded to `%2F` before the query string is sent to the server.

The **pack** function converts these hex values back to their original ASCII characters. Note how the **s** function identifies these characters with the pattern `%( . . )` that uses a TRE. The `ge` flags ensure that **pack** is interpreted as an expression and not treated literally.

The **foreach** loop picks up each *name=value* pair from the array `@in`. After decoding, each element of the array is split again, this time on the `=`, and stored in the variables `$key` and `$val`. The first is set as the key and the other as the value in the associative array `%in`. This array is returned to the calling program. Note that in this program we used `in` as a variable (`$in`), as a scalar list (`@in`) and an associative array (`%in`) without conflict.

Output of CGI program **emp_add.pl**

emp_query.pl: The Query Program

The figure below shows the output of **emp_add.pl** on the browser window after you have pressed the *Add* button of the form shown previously. Note the hyperlink which offers to show all lines of the database. A click here runs the **emp_query.pl** program and displays the list of people as elements of a table. Here's the listing of the program:

```
$ cat emp_query.pl
#!/usr/bin/perl
require "web_lib.pl" ;

open (OUTFILE, "/home/sumit/public_html/emp_out.lst") ;
print "Content-type: text/html\n\n";
&Htmlheader("Retrieving from Database", "Result of Query:") ;
print "<table border=1 bordercolor=magenta bgcolor=cyan>" ;
print "<tr><th>Emp-id</th><th>Full Name</th><th>Designation</th>" ;
print "<th>Department</th><th>Date of Birth</th><th>Salary (\$)</th></tr>" ;

while (<OUTFILE>) {
     ($empid, $ename, $desig, $dept, $dtbirth, $salary) = split (/\|/) ;
      print "<tr><td>$empid</td><td>$ename</td><td>$desig</td>" ;
      print "<td>$dept</td><td>$dtbirth</td><td>$salary</td></tr>" ;
}
print "</table>" ;
&Htmlfooter ;
```

The table headers are printed with the `<tr>` and `<th>` tags. The program picks up each line of `OUTFILE`, splits it and then prints it as a table row with the `<tr>` and `<td>` tags.

Because of its powerful text manipulation capabilities, **perl** is the most widely used language for CGI programming on the Internet. However, CGI is a security threat on the Internet as a result of which the server administrator often disables CGI operation by individual users. In case you find this restriction on your system, contact the administrator.

Output of CGI program **emp_query.pl**