

More Perl



Perl and the OS

- ◆ Remember all those Perl built in functions shown earlier –
 - Access to file and directory information
 - Process management
 - Ports and Sockets
- ◆ Perl scripts exploiting these functions are often used to automate repetitive tasks for the "system's administrator".

Perl : files and directories



Information on files and directories ...

- ◆ File (and subdirectory) names in a directory
- ◆ Properties of an individual file

Inevitably some system dependent features – Unix files and Windows files do have different properties.

Reading a directory

- ◆ Directory handle
 - Obtained using `opendir`
 - Read via `readdir`
- ◆ Returns list of names of all entries in directory
 - Subdirectories
 - Links
 - Files, including hidden files whose names start with "."
 - References to current and parent directory (".", "..")

Basic properties of a file

- ◆ Perl has "file tests" similar to those that exist in shell.
- ◆ Used as
 - `<test operator> filename`
 - Mostly return true/false result

File tests

◆ File test operators

- -x is file (or directory) "executable"?
- -r is it readable?
- -w is it writeable?
- -d is it a directory?
- -l is it a link?
- -f just a plain old file maybe?
- -T is it "text"?
- -e does it even exist?
- -s size in bytes
- -M days (as real number) since last modified
- ...

Example: accessing files ...

- ◆ Input: fully qualified path name of directory of interest
- ◆ Output: a listing of directory contents,
 - Directories and links: *just identify by type*
 - Files: *highlight those that are executable; those that are "text" should have length given in report.*

Osutil1.pl

```
#!/share/bin/perl

while(1) {
    print "Enter pathname of directory : ";
    $directory = <STDIN>;
    ...
    #get list of names of contents of directory
    ...
    foreach $name (@names) {
        ...
    }
}
}
```

Osutil1.pl

```
while(1) {
    print "Enter pathname of directory : ";
    $directory = <STDIN>;
    chomp($directory);
    if($directory =~ /^Quit$/i) { last; }
    if(!opendir(DIRHANDLE, $directory) ) {
        print "Couldn't open that directory\n";
        next;
    }
    $directory =~ s#/####;
    @names = readdir DIRHANDLE;
```

Osutil1.pl

opendir(DIRHANDLE, \$directory);

- ◆ First argument is a Directory Handle (*like file handles, directory handles are usually given names consisting entirely of capital letters*)
- ◆ Second argument is pathname of directory
\$directory =~ s#/####;
- ◆ Simply tidying up for later. Some people enter directory names with a trailing /, some don't. This removes a trailing slash if one is there so that later formatting can be done more consistently.

Osutil1.pl

```
@names = readdir DIRHANDLE;

foreach $name (@names) {
    if($name =~ /\^\.\+$/i) { next; }
    $fullname = $directory . "/" . $name;
    ...
}
}
```

Osutil1.pl

```
if( -d $fullname) { print "Subdirectory: $fullname\n"; }
elsif( -l $fullname) { print "$fullname is a link\n"; }
else {
    print "$fullname\n";
    if( -x $fullname) { print "\texecutable\n"; }
    if( -T $fullname) {
        $size = -s $fullname;
        print "\tText with $size bytes\n";
    }
}
```

Of course, there is another way ...

- ◆ Actually, there are two other ways of getting lists of files in directories
- ◆ And you can use "stat" function on files to get lots and lots of extraneous information about a file.

globbing

- ◆ Use simple shell style pattern to specify entry names of interest
 - * everything
 - *.pl all perl scripts
 - [AB]*.cc cc programs whose names start with A or with B (note * means something quite different here than in regular expression)
- ◆ Invoke either by
 - < >
 - glob

globbing

- ◆ Differs from readdir in that get back fully qualified file names
- ◆ Shell is used to do pattern matching to select the files that you want (note * doesn't return files beginning with ".")

Osutil2.pl

```
#!/share/bin/perl
while(1) {
    print "Enter pathname of directory : ";
    $directory = <STDIN>;
    chomp($directory);
    if($directory =~ /^Quit$/i) { last; }
    unless(chdir( $directory) ) {
        print "Couldn't change to that directory\n";
        next;
    }
    ...
}
}
```

Osutil2.pl

```
while(1) {
    ...
    foreach (<* >) {
        if( -d ) { print "Subdirectory: $_\n"; }
        elsif( -l ) { print "$_ is a link\n"; }
        else {
            print "$_\n";
            if( -x ) { print "\texecutable\n"; }
            if( -T ) {
                $size = -s;
                print "\tText with $size bytes\n";
            }
        }
    }
}
```

Caution ...

- ◆ Code using implicit unnamed variable (\$_) can be cutely concise

```
(<*>)  
if(-d)  
$size = -s;
```

- ◆ But remember

- **Code is generally "WORM"y**
 - Write once
 - Read many

- ◆ Too many linguistic tricks will result in problems for readers

Other file operations

- ◆ Perl core has functions:

- rename
- unlink
- chmod
- mkdir
- rmdir

- ◆ Mostly take filename (file in current working directory or file with fully qualified pathname)

Perl : processes



system(), fork(), exec()

- ◆ Since Perl interpreter is a big C program, naturally have access to the C functions in `stdlib.h` and `unistd.h`

- **system**

- Argument is string with command(s) interpreted by `sh` (or passed directly to `execvp`)
- Subprocess shares `stdin`, `stdout`, `stderr` with perl program that launched it
- Perl process waits for subprocess to terminate
- On return `$?` holds information about termination
 - `$? >> 8` exit code

... fork,

- ◆ **fork()**

- The usual (almost)
 - Parent process (that performed `fork()`) gets process id of child
 - Child gets a zero value as its return value from `fork`
 - If attempt to `fork` fails (process table full or similar problem) then instead of negative value, get "undefined" returned

After a fork

- ◆ Child process goes and does its stuff

- ◆ Parent can continue, or can wait for child to terminate

- Usual problem with "zombie" processes if don't wait for child terminations
- Can play with signal mechanisms if this is a problem

... exec

- ◆ The usual
 - Replace code segment of current process with executable specified in exec call
 - Other arguments build up argv argument vector, and if desired the envp environment vector

Using other processes

- ◆ Fork – exec
 - Well, if you want
- ◆ System
 - Commonly used to run small subtasks
- ◆ Backticks
 - Like “system”, but a better way of collecting the output from a subprocess so that it is available in to the Perl program that launched the subtask

System and backticks

- ◆ system(“date”)
 - Invokes date operation in shell, which prints to stdout shared with Perl process
- ◆ \$date=`date` ;
\$str = “Today is \$date”;
 - Invokes date operation in shell, grabbing the output and in this case interpolating it into a string (Just an example; this isn't the way to do it for real as perl has its own functions to get at dates.) Note, you can't embed a backtick operation directly into a string.

Example: getting system data

- ◆ Your organization
 - Uses Sun's NIS+ system to store data like users and hosts
 - Has its machines named:
 - ◆ Machine.dept.company_URL
- ◆ You want a printout
 - Grouped by department
 - Lists machines and their IP addresses

Data is available from niscat

- ◆ niscat hosts.org_dir
- ◆ Returns list in format:
Canonical_host_name alias IP comments
- ◆ Example
red.accounting.ourorg.com red.accounting.ourorg.com 209.208.207.1
red.accounting.ourorg.com red 209.208.207.1
blue.accounting.ourorg.com blue 209.208.207.2
- ◆ Machine may appear several times with different aliases

Hosts.pl

- ◆ Program needs to grab output from niscat command
 - Split the line to get name, alias, ip
 - Examine the name for machine and department fields (things like “localhost” will also appear, and should be ignored)
 - Stores unique [dept, name, ip] combinations
- ◆ Prints report of sorted data

Need some data structures!

- ◆ Data structures and references not part of this introductory Perl component.
- ◆ perldoc perldsc
 - Examples of lists of lists, lists of hashes, hashes of list, and what is needed here a hash of hashes
- ◆ Hash 1
 - Indexed by department name
 - Value is Hash 2
- ◆ Hash 2
 - Indexed by machine name
 - Value is IP address

Hosts.pl

```
#!/share/bin/perl

system("echo $LOGNAME");
$date = `date`;
$str = "Today is $date";
print $str, "\n";

foreach ( `niscat hosts.org_dir` ) {
    ...
}

foreach $dept (keys %machines) {
    ...
}
```

Hosts.pl

- ◆ System call; executed while Perl waits
- ```
system("echo $LOGNAME");
```
- ◆ Using backticks to grab output ...
- ```
$date = `date`;

foreach ( `niscat hosts.org_dir` ) {
    ...
}
```

Hosts.pl

```
foreach ( `niscat hosts.org_dir` ) {
    ($name, $alias, $ip) = split //;
    if ($name =~ /^(\w+)\.(\w+)/) {
        $machine = $1;
        $dept = $2;
        $machines{$dept}{$machine} = $ip;
    }
}
```

Hosts.pl

```
foreach $dept (keys %machines) {
    print "$dept\n";
    foreach $machine
        (keys %{ $machines{$dept} }) {
        $ip = $machines{$dept}{$machine};
        printf "\t%-20s\t%16s\n", $machine, $ip;
    }
}
```

Example: processes etc



Examples

Change all occurrences of "cat" to "dog" (dogastrophe)

Use system call to run "date" piping output to lp

Find all words in the dictionary that contain all the vowels

Uhm, why can't we have REAL examples

Perl script for processing assignments ...

- ◆ This example meant to illustrate a more realistic use of Perl for a tedious system's administration task.
- ◆ "Unix shell" (or Python) scripts would be as good – the Perl is (in my opinion) a little easier to understand than a "sh" script.

The task ...

- ◆ Process assignments submitted for a lab class.
 - Assignment submission system creates a directory for each student who submitted work for assessment; this directory holds the file(s) submitted by the student. Student directories grouped in "lab class" directories.
 - Tutors need printouts for students in their labs:
 - Listing of code
 - Report on test compilation
 - Report on test run
 - Output to be clearly labeled for each student in lab.

Task ...

- ◆ Typical problems
 - Students submitting incorrect files
 - Students who submit code that results in 1000 lines of compiler error messages
 - Students submitting programs that handle I/O tasks incorrectly; example:
 - program is supposed to read from a file
 - Instead program "prompts for input" and waits (for ever?)
 - Programs that loop and generates 10000 lines of output

Perl script

- ◆ Part is general purpose, used for any assignment
 - Get tutor name, lab name (to identify printout)
 - Switch to correct lab directory
 - For each student subdirectory in lab directory, process submission
- ◆ Part is specific to requirements of an individual assignment
 - *This assignment requires a Java program to be compiled and then run reading input from file ...*
 - *This assignment has a C++ main.cc and a Graph(.h,.cc) files, to be compiled with options ... and run ...*

Specific example task ...

- ◆ Students writing Java
 - File A.java provided by lecturer, *possibly* modified by student
 - File B.java, written by student, contains main() for Java program
 - Other Java files (depending on student's design of Java program)
 - README.txt documentation file
 - All files to be submitted as a single "tarred and gzipped" file A1.tar.gz

Specific Example task ...

- ◆ Processing of one submission:
 - Unpack files
 - If student included an A.java that is unchanged from that supplied by lecturer, remove it.
 - List README.txt and .java files
 - Copy default A.class file
 - Compile the java code
 - ◆ Limit output to the last 100 lines of any compilation error report
 - If compiled successfully
 - ◆ Run the program, with a time limit of 5 seconds to catch those infinite loops

Specific Example task ...

- ◆ Processing of one submission:
 - If compiled successfully
 - ◆ Run the program, with a time limit of 5 seconds to catch those infinite loops
 - ◆ Program requires a single command line parameter, and no "stdin" input.
 - ◆ Send program output to file.
 - Print last 100 lines of output file resulting from test run.
 - Remove all files created; restore original tar gzip file
 - Make sure that all error messages (to stderr) are merged with stdout at all stages of processing.

Specific Example task ...

- ◆ General
 - Start printout with a banner page with tutor's name
 - List identifiers of all students who submitted in that lab
 - List student output (output and error output from steps just described) with identification information
 - Send to console a brief indication of progress as script runs together with identification of any students whose submitted files appear corrupt

Lablist.pl

- ◆ Main program
 - Input of tutor name and directory name
 - Change to that directory
 - ◆ Holds submissions for a lab class (one subdirectory per student)
 - ◆ "Control" subdirectory used by electronic submission system
 - ◆ Files: A.java, A.class
 - For each entry in directory if is a subdirectory with student submission then process that submission

Lablist.pl

- ◆ Main line
- ◆ Two subroutines
 - "valid"
 - ◆ identifies student subdirectories (must be a directory not a file, must not be called "control")
 - processsubmission
 - ◆ Performs processing steps for individual submission

Lablist.pl

- ◆ system() call used to launch subtasks
 - Unpack student files
 - Compare student's A.java with default; maybe delete file
 - List to java and other files
 - Compile student's B.java (and rest of java code)
 - Run student's code
- ◆ Output from script must merge correctly with output from system calls.

Lablist.pl

```
#!/share/bin/perl -w

sub processsubmission { ... }

sub valid { ... }

$tutor = $ARGV[0];
$dirname = $ARGV[1];
...
foreach (@valid) {
    ...
}
```

Lablist.pl: main line, command line arguments

```
$tutor = $ARGV[0];
$dirname = $ARGV[1];
if(! ((defined $dirname) && (defined $tutor))) {
    print STDERR <<MSG;
}
Tutor:
This script requires two command line arguments:
1) your name as a banner for printout
2) the directory with the submissions for your lab.
MSG
    exit 1;
}
```

Lablist.pl : change to lab directory, prepare for output, ...

```
...

if(!chdir($dirname)) {
    print STDERR "Couldn't cd to the directory $dirname\n";
    exit 1;
}
$| = 1;
system("banner $tutor");
print "\f";
```

More Perl magic

- `$| = 1;`
 - ◆ Switches current default output (STDOUT) to autoflush mode
 - Output buffers flushed after each write
 - ◆ Script will print header line, then invoke `system()`; want header line to appear before output from `system()`!
 - ◆ Recent Perl implementations would do a flush before an `exec` call (like `system`) but older interpreters don't.

Lablist.pl : check subdirectories

```
opendir(CWD, ".");
@entries = readdir(CWD);
closedir(CWD);

@entries = sort @entries;

print "Submissions for your lab received from:\n";
@valid = ();
foreach (@entries) {
    if(valid($_)) {
        print $_, "\n";
        push(@valid, $_);
    }
}
}
```

Lablist.pl

```
sub valid {
    $name = $_[0];
    if($name =~ /^\/\.\.+$/ ) { return 0; }
    if($name eq "control") { return 0; }
    if(-d $name) { return 1; }
    return 0;
}
```

Lablist.pl

```
foreach (@valid) {
    printf "\fStart of record for $_\n";
    print STDERR "$_\n";
    chdir($_) ||
        die "Failed to cd into student's subdirectory\n";
    processsubmission;
    chdir("../");
}
```

Lablist.pl: process submission

```
sub processsubmission {
    #Prepare files
    #Deal with A.java
    #Print required files
    #Try compile
    #Try run
    #Tidy remaining files
}
```

Lablist.pl : prepare files

```
sub processsubmission {
    system("gunzip A1.tar.gz");
    $errcode = $? >> 8;
    if($errcode) {
        print "Corrupted gzip file?\n";
        print STDERR "$_ has corrupt gzip file\n";
        return;
    }
    system("tar -xf A1.tar");
    if($errcode) {
        print "Corrupted tar file?\n";
        print STDERR "$_ has corrupt tar file\n";
        return;
    }
    ...
}
```

Lablist.pl: Deal with A.java

```
sub processsubmission {
    ...
    #Remove any .class files
    system("rm -f *.class");
    system("ls -l");
    #has student submitted an A.java
    if(-r "A.java") {
        # if same as that supplied, remove it
        system("diff A.java ../A.java 1>/dev/null 2>/dev/null");
        $code = $? >> 8;
        if($code == 0) {
            system("rm A.java");
        }
    }
    system("cp ../A.class .");
}
```

Lablist.pl : print files

```
sub processsubmission {
    ...
    system("cp ../A.class .");
    @files = <*.java,txt>;
    foreach $file (@files) {
        print "\f$file:\n";
        system("cat $file");
    }
}
```

Lablist.pl : compile Java

```
sub processsubmission {
    ...
    system("javac B.java 2>&1 > compile_errs");

    $compile_error = $? >> 8;
    if($compile_error) {
        system("tail -100 compile_errs");
        print "\fAssignment not run because of compilation errors\n";
    }
    else {
        ...
    }
}
```

Lablist.pl : Run Java

```
sub processsubmission {  
    ...  
    if($compile_error) { ... }  
    else {  
        $command = '  
    ulimit -t 5  
    java B 17 2>&1 < /dev/null > output  
    tail -100 output  
';  
        system($command);  
    }  
}
```

Lablist.pl : Tidy up

```
sub processsubmission {  
    ...  
    system("rm -f *.java *.class *.txt compile_errs output");  
    system("gzip A1.tar");  
}
```

Perl : network stuff



use Socket;

- ◆ Perl core does contain a number of the Unix standard network functions
 - socket(), gethostbyname(), ...
- ◆ Problem
 - Many of these functions require obscure constants (e.g. integers representing different protocols) and/or data structures
- ◆ Solution
 - The constants, structures, functions etc are defined in the `Socket` module.

Use a module ...

- ◆ More on modules shortly ...
- ◆ Using a module –
 - Like `#include` for C/C++
 - Like `import` for Java
- ◆ Program starts with “use” directives.
- ◆ Then can utilize functions declared in that module.

Network example 1: http client

- ◆ A minimal http client!
 - Sends a “GET” request for a specified file
 - Prints all the response that it receives
 - Command line arguments identify host and port number.

Client.pl

```
#!/share/bin/perl -w

use Socket;

# get arguments with host/port
# build "address" data structures
# open socket connection
# set I/O options on socket connection
# get user to enter file name
# send "Get" request
# loop printing all lines of response
```

Client.pl

```
if(@ARGV < 1) {
    print "Invoke with one or two arguments,
    hostname and optional port\n";
    exit(1);
}

$server = shift @ARGV;
$port = shift @ARGV || 80;
```

Client.pl

```
$iaddr = inet_aton($server)
    || die "no host: $server\n";
$paddr = sockaddr_in($port, $iaddr);

$proto = getprotobyname('tcp');
socket(SOCK, PF_INET, SOCK_STREAM, $proto)
    || die "socket: $!\n";
connect(SOCK, $paddr) || die "connect: $!\n";
```

Client.pl

```
connect(SOCK, $paddr) || die "connect: $!\n";

select(SOCK);
$|=1;
select(STDOUT);

print "File :";
$input = <STDIN>;
chomp($input);
```

Client.pl

```
print SOCK "GET $input HTTP/1.0\n\n";
while($line = <SOCK>) {
    print $line;
}

close (SOCK) || die "close: $!\n";
exit;
```

Network example 2: "Analyze" a web log

- ◆ A minimal analyzer
 - Read access log
 - Extract client IP addresses, forming list of unique IP addresses
 - Lookup hostnames corresponding to IP addresses
 - Strip machine name to get domain
 - Form list of unique domains identifying our clients
 - Print sorted list

Input data

◆ Access log with entries like:

```
210.84.124.193 - - [20/Jun/2001:00:30:50 +1000] "GET /subjects
HTTP/1.1" 301 253
```

```
203.132.226.245 - - [20/Jun/2001:00:36:22 +1000] "GET
/images/staff.gif HTTP/1.0" 304 -
```

◆ Take client address:

- 203.132.226.245
- P53-max4.wgg.ihug.com.au

◆ Machine not significant, so simplify to wgg.ihug.com.au

Web.pl

```
#!/share/bin/perl
```

```
use Socket;
```

```
#read file, saving unique client IPs
```

```
#loop using "get host by address" to try to get host name
```

```
# strip machine, and keep list of unique domains
```

```
#sort and print
```

Web.pl

```
while(<STDIN>) {
    @data = split;
    $callers{$data[0]} = 1;
}

foreach (keys %callers) {
    $addr = inet_aton($_);
    $name = gethostbyaddr($addr, AF_INET);
    if($name eq "") { next; }

    $pos = index($name, ".");
    $name = substr($name,$pos+1);
    $names{$name} = 1;
}
}
```

Web.pl

```
print "Sorting and reporting\n";
foreach (sort (keys %names)) {
    print "$_\n";
}
$val = scalar keys %names;
print "Our clients were from $val different domains\n";
```

Whose is bigger?



Whose is bigger? Java's or Perl's?

◆ "Class library" that is.

◆ You "sort of" know about Java's packages.

◆ Perl also makes a substantial effort.

- Not a "class library" – a collection of modules.
- www.cpan.org/modules
- There are hundreds

Modules ...

- ◆ Language extensions and documentation tools
- ◆ Experimental thread support
- ◆ Utilities (perl programs for file and directory manipulations)
- ◆ Networking: Socket wrappers, ICQ, Gnutella, LDAP, Japper, SMTP, SSL,
- ◆ Data types and algorithms
- ◆ Database interfaces
- ◆ User interfaces (including TCL/TK, Xwindows, ...)

And more modules

- ◆ File utilities
- ◆ Text manipulation
- ◆ Natural language helpers (spell check, word stemming, hyphenation, ...)
- ◆ XML and XSLT
- ◆ Encryption, digests, etc
- ◆ WWW, CGI, and HTTP
- ◆ MS-Windows stuff
- ◆ Junk, ...

Modules

- ◆ Modules have to be “installed” into your Perl environment
 - e.g. PPM tool
 - ◆ Download zip file with module you want (from www.cpan.org or www.activerperl.com)
 - ◆ Use ppm to install

Our interest in modules ...

- ◆ Databases
- ◆ WWW, CGI stuff

Databases

- ◆ Perl's approach to using database is very similar to that you learnt with Java.
 - High level API providing a uniform mechanism for submitting SQL queries
 - Low level, database specific drivers that accept requests via high level interface and invoke operations on the actual database

www.oreilly.com/catalog/perl/dbi/ch04.html

DBI and DBD

- ◆ DBI “Data base interface”
- ◆ DBD “Data base driver”
 - Many DBD variants, one (or more) for each make of database
 - ◆ DBD-Adabas, DBD-Excel (yes can treat Excel spreadsheet as a database), DBD-Ingres, DBD-Informix, DBD-Interbase, DBD-ODBC, DBD-Oracle, ...

DBD

- ◆ Fortunately, average Perl programmer does not have to bother much about the DBD level.
 - Find what drivers you have installed
 - Install additional drivers as needed.

What drivers do you have?

- ```
use DBI;
@drivers = DBI->available_drivers;
print "Available drivers : @drivers\n";
```
- ◆ On my Windows PC, I got things like "Proxy", "ADO", and "ODBC"
  - ◆ On Unix, I got things like "Proxy", and "Oracle"
  - ◆ Useful ones were ODBC and Oracle
    - I had to install the DBD-ODBC component on my PC, it doesn't come by default with PC-Perl.

## What data sources do you have?

- ◆ OK, you have a ODBC on your PC, or an Oracle driver on Unix.
  - ◆ What databases can you use these to access?
- ```
use DBI;  
print "Enter driver name : ";  
$drivename = <STDIN>;  
chomp($drivename);  
@sources = DBI->data_sources($drivename);  
print "sources: @sources\n";
```

Datasources

- ◆ On PC, I got response
 - DBI:ODBC:Visual FoxPro Database
 - DBI:ODBC:dBase Files
 -
 - DBI:ODBC:epalfinder
 - ◆ This is the meaningful one, it is the entry that I had created (Control Panel/ODBC Data sources) for a Microsoft Access database with my data table

Datasources

- ◆ On Unix I got
 - List of various Oracle databases on systems in uni
 - ◆ dbi:Oracle:CSCI.CS.UOW.EDU.AU
 - ◆ This was the important one – it is the Oracle system used for database assignments

Note: current Oracle for student assignments is

dbi:Oracle:CSCI

Datasources

- ◆ Typically, you know the name of the datasource that your program uses.
- ◆ It gets coded as a constant string in your program.
- ◆ You rarely need to use those DBI calls in actual programs.

Use of datasource

- ◆ Similar to your previous use of JDBC
- ◆ You connect to a datasource and get a "database handle" (Java's Connection object)
- ◆ You use your database handle to prepare an SQL statement for execution.
- ◆ You run your query, getting in return an "object" that you can use to get each row of a response (for select queries etc)

Connecting to a datasource

- ◆ Get "database handle"
 - Provide "data source" name string
 - If necessary, provide user-name and password
 - Preferably, also supply attributes (e.g. specify that want "autocommit" of operations)
- ◆ DBI->connect(...)
 - Returns a "scalar", the database handle (or fails)

```
$dbh = DBI->connect("DBI:ODBC:epalfinder");
```

```
$dbh = DBI->connect(
```

```
    "dbi:Oracle:CSCI.CS.UOW.EDU.AU","HSimpson", "Doh");
```

```
$dbh = DBI->connect("dbi:Oracle:CSCI.CS.UOW.EDU.AU",  
    "HSimpson", "Doh", { AutoCommit => 1 } );
```

Preparing statements

- ◆ Simple examples:
 - Get everything in Table1

```
$sth = $dbh->prepare(  
    "SELECT * FROM Table1");  
...  
$sth->execute;
```
 - Get subset of stuff from Table1

```
$sth = $dbh->prepare(  
    "SELECT * FROM Table1 WHERE GRADE > 3");  
...  
$sth->execute;
```

More statements:

- ◆ Get subset of stuff from Table1, depending on user input

```
print "Enter 'Grade' cutoff : ";  
$grade = <STDIN>;  
chomp($grade);  
$sth = $dbh->prepare(  
    "SELECT * FROM Table1 WHERE GRADE > $grade");  
...  
$sth->execute;
```

Still more statements

- ◆ Advice is to "prepare statements" with placeholders for arguments, rather than build string for statement each time.

```
$sth = $dbh->prepare(  
    "SELECT * FROM Table1 WHERE GRADE > ?");  
...  
...  
print "Enter 'Grade' cutoff : ";  
$grade = <STDIN>;  
chomp($grade);  
$sth->execute($grade);
```

 - Partly, this is for efficiency; but there are other reasons ...

Still more statements ...

- ```
$sth = $dbh->prepare(
 "SELECT GRADE FROM Table1 WHERE NAME='Smith'");
```
- ◆ As usual, need single quotes around string data in an SQL statement.

```
print "Enter name : ";
$name = <STDIN>;
chomp($name);
$sth = $dbh->prepare(
 "SELECT * FROM Table1 WHERE name= '$name'");
```

    - Oops, suppose the name is O'Brien!
    - Don't have problems if use alternative with "?" placeholders (which is another reason for using them)

## Statements to insert, update, and delete data ...

```
$sth = $dbh->prepare(
 "INSERT INTO TABLE1 VALUES (?, ?, ?)");
$sth = $dbh->prepare(
 "UPDATE TABLE SET GRADE= ? WHERE NAME = ?");
$sth = $dbh->prepare(
 "DELETE FROM TABLE1 WHERE NAME='Smith'");
```

## Statement preparation

- ◆ If SQL is faulty, then prepare will fail, get "undef" as result.

```
$sth = $dbh->prepare(...);
unless defined $sth die "SQL problem?";
```

## Executing statements

### ◆ Inserts, Updates, Deletes

- Succeed or fail
- Can check  
`$sth->execute($data) || die "delete failed";`
- Can get more information from `DBI::errstr`.
  - ... die "Delete failed because `DBI::errstr`";

## Executing statements

### ◆ Getting selected data

- Fetch successive rows of a result table  
`@row = $sth->fetchrow_array;`
- Returns undef when have processed all data.
- ◆ Typical code:  

```
$sth = $dbh->prepare("select ... = ?");
...
$sth->execute($data1, $data2);
while(@row = $sth->fetchrow_array) {
 ...
}
```

## Close your database handle

- ◆ It will be closed when your program terminates.
- ◆ But best if explicitly close connection as soon as possible
- ◆ `$dbh->close;`

er ... shouldn't that be disconnect

## What are the -> things?

- ◆ Objects in Perl of course.
  - Invoke method of object
- ◆ Mostly beyond the scope of our introductory treatment
- ◆ Easy to use for simple standard database operations.
  
- ◆ So, don't worry about them!

## My usual DB test

### ◆ Table "Teams"

```
create table Teams (
 Team1 varchar(32),
 Team2 varchar(32),
 Score1 number,
 Score2 number
);
insert into teams values ('norths',
 'souths', 1, 0);
```

## Connect to db and run

```
use DBI;
$dbh = DBI->connect("dbi:Oracle:CSCI",
 "homer","doh") || die "It didn't connect";

print "Connected\n";
$searchhandle = $dbh->prepare("select * from teams");

$searchhandle->execute || die "Select request failed because
 $DBI::errstr";

while(@row=$searchhandle->fetchrow_array) {
 foreach $item (@row) { print "$item "; }
 print "\n";
}
```

## It should run

```
$ perl testdb.pl
Connected
norths souths 2 0
Eastss Westss 3 2
City Country 2 2
River Forest 0 0
$
```

## Database example



## e-pal example

- ◆ Simple service for finding email- pen- friends with similar interests
- ◆ Database
  - Insert a record that describes you and your interests
  - Search against existing records to find someone with common interests

## e-pal

- ◆ Database records
  - Your email identifier (unchecked character string of 32 characters)
  - Your "gender" (male, female, "eperson" – if you don't want to specify gender)
  - "Gender" of desired correspondents (male, female, eperson, any)
  - Interest1, interest2, interest3, interest4, interest5
    - Integer values
    - Index numbers of interests in a predefined array of approximately 100 topics – Aerobics, Archery, Ballet, Cats, Cars, Drinks, ..., Yoga, Zen

## e-pal

- ◆ Unique key email
- ◆ To add data
  - Submit email, gender, wanted gender, five interests picked from list
- ◆ To search
  - Submit your gender, wanted gender, five interests picked from list
    - ◆ Match must satisfy you wanted gender
    - ◆ You must satisfy match's desired gender
    - ◆ You must have some common interests.

## e-pal

- ◆ Match criteria are difficult to specify in SQL
  - "any" matches "male", "female", or "eperson"
  - Interests represented as 5 integers in random order
- ◆ So don't really exploit database capabilities!
  - Retrieve complete records
  - Examine records procedurally in program code

## e-pal : program structure

- ◆ Initialization:
  - Create lists and hashes with predefined interests
  - Open database connection
  - Prepare a statement that will be used for insertion
- ◆ DoAdd
  - Prompt for email, gender, desired gender, list of interests (each part handled by subroutine)
  - Run parameterized insertion request
- ◆ DoSearch
  - Prompt for gender, desired gender, list of interests
  - Run search request

## e-pal : program structure

- ◆ Handle search results
  - For each retrieved record, match against specified requirements; print emails of those who match
- ◆ Main
  - Initialize
  - While not quitted
    - ◆ Ask user for add or search request
    - ◆ DoAdd or DoSearch

## The epal table

```
CREATE TABLE EPAL
(email VARCHAR(32) NOT NULL,
 type varchar(8) NOT NULL,
 want varchar(8) NOT NULL,
 interest1 number(4),
 interest2 number(4),
 interest3 number(4),
 interest4 number(4),
 interest5 number(4),
 CONSTRAINT id_pkey PRIMARY KEY(email),
 CONSTRAINT type_check CHECK (type in ('MALE',
 'FEMALE', 'EPERSON')),
 CONSTRAINT want_check CHECK (want in ('MALE',
 'FEMALE', 'EPERSON', 'ANY'))
);
```

## Epal code

- ◆ General:
  - use DBI
    - ◆ "include" function definitions for database
  - use strict
    - ◆ Really an interpreter switch.
    - ◆ It checks that all variables are properly declared and scoped (*this is a slightly larger program, several subroutines, tendency to use similar names "interest", "want" etc in different subroutines – don't want any confusions!*)

## Epal code

### ◆ Structure

- "uses"
- Declaration of filescope "my" variables, some initialized at declaration
- Functions
- main

## epal

```
#!/share/bin/perl -w
use strict;
use DBI;

my $data_source = "dbi:Oracle:CSCI";
...
sub initialize { ... }
sub owntype { ... }
sub getinterests { ... }
sub wanttype { ... }
sub doSearch { ... }
sub doAdd { ... }
initialize;
while(1) { ... }
```

## Epal filescope "my" variables

```
my $data_source = "dbi:Oracle:CSCI";
my $dbh;
my $searchHandle;
my $insertHandle;
my %interesttable;
my @interestlist = ("Abseiling", "Aerobics", ...
...
"WebDesign", "WineTasting", "Yoga", "Zen"
);
```

## Epal : "main()"

```
initialize;

while(1) {
 my $cmd;
 print "Enter command (add,search, list (interests), quit): ";
 $cmd = <STDIN>;
 if($cmd =~ /quit/i) { last; }
 elsif($cmd =~ /search/i) { doSearch; }
 elsif($cmd =~ /add/i) { doAdd; }
 elsif($cmd =~ /list/i) {
 print "Interest list @interestlist\n";
 }
 else { print "Command not recognized\n"; }
}
```

## Epal : initialize

```
sub initialize {
 my ($id, $interest);
 $id = 0;
 foreach $interest (@interestlist) {
 $interesttable{$interest} = $id; $id++;
 }
 $dbh = DBI->connect($data_source,
 "HSimpson", "Doh", { AutoCommit => 1})
 || die "Couldn't connect to db\n";
 $searchHandle = $dbh->prepare(
 "SELECT * FROM epal");
 $insertHandle = $dbh->prepare(
 "INSERT INTO epal VALUES (?, ?, ?, ?, ?, ?, ?)");
}
```

## Epal : doAdd

```
sub doAdd {
 my ($you, $desire, $email, @interests);
 $you = owntype;
 $desire = wanttype;
 @interests = getinterests;

 print "Your email address : ";
 $email = <STDIN>;
 chomp($email);
 $insertHandle->execute(
 $email, $you, $desire, @interests)
 || die "Failed to insert record because $DBI::errstr!";
}
```

## Epal : doSearch : 1 : SQL request

```
sub doSearch {
 my ($you, $desire, @interests, @row);
 my ($theiremail, $theirtype, $theirdesire, @theirinterest);
 my $score;
 $you = owntype;
 $desire = wanttype;
 @interests = getinterests;

 $searchHandle->execute
 || die "Select request failed because $DBI::errstr";
 ...
}
```

## Epal : doSearch : 2 : check results

```
sub doSearch {
 ...
 $score = 0;
 while(@row = $searchHandle->fetchrow_array) {
 ($theiremail, $theirtype, $theirdesire, @theirinterest)
 = @row;

 ...
 print "Contact mail $theiremail:\tCommon interests: ";
 foreach (@common) {
 print "\t$interestlist[$_]";
 }
 print "\n";
 }
 unless ($score > 0) { print "Sorry, we currently don't have any
 contacts for you!\n"; }
}
```

## Epal : doSearch : 3 : matching

```
while(@row = $searchHandle->fetchrow_array) {
 ($theiremail, $theirtype, $theirdesire, @theirinterest)
 = @row;
 next unless (
 ($desire eq "ANY") || ($desire eq $theirtype));
 next unless (
 ($theirdesire eq "ANY") || ($theirdesire eq $you));
 my @common = (); my $interest;
 foreach $interest (@interests) {
 push(@common, $interest)
 if (grep {$interest == $_} @theirinterest);
 }
 next if ((scalar @common) == 0);
 $score++;
 print "Contact mail $theiremail:\tCommon interests: ";
```

## Epal: minor subroutines

```
sub owntype {
 my $type;
 my @allowed;
 while(1) {
 print "Your type: ";
 $type = <STDIN>;
 chomp $type;
 @allowed = qw(MALE FEMALE EPERSON);
 if(grep {/$type/i} @allowed) { last; }
 print "Unrecognized type; allowed values are @allowed\n";
 }
 return uc $type;
}
```

## Epal: minor subroutines

```
sub getinterests {
 my @list; my @nlist; my $temp;
 while(1) {
 print "Enter five personal interests : ";
 $temp = <STDIN>;
 @list = split /\s/, $temp;
 my $length;
 $length = @list;
 if($length != 5) { print "5 interests required\n"; next; }
 $temp = 1;
 ...
 if($temp) { last; }
 }
 return @nlist;
}
```

## Epal: minor subroutines

```
sub getinterests {
 ...
 while(1) {
 ...
 $temp = 1;
 my $item;
 foreach $item (@list) {
 unless (exists $interesttable{$item}) {
 print "Interest $item not recognized\n";
 $temp = 0;
 next;
 }
 push(@nlist, $interesttable{$item});
 }
 ...
 }
}
```

## Perl : CGI



## Why Perl for CGI?

- ◆ Most CGI work requires text matching and text generation – Perl is good at this.
- ◆ Most web sites include at least some interaction with databases – DBI module makes this a lot easier than embedded SQL in C/C++.
- ◆ So, inherently, Perl is a fairly good match for CGI requirements.

## Roll your own

- ◆ Easy isn't it.
- ◆ Data from a form will be available as x-www-urlencoded name=value strings
  - From <STDIN> if form does POST
  - From environment variable QUERY\_STRING if form does GET
- ◆ So
  - Generate standard header
  - Examine submitted data
  - Generate HTML for response

## The soccer league

- ◆ How about a very simple CGI program that just dumps the soccer results table as a HTML page  
<http://localhost:8080/soccer1.cgi> *(or whatever)*
- ◆ Get request (as always if you enter URL in browser)
- ◆ No inputs
- ◆ Just connect to database and copy data into HTML table

## A problem

- ◆ Environment variables defining Oracle property are usually not going to be defined for CGI program started by apache
- ◆ Have to set them in program

```
#!/usr/bin/perl
use DBI;
Standard header
print "Content-type: text/html\n\n";
Set up Oracle stuff
$ENV{"ORACLE_HOME"} = "/usr/local/instantclient_10_2";
...
Do work
$data_source = "dbi:Oracle:CSCI";
$dbh = DBI->connect($data_source, "homer", "doh", {
 AutoCommit => 1}) ||
 die "Sorry, the database is not currently accessible";
$teamsh = $dbh->prepare("SELECT * FROM Teams");

print <<HERE;
HTML markup for page
HERE
$teamsh->execute || die "Couldn't access team data";
while(@row=$teamsh->fetchrow_array) { ... }
$dbh->disconnect;
...
```

## Environment


```
$ENV{"ORACLE_HOME"} =
 "/usr/local/instantclient_10_2";
$ENV{"LD_LIBRARY_PATH"} =
 "/usr/local/instantclient_10_2";
$ENV{"TNS_ADMIN"} =
 "/usr/local/instantclient_10_2";
$ENV{"ORACLE_SID"} = "csci";
$ENV{"TWO_TASK"} = "csci";
$data_source = "dbi:Oracle:CSCI";
```

## Environment

- ◆ That was correct for setup on my Ubuntu machine;
- ◆ Should be identical (or very similar) on lab machines
  - (Any changes explained in lab)

```
$ENV{"ORACLE_HOME"} = "/usr/local/instantclient_10_2";
...
$ENV{"TWO_TASK"} = "csci";
$data_source = "dbi:Oracle:CSCI";
$dbh = DBI->connect($data_source, "homer", "doh", {
 AutoCommit => 1}) ||
 die "Sorry, the database is not currently accessible";
$teamsh = $dbh->prepare("SELECT * FROM Teams");

print <<HERE;
<html><head><title>Soccer league</title></head>
 <body>
 <h1>Soccer results</h1>
 <table border='1' align='center'>
 <tr><th>Team1</th><th>Team2</th><th>Score1</th>
 <th>Score2</th></tr>
 </table>
 </body>
HERE
```



The screenshot shows a web browser window with the title 'Soccer league'. The page content is as follows:

Team1	Team2	Score1	Score2
norths	souths	2	0
Easts	Wests	3	2
City	Country	2	2
River	Forest	0	0

## But usually you want some input ...

- ◆ All CGI programs are the same
  - Determine where input located (QUERY\_STRING or stdin)
  - Read string
  - Split out name=value pairs
  - Process values

## Form1 example:

```
What are you?
◇ Boy ◇ Girl ◆ Unsure

```

## Form1.html

```
<html><head><title>Test page</title></head>
<body>
<form method=post
 action="http://www.dostuff.com/cgi-bin/form1.cgi">
What are you?

<input type=radio name=sex value=Boy>Boy
<input type=radio name=sex value=Girl>Girl
<input type=radio name=sex value=Unsure checked>Unsure

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

## Processing input

- ◆ Form specified "post", so read from standard input
- ◆ Expect a single name=value combination (sex=Boy, sex=Girl, sex=Unsure)
- ◆ Have to process these three legal inputs, and also allow for illegal input that does not match any of the expected forms.

## Processing

- ◆ Generate the header first!
  - MUST START BY RESPONDING  
Content-type: text/html\n\n
- ◆ Read input, try to split out value field
- ◆ Four way if ... elsif ... elsif ... else ... test.
  - Each branch generates a different HTML response page

## Form1.pl (form1.cgi)

```
#!/share/bin/perl

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

$person= <STDIN>;
chomp($person);
$type = "Hacker";
if($person =~ /sex=(\w*)&*/) {
 $type = $1;
}
...
```

## Form1.pl (form1.cgi)

```
if($type eq "Boy") {
 print <<BOYS;
 <html><head><title>Boys</title></head>
 <body bgcolor=#0303cc>
 <h1>Boys</h1>
 Frogs and snails and puppy dogs tails,

 That's what little boys are made of.
 </body></html>
 BOYS
}
elsif($type eq "Girl") {
 ...
```

## Form1.pl (form1.cgi)

```
elsif($type eq "Girl") {
 print <<GIRLS;
 <html><head><title>Girls</title></head>
 <body bgcolor=pink>
 <h1>Girls</h1>
 Sugar and spice and all things nice

 That's what little girls are made of.
 </body></html>
 GIRLS
}
```

## Form1.pl (form1.cgi)

```

elsif($type eq "Unsure") {
 print <<OTHERS;
 <html><head><title>Others</title></head>
 <body>
 <h1>Others</h1>
 You should visit the student counseling services.
 </body></html>
 OTHERS
}
else {

```

## Form1.pl (form1.cgi)

```

else {
 print <<HACKER;
 <html><head><title>Evil</title></head>
 <body background=0xff0b0a>
 <h1>Evil hacker!</h1>
 I guess you want something like this
 HACKER
 $stuff = `niscat passwd.org_dir`;
 print $stuff;
 print "</body></html>";
}

```

## Roll your own CGI stuff?

- ◆ OK, that program generated HTML code correctly in response to a single input.
- ◆ But suppose:
  - 'get' rather than post
    - ◆ Index into %ENV hash using QUERY\_STRING
  - Multiple name=value pairs
    - ◆ Split at &
    - ◆ Split each pair at =
  - X-www-urlencoded data
    - ◆ Plus to space
    - ◆ \$xx to character

## OK, try a part with E-pal CGI

- ◆ The same E-pal application as just used to illustrate DBI module
- ◆ Now
  - HTML form
    - ◆ Join or Search option
    - ◆ Radio button clusters for preferences
    - ◆ Text input for email
    - ◆ Multi choice selection list for interests

What do you want to do?

Join e-pal     Search e-pal

What are you?

Male     Female     EPerson

Who do you want to contact?

Male     Female     EPerson     Any

Your email

Pick exactly 5 interests

- Abseiling
- Archery
- Astronomy
- Basketball

Submit

Your email and interest list may be used for targeted advertising by the sponsors of this site.

```

<html><head><title>EPals R' Us</title></head>
<body><h1>Finding an email-friend with common interests</h1>
<form method=get action="http://www.stuff.com/form2.cgi">
What do you want to do:

<input type=radio name=act value=add>
Add your details to e-pal database

<input type=radio name=act value=search checked>
Search the database for contacts
<hr>You are

<input type=radio name=self value=Male>Male
<input type=radio name=self value=Female>Female
<input type=radio name=self value=EPerson checked>EPerson

<hr>You want to contact

<input type=radio name=other value=Male>Male
<input type=radio name=other value=Female>Female
<input type=radio name=other value=EPerson>EPerson
<input type=radio name=other value=Any checked>Any

<hr>Your email address :<input type=text size=20 name=email>

<hr>Pick EXACTLY FIVE (5) interests from the following list:

<select name=interests size=8 multiple>
<option>Abseiling... <option>WineTasting<option>Yoga<option>Zen
</select>
<input type=submit>
<hr></form></body></html>

```

## Handling the data: 1

- ◆ Get the QUERY\_STRING
- ◆ Break (at &) into name/value pairs
- ◆ Break name/value pairs into name and value
- ◆ Plus to space on value
- ◆ %xx to character on value
- ◆ Simply echo what we got.

## Checking Form data

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

$stuff = $ENV{"QUERY_STRING"};
print "<html><head><title>test</title></head><body>\n";

@splitstuff= split /&/ , $stuff;
print "";
foreach (@splitstuff) {
 print "";
 ...
 print "$name\t:$value\n";
}
print "";
print "</body></html>";
```

## Checking Form data

```
foreach (@splitstuff) {
 print "";
 ($name, $value) = split /=/;
 $value =~ s/\+/ /g;
 while($value =~ /[0-9A-Fa-f]{2}/) {
 $old = "%$1";
 $chrval = hex $chrval;
 $symbol = chr $chrval;
 $value =~ s/$old/$symbol/;
 }
 print "$name\t:$value\n";
}
```

## Testing CGI Perl scripts

- ◆ Before you set up the web server:
  - Test that it compiles!
  - Test run it
    - ◆ In shell define a value for QUERY\_STRING  
QUERY\_STRING="self=MALE&other=FEMALE"  
export QUERY\_STRING
    - ◆ Now run script, it will use QUERY\_STRING that you just defined

## Handling the data : 2

- ◆ Real data handling requires database operations.
  - User name and password can be defined in program.
- ◆ Problem:
  - Database system will also need lots of environment variables – usually set as you login for terminal session
  - Environment variables not set automatically for a cgi script that is run as "nobody" or "www"

## Setting environment variables

- ◆ Perl stores environment variables in the hash ENV
- ◆ Database specific stuff must be added before the DBI request to get a database handle.

## E-pal as CGI program : 1

### ◆ Main line

- Output http header
- Call initialize subroutine
  - ◆ Create lists, hash tables with interests (used to check submitted data)
  - ◆ Modify environment
  - ◆ Open database connection
  - ◆ Create statement handles
- ...

## E-pal as CGI program : 2

### ◆ Main line

- ...
- Get QUERY\_STRING from %ENV
- Loop extracting and decoding name/value pairs
  - ◆ If name is 'act' : record action value
  - ◆ If name is 'self' : record requestor value
  - ◆ If name is 'other' : record desired value
  - ◆ If name is 'email' : record email value
  - ◆ If name is 'interests' : add interest value to a list

## E-pal as CGI program : 3

### ◆ Main line

- ...
- Check that action, own type, other type and interest list have defined values; if not, use subroutine to print an HTML error page.
- Check that if email is defined if action is add.
- Check that values for own type and wanted type are in legal sets (MALE, FEMALE, EPERSON) and (MALE, FEMALE, EPERSON, ANY)
- Check that five interests from the standard list were supplied, and convert from names to numeric values for entry in database

## E-pal as CGI program : 4

### ◆ Main line

- ...
- If action is add
  - ◆ Execute SQL insert statement with data values from form
  - ◆ Generate a page welcoming latest e-pal
- If action is search
  - ◆ Retrieve all records, checking each for suitability
  - ◆ Generate HTML text with details of matches

## Form2.cgi

```
#!/share/bin/perl
use strict;
use DBI;

sub initialize { ... }
sub badInterests { ... }
sub checkinterests { ... }
sub checktype { ... }
sub checkwant { ... }
sub badSubmission { ... }
sub badEmail { ... }
sub databaseFailure { ... }
main program
...
```

## Form2.cgi : functions generating HTML error pages ...

```
sub badInterests {
 print <<BAD;
 <html><head><title>Please complete interest
 list</title></head>
 <body>

 Sorry but we are unable to handle your request. You must pick
 exactly 5 entries from the list of possible interests.

 </body></html>
 BAD
}
```

## Form2.cgi: data checks

```
sub checktype {
 my $type= $_[0];
 my @allowed;
 @allowed = qw(MALE FEMALE EPERSON);

 if(grep { /$type/ } @allowed) { return 1; }
 return 0;
}
```

## Form2.cgi : data checks

```
sub checkinterests {
 my @list = @_;
 my (@nlist,$temp,$length);
 $length = scalar @list;
 if($length != 5) { badInterests; exit; }
 $temp = 1;
 my $item;
 foreach $item (@list) {
 unless (exists $interesttable{$item}) {
 badInterests; exit;
 }
 push(@nlist, $interesttable{$item});
 }
 return @nlist;
}
```

## Form2.cgi : globals etc

```
my ($stuff, @splitstuff, $name, $value);
my ($action, $enteredemail, $enteredtype, $enteredemail,
 @enteredinterest);

my $data_source = "dbi:Oracle:CSCI.CS.UOW.EDU.AU";
my $dbh;
my $searchHandle;
my $insertHandle;

my %interesttable;
my @interestlist = ("Abseiling", "Aerobics", ..., "Zen");
```

## Form2.cgi : initialization

```
sub initialize {
 my ($id, $interest);
 $id = 0;
 foreach $interest (@interestlist) {
 $interesttable{$interest} = $id;
 $id++;
 }
 ...
}
```

## Form2.cgi : initialization

```
sub initialize {
 $ENV{"ORACLE_HOME"} =
 "/usr/local/instantclient_10_2";
 $ENV{"LD_LIBRARY_PATH"} =
 "/usr/local/instantclient_10_2";
 $ENV{"TNS_ADMIN"} =
 "/usr/local/instantclient_10_2";
 $ENV{"ORACLE_SID"} = "csci";
 $ENV{"TWO_TASK"} = "csci"; ...
}
```

## Form2.cgi : initialization

```
sub initialize {
 ...
 $dbh = DBI->connect($data_source,
 "HSimpson", "Duh", { AutoCommit => 1}) ||
 die "Couldn't connect to db\n";
 $searchHandle = $dbh->prepare(
 "SELECT * FROM epal");;
 $insertHandle = $dbh->prepare(
 "INSERT INTO epal VALUES (?, ?, ?, ?, ?, ?, ?)");;
}
```

## Form2.cgi : main 1

```
print "Content-type: text/html\n\n";
initialize;
$stuff = $ENV{"QUERY_STRING"};
@splitstuff = split /&/, $stuff;
foreach (@splitstuff) {
 ($name, $value) = split /=/;
 $value =~ s/\+//g;
 while($value =~ /[0-9A-Fa-f]{2}/) { ... }
 if($name eq "act") { $action = $value; }
 elsif($name eq "self") { $enteredtype = $value; }
 ...
 elsif($name eq "interests") { push(@enteredinterest, $value); }
}
else { #some hacker out there. ignore whatever trash they put in }
}
```

## Form2.cgi : main 2

```
unless((defined $action) && (defined $entereddesire) &&
(defined $enteredtype) &&
(scalar @enteredinterest > 0)) {
 badSubmission; exit; }
if($enteredemail eq "") { undef $enteredemail; }
if(($action eq "add") && !(defined $enteredemail)) {
 badEmail; exit; }

badSubmission unless (checkwant($entereddesire) &&
checktype($enteredtype));

my @numericlist;
@numericlist = checkinterests(@enteredinterest);
```

## Form2.cgi : main 3

```
if($action eq "add") {
 $insertHandle->execute($enteredemail, $enteredtype,
 $entereddesire, @numericlist) ||
 databaseFailure;
 print <<JOIN;
<html><head><title>Thanks for registering with e-pal</title></head>
<body>
<h1>You are registered</h1>
We hope you find lots of new friends via e-pal.
</body></html>
JOIN
}
```

## Form2.cgi : main 4

```
elsif($action eq "search") {
 print "<html><head><title>E-pals for you</title></head><body>";
 my ($score, @row);
 my ($theiremail, $theirtype, $theirdesire, @theirinterest);

 $searchHandle->execute ||
 die "Select request failed because $DBI::errstr";
 $score = 0;
 while(@row = $searchHandle->fetchrow_array) {
 ...
 }
 unless ($score > 0) { print "Sorry, we currently don't have any
contacts for you\n"; }
}
else { badSubmission; }
exit;
```

## Just suppose it did die ...

- ◆ Example code just shown had a few "die ..." clauses, mostly relating to database problems.
- ◆ What would Web client see?
  - Probably a message like "document contained no data", or maybe an incomplete page
- ◆ Where would the programmer find the error message?
  - In the web server's log file --- tail www-errors

## Roll your own CGI Perl scripts?

- ◆ Not hard!
- ◆ Code to find parameter values tends to be a little clumsy (loop through all name value pairs, checking the names)
- ◆ Most HTML code can be output in blocks using "here strings"
  - But do still get fragments of HTML buried in code
- ◆ Pretty page formats (e.g. a table for the email addresses and common interests) would be straightforward but tiresome code.

## Perl : CGI module(s)



## Perl and CGI

### ◆ *Everybody is doing it*

### ◆ *So*

- Lots of little utility components get created
  - Get all name/value pairs into a hash and provide a function that retrieves particular one
  - Provide "helper functions" for formatting tables, forms, etc
  - ...
- Lots of utility components submitted to CPAN
- Some become standardized Perl modules

## CGI modules : dual interface

- ◆ CGI module (and several others) support both "object style" and "function style" interfaces
  - Object style
    - Create me an X object
    - X do this
    - But usually a singleton, so it isn't really that object based (just a group of functions and a little data)

## CGI module : arguments

- ◆ Syntax of calls to methods (functions) is quite varied (you will see totally dissimilar code in different sources)
  - Quite often, a function will have a number of optional parameters
  - Basically, these are passed as a hash initialized from some literal data:
    - ( -bgcolor => 'pink', -font => 'serif', ... )
    - ? -bgcolor

## CGI modules

- ◆ Selective imports
  - These modules declare large numbers of functions
  - Can selectively import subsets
    - Specific functions
    - Predefined groups of functions
- ◆ So get several variations in "use CGI" clause.

## CGI

- ◆ For more information
  - Perl documentation of course
  - (though I haven't found anywhere that listed all the optional arguments for some of the function calls!)

## Use param(), h1(), header(), ...

```
#!/share/bin/perl

use CGI qw/:standard/;

print header(-type => 'text/html');

$type = param("sex");

if($type eq "Boy") { ... }
elsif($type eq "Girl") { ... }
elsif($type eq "Unsure") { ... }
else { ... }
```

## Use param(), h1(), header(), ...

```
if($type eq "Boy") {
 print start_html(
 -title=>'Boys, girls, others, and evil ones',
 -meta=>{'keywords'=>'Nursery rhymes' },
 -BGCOLOR=>'blue');
 print h1("Boys"), p, "Frogs and snails and puppy dogs tails," , p,
 "That's what little boys are made of." , p;
 print end_html;
}
```

## Read the documentation ...

- ◆ Creating a table ...
- ◆ Placing http links in documents ...
- ◆ Making up a form with input fields of differing types ...

## Perl : Tainted perls



## CGI : files and processes

- ◆ As discussed earlier, Perl can launch processes and perform lots of file manipulations.
- ◆ So
  - In principle, a Perl CGI program can launch processes and perform lots of file manipulations – as requested by the hacker user out in cyberspace.

## *Tainted data : something nasty picked up via the Internet*

- ◆ If you are going to run commands that depend on user-supplied data, then you are risking hacker attacks.
- ◆ Perl has a scheme for marking data as "sus"
  - Switch -T on #!/usr/bin/perl line
  - Subsequently, all input data (and data derived from input data) are "tainted" and cannot be used with calls like system(), backticks, open etc

## Taints

- ◆ Read the documentation
  - perlsec

## Perl : finish



## Perl

- ◆ Convenient for many tasks
  - Text manipulations
  - System administration scripts
  - Database access
  - Some web stuff
- ◆ Use it.

## I liked Perl, others ...

- ◆ Perl is like vice grips. You can do anything with it, and it's the wrong tool for every job.
- ◆ Perl: the only language that looks the same before and after RSA encryption.
- ◆ Perl is worse than Python because people wanted it worse.
- ◆ Perl [is] a language. A way of life. No cure is known.
- ◆ Perl is another example of filling a tiny, short-term need, and then being a real problem in the longer term.
- ◆ Perl is the crystal meth of programming: it's so incredibly useful when you need to do a large amount of work in a small amount of time that you tend to overlook the fact that it's basically precipitating the implosion of your vital organs.
- ◆ PHP is a minor evil perpetrated and created by incompetent amateurs, whereas Perl is a great and insidious evil, perpetrated by skilled but perverted professionals.
- ◆ I would actively encourage my competition to use Perl.

## Leave final word to Larry Wall

- ◆ Most of you are familiar with the virtues of a programmer. There are three, of course: laziness, impatience, and hubris.