

PERL FOR ORACLE DEVELOPERS

Roger Gough, Computer Resource Team, Inc.

What is Perl?

Open source aficionados always look to see whether someone else has previously solved a problem or answered a question, so I will too. Here's an answer from the FAQ: "Perl is a high-level programming language with an eclectic heritage written by Larry Wall and a cast of thousands. It derives from the ubiquitous C programming language and to a lesser extent from sed, awk, the Unix shell, and at least a dozen other tools and languages. Perl's process, file, and text manipulation facilities make it particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, database access, graphical programming, networking, and world wide web programming. These strengths make it especially popular with system administrators and CGI script authors, but mathematicians, geneticists, journalists, and even managers also use Perl. Maybe you should, too."

I like that definition. I also like what Larry has said: "In a nutshell, Perl is designed to make the easy jobs easy, without making the hard jobs impossible." And I suppose that's what makes Perl so appealing to me. Not only does it rarely get in the way and present problems, as so many tools do, but its vocabulary is endlessly rich. The core language is especially powerful, particularly with regard to regular expressions, string manipulation, operating system access, and list processing. It is ideal for cleaning up and restructuring data in preparation for migrating to Oracle.

And to supplement Perl's traditional virtues, version 5 provides the following additional features:

- Compilability
- Regular expression enhancements
- Arbitrarily nested data structures
- Object-oriented programming
- POSIX compliant
- Multiple simultaneous DBM implementations
- Innumerable unbundled modules
- Simplified grammar
- Multithreading
- Lexical scoping
- Modularity and reusability
- Embeddable and extensible
- Package constructors and destructors
- Subroutine definitions may now be autoloaded
- Many usability enhancements

Perl's extensibility has given rise to an abundant repository of modules that add even more functionality to the core feature set. Perl is open source software and is available for a variety of operating systems, including UNIX (naturally) and its variants, MS-DOS, VMS, OS/2, AS/400, Macintosh, and Microsoft Windows whatever.

Some examples

We're going to just dive in now, and we're going to be looking at a lot of code, so hang on. To get acquainted with Perl's syntax and structure, we'll first run through a useful script that solves a real problem, but is only tangentially related to Oracle. Then we'll cover some of the basic techniques for hooking up to the database. Finally, we'll use some advanced features of PL/SQL in conjunction with Perl.

Managing archive files

The accompanying script was written to perform a few simple tasks that were consuming an unwarranted amount of time by a client database administrator. We simply want to *move* archive logs to a staging area if they're a certain age and have been backed up. Once in the staging area, we want to *delete* them after a certain age, but only if they've been backed up. Ideally, we'd also like to mail the results to the database administrator. Here are some notes to explain what's going on.

First, here's a sample configuration file:

```
# Configuration file for archive.pl
SOURCE      = z:/home/src/archive/testdir/src
DEST        = z:/home/src/archive/testdir/dst
LOG_DIR     = z:/home/src/archive
```

```

AGE_TO_MOVE      = 3
AGE_TO_DELETE    = 7
SENDER           = Roger D Gough      # profile name
RECIPIENT        = GoughRD

```

Now, on to the script ...

```

use File::Copy;
use Win32::File;
use Win32::File qw(GetAttributes);
use Win32::OLE;

require "ctime.pl";

```

use requests that the specified modules be accessible; **require** is similar, but is more like "include", whereas **use** is more like "link".

```

open(CONFIG, "archive.cfg") || die "Can't open configuration file.\n";

```

Open the configuration file, which is of the form VARIABLE=VALUE; **CONFIG** is the file handle that will reference the file. The `||` is a logical exclusive **or** (which we could also have used), so that if the **open** does not succeed, then we **die** (exit) with the specified message.

```

while (<CONFIG>) {
    chomp;          # remove line separator (the default is a newline)
    s/#.*//;       # "    comment lines
    s/^\s+//;      # "    leading blanks
    s/\s+$//;      # "    trailing blanks
    next unless length;
    ($var, $value) = split(/\s*=\s*/, $_, 2);
    $Config{$var} = $value;
}

```

This loop implicitly reads each line in **CONFIG** and performs the operations within the curly braces; it also handles the end-of-file condition. **chomp** removes the last character in a string, but only if it is the *Input Record Separator*, which in this case is the newline, or `\n`. The substitute command, `s/<regular expression>/<replace>/`, then helps us tidy things up. Here, we're simply replacing comments and blanks with nothing. **length** implicitly refers to the current line we're working with (`$_`), and in this case will return 0 if we've ended up with a blank line. **split** is instructed to parse `$_` and return two tokens as instructed by its first argument, which represents an "=" surrounded by white space. It then assigns the values to the list represented on the left hand side of the assignment. **\$Config**, because of the curly braces, creates a hash table, indexed by **\$var**, and assigns **\$value** to that element. This is just a simple symbol table.

```

$source          = $Config{"SOURCE"};
$dest            = $Config{"DEST"};
$page_to_move    = $Config{"AGE_TO_MOVE"};
$page_to_delete  = $Config{"AGE_TO_DELETE"};
$log_dir         = $Config{"LOG_DIR"};
$sender          = $Config{"SENDER"};
$recipient       = $Config{"RECIPIENT"};

```

Now we're just initializing variables with the values in the symbol table; they're a little easier to read that way. Note that the variables, scalars in this case, start with a `$`. Arrays start with `@` and hashes start with `%`.

```

$errors = 0;

($sec, $min, $hour, $mday, $mon, $year) = localtime(time);

```

`localtime()` returns a list of values; we're only interested in the first six -- the rest are discarded.

```
$log_file = sprintf "%s/arc-%4d-%02d-%02d.log", $log_dir, $year+1900, $mon+1, $mday;
open(LOG,">> $log_file") || die "Can't open log file.\n";
```

Here, we're creating the file name we'll use for logging messages and opening the file for appending (as directed by `>>`).

```
chdir($dest) || die "Can't change to directory $dest.\n";

# pipe the output of 'dir' to file handle DIRLIST
open(DIRLIST, "dir /b |") || die "Can't open directory $dest.\n";

print LOG "\n", &ctime(time);
print LOG "Deleting files ... \n from $dest\n\n";

while (<DIRLIST>) {
    chomp;
    $file = $_; # give $_ a meaningful name
    $age = int(-M $file);
    $is_file = -f $_;
    if ($is_file && $age >= $age_to_delete ) {
        GetAttributes($file, $attr);
        $backed_up = !($attr & ARCHIVE);
        if ($backed_up) {
            print LOG "$file (age=$age)\n";
            unlink($file);
        }
        else {
            $errors++;
            print LOG "ERROR: $file (age=$age) not backed up.\n";
        }
    }
}
close(DIRLIST);
```

Now we want to read the list of files in the destination directory. Because we're on NT here, we probably don't have `ls`, so we'll use a variation of `dir`. Rather than create a file with the list of file names, we'll read the names from a pipe, as indicated by the `|`.

`-M` returns the number of days since the file was modified; `int` has the effect of truncating that value. `-f` tells us if we're really looking at a regular file (as opposed to a directory or device). The `"_"` is just a shorthand way of referring to the last file that required `"stat"` to be called to figure all this stuff out. `GetAttributes` is part of the `Win32:File` module, and provides some useful information about `$file` – in this case, whether or not the archive bit is set. `unlink` deletes the file. Finally, we don't really have to `close` the file, but it's the nice thing to do.

```
chdir($source) || die "Can't change to directory $source.\n";

open(DIRLIST, "dir /b |") || die "Can't open directory $source.\n";

print LOG "\n", &ctime(time);
print LOG "Moving files ... \n from $source\n to $dest\n\n";

while (<DIRLIST>) {
    chomp;
    $file = $_;
    $age = int(-M $file);
```

```

    $is_file = -f _;
    if ($is_file && $age >= $age_to_move)
    {
        GetAttributes($file, $attr);
        $backed_up = !($attr & ARCHIVE);
        if ($backed_up) {
            print LOG "$file (age=$age)\n";
            move($file, "$dest/$file") || die "Can't move $file.\n";
        }
        else {
            $errors++;
            print LOG "ERROR: $file (age=$age) not backed up.\n";
        }
    }
}
close(DIRLIST);
close(LOG);

```

Pretty much the same thing as above; probably time to think about a subroutine.

Now we begin the mail notification. This turned out to be one of the trickier parts of the script, primarily because I don't know much about OLE. But my faith that someone had already figured out how to do this led me to a place from which I stole much of the code. There was also actually some useful information on Microsoft's web site.

```

$session = Win32::OLE->new("MAPI.Session") || die "Could not create a new MAPI Session: $!\n";

```

This illustrates the creation of a Perl object, in this case the mail **session**. Note here that we're using **#!** in the argument to **die** which is a builtin variable containing the error message.

```

$session->Logon($sender) && die "Logon failed: $!\n";

```

And this illustrates how to access a method for an object. The session object is documented in the Microsoft SDK, and is available online.

```

$msg = $session->Outbox->Messages->Add();

$rcpt = $msg->Recipients->Add();
$rcpt->{Name} = $recipient;
$rcpt->Resolve();

$msg->{Subject} = "Archive log results ($errors error(s))";

$position = 0;
$CdoFileData = 1; # attachment is the contents of a file

$msg->Attachments->Add($log_file, $position, $CdoFileData, $log_file);

$msg->Update();
$msg->Send(0, 0, 0);
$session->Logoff();

```

And we're done.

Basic access to Oracle

Well, all this is fine and good ... very good indeed. But what about getting at a database? Enter Perl's inimitable extensibility and the imagination of Tim Bunce, who initiated and oversees the DBI project. DBI is the extension that provides an interface to many disparate database systems, including Oracle, through database drivers, or DBDs. The basic architecture is illustrated in **Figure 1**.

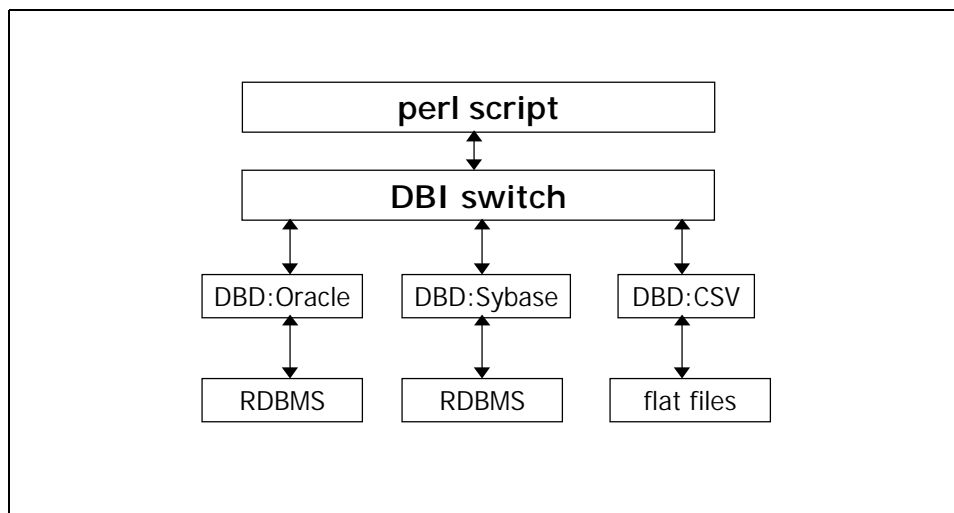


Figure 1. DBI – The Database Interface

Loading data

The first step is, of course, making the connection. The DBD:Oracle driver uses SQL*Net, so make sure that your network is properly configured.

```
#!/usr/bin/perl
use DBI;
$host = "orcl"; $user = "scott"; $pass = "tiger";
$dbh = DBI->connect("dbi:Oracle:$host", $user, $pass);
```

We need to `use DBI` in order to harness the driver. Then we create a database handle, `$dbh`, which will give us access to our connection. By the way, the first line instructs the shell to interpret the succeeding commands with Perl, not itself.

```
$sth = $dbh->prepare("SELECT * FROM demo");
$sth->execute;
print("\nSELECT\n");
while (@row = $sth->fetchrow()) {
    print "Row returned: @row\n";
    # print "Row returned: $row[0] $row[1] $row[2]\n";
}
```

`$sth` is our statement handle. The logic here resembles dynamic SQL in the sense that we have separate steps for preparation and execution. Notice that `@row` is shorthand for all the columns. Here are the results:

```

SELECT
Row returned: 10 ACCOUNTING NEW YORK
Row returned: 20 RESEARCH DALLAS
Row returned: 30 SALES CHICAGO
Row returned: 40 OPERATIONS BOSTON

```

Now, we'll load two more rows into the table which we'll retrieve from the file:

```

demo.dat
50,TRAINING,RICHMOND
60,MANUFACTURING,ROANOKE

```

```

open(DATA,"demo.dat");
print("\nINSERT\n");
$stmt = $dbh->prepare ("INSERT INTO demo VALUES (?, ?, ?)");
while (<DATA>) {
    chomp;
    ($deptno, $dname, $loc) = split(/,/, $_, 3);
    $stmt->execute($deptno, $dname, $loc);
}
close(DATA);

```

First, we open the file as we've seen before and read a line at a time, parsing it into its three components. We also `prepare()` the `INSERT` statement ahead of the loop to avoid redoing it with every line. After closing the file, the loop we used in the above `SELECT` example produces:

```

INSERT
Row returned: 10 ACCOUNTING NEW YORK
Row returned: 20 RESEARCH DALLAS
Row returned: 30 SALES CHICAGO
Row returned: 40 OPERATIONS BOSTON
Row returned: 50 TRAINING RICHMOND
Row returned: 60 MANUFACTURING ROANOKE

```

To illustrate `DELETE`, we'll remove the rows we just added.

```

print("\nDELETE\n");
$dbh->do("DELETE FROM demo WHERE deptno >= 50");
$stmt = $dbh->prepare("SELECT * FROM demo");
$stmt->execute;
while (($deptno, $dname, $loc) = $stmt->fetchrow()) {
    print "deptno: ", $deptno, "\n";
    print "dname:  ", $dname, "\n";
    print "loc:    ", $loc, "\n\n";
}

```

Notice that we use `do()` here to execute the `DELETE` command immediately. Also shown is an alternative way to fetch the rows from the `SELECT` statement by giving each column its own name.

```

DELETE
deptno: 10                deptno: 30
dname:  ACCOUNTING        dname:  SALES
loc:    NEW YORK          loc:    CHICAGO

deptno: 20                deptno: 40
dname:  RESEARCH          dname:  OPERATIONS
loc:    DALLAS            loc:    BOSTON

```

And when we're done, we should politely `$dbh->disconnect`.

Accessing OS commands from PL/SQL

Our final example will use the `DBMS_PIPE` built-in package to establish communication between two Oracle sessions, one of which is embedded in a Perl script. Once we have this connection, the Perl script will act as a server to issue commands to the operating system. We have used this technique, for example, to unzip zipped files which are then processed by SQL*Loader, and report results by sending mail to a specified recipient. **Figure 2** illustrates the architecture of this technique.

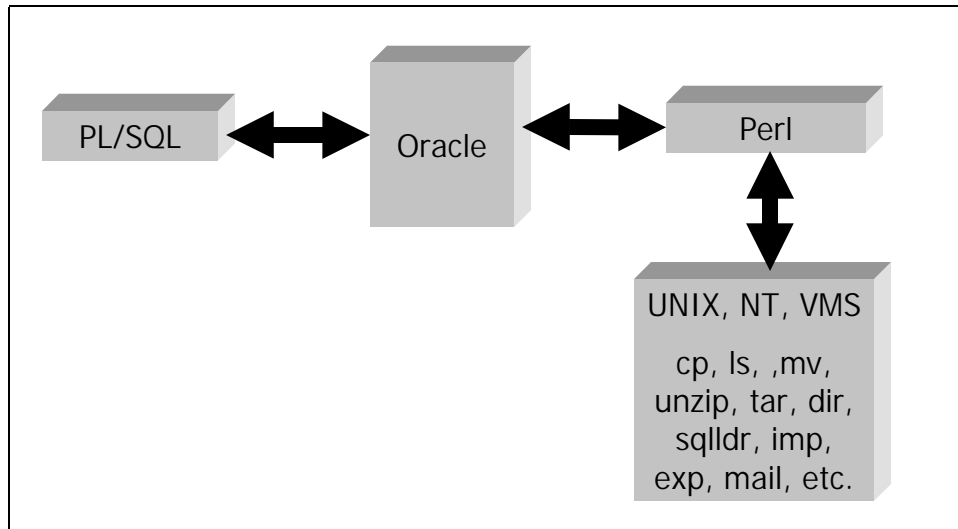


Figure 2. Interprocess Communication

Let's begin by presenting a minimal PL/SQL package which may be accessed by any environment which uses PL/SQL, such as SQL*Plus or Developer or WebDB.

```

CREATE OR REPLACE PACKAGE Perld AS

    pipe_name      VARCHAR2(100) := 'pipe_name';

    FUNCTION host(command IN VARCHAR2) RETURN NUMBER;
    PROCEDURE stop;

END Perld;
  
```

We'll define two routines here: `host`, which will issue a command to the operating system, and `stop`, which terminates the Perl daemon that is serving the requests. The basic steps consist of packing messages in a well-defined way, then sending the message along the sending pipe to the daemon. Depending upon the protocol established, we may also want to receive data back from the daemon. In this case, each client is assigned its own receiving pipe along which to receive data; note that we replace '\$' with '-' to avoid conflicts with UNIX shells.

```

CREATE OR REPLACE PACKAGE BODY Perld AS

    ret_pipe_name  VARCHAR2(128) := REPLACE(dbms_pipe.unique_session_name, '$', '-');
    maxwait        NUMBER := 2;

    PROCEDURE stop AS
        rc          NUMBER;

    BEGIN
        dbms_pipe.reset_buffer;
        dbms_pipe.pack_message('stop');
  
```

```

    rc := dbms_pipe.send_message(pipe_name, maxwait);

END; --stop

FUNCTION host(command IN VARCHAR2) RETURN NUMBER AS
    rc          NUMBER;
    os_rc       VARCHAR2(5);

BEGIN
    dbms_pipe.purge(ret_pipe_name);
    dbms_pipe.reset_buffer;
    dbms_pipe.pack_message('host');
    dbms_pipe.pack_message(command);
    dbms_pipe.pack_message(ret_pipe_name);
    rc := dbms_pipe.send_message(pipe_name, maxwait);

    rc := dbms_pipe.receive_message(ret_pipe_name, maxwait);
    dbms_pipe.unpack_message(os_rc);

    RETURN TO_NUMBER(os_rc);

END; --host

END Perld;

```

The simplest command is `stop`, which simply packs the message and sends it. `host` is only slightly more complicated: its message consists of (1) the keyword `'host'`, (2) the command to be issued to the operating system, and (3) the name of the pipe along which to receive the OS return code. This return code is passed back as a string to simplify the protocol.

Now, on to the daemon. Most of the interesting things occur in the subroutines, which we'll get to after first getting a feel of what needs to happen.

```

#!perl

use DBI;
use File::Basename;
use File::Copy;
use Config;

get_config();

$host  = $localConfig{"HOST"};
$user  = $localConfig{"USER"};
$pass  = $localConfig{"PASS"};
$loader = $localConfig{"LOADER"};

$bufsz = 2000;
$pathsep = ($Config{osname} eq 'MSWin32') ? "\\\" : "/";

$pipe = open_database();

```

The first part of the script is devoted to initializing the environment. We need to `use` several modules, then we retrieve values from a configuration file which will be needed in the script, primarily for connecting to the database. Because we'd like our script to be portable, we check to see if we're on a Windows system, so that we can set `$pathsep` to be a backward rather than forward slash.

Now we get to the main loop:

```

while(1) {
    get_message();
    $cmd = lc get_packet();
    print "\ngot message: <$cmd>.\n";
    for ($cmd) {
        /^host$/ && do {
            $cmd = get_packet();

```

```

        print "got host command: <$cmd>.\n";
        $dest_pipe = get_packet();
        $rc = system $cmd;
        $rc >= 8;
        put_packet($rc);
        put_message($dest_pipe);
    };
    /^dir$/ && do {
        chdir get_path();
        @files = glob(get_packet());
        $dest_pipe = get_packet();
        foreach $file (@files) {
            put_packet($file);
        }
        put_message($dest_pipe);
    };
    /^mail$/ && do {
        $to = get_packet();
        $subj = get_packet();
        open MAIL, qq{| mail -s "$subj" $to};
        while ($line = get_packet()) {
            print MAIL $line;
        }
        close MAIL;
    };
    /^ping$/ && do {
        $dest_pipe = get_packet();
        put_packet(1);
        put_message($dest_pipe);
    };
    /^pathsep$/ && do {
        $dest_pipe = get_packet();
        put_packet($pathsep);
        put_message($dest_pipe);
    };
    /^stop$/ && do {
        close_database;
        exit;
    };
}
}

```

The loop shows the daemon's implementation of several commands: `host`, `dir`, `mail`, `ping`, `pathsep`, and `stop`. The possibilities are literally endless, but the idea is always the same: determine the nature of the request, perform it, and possibly return results back to the client. Indeed, sometimes the trickiest part is working out the handshaking between client and server (daemon).

One new idiom introduced here is `qq{ ... }`. It's simply a nifty way to say quote-quote (double-quote), when what you want to double-quote already has double-quotes. We'll use it frequently. And you can quote me on that.

```

sub get_message
{
    print "perld waiting ... \n";
    my $csr = $dbh->prepare(qq{
        declare
            rc number;
        begin
            rc := dbms_pipe.receive_message('$pipe');
        end;
    });
    $csr->execute;
}

```

`get_message()` simply waits to receive a message from a client. Here is our first example of embedding the language elements of PL/SQL inside a Perl script. It's also our first subroutine, and `my` makes the `$csr` cursor invisible outside the subroutine, i.e., it performs lexical scoping.

```
sub get_packet
{
    my $buf;

    my $csr = $dbh->prepare(qq{
        declare
            buf varchar2($bufsz);
        begin
            if dbms_pipe.next_item_type > 0
            then
                dbms_pipe.unpack_message(buf);
                :buf := buf;
            else
                :buf := '';
            end if;
        end;
    });
    $csr->bind_param_inout(":buf", \$buf, $bufsz);
    $csr->execute;
    return $buf;
}
```

Once we've got our message, it needs to be unpacked, and `get_packet()` does that for us. `bind_param_inout()` takes the contents of the placeholder `:buf` and places it into Perl variable `$buf` by means of a pointer to it, i.e., `\$buf`.

The remaining subroutines are shown here for completeness, but contain little new otherwise.

```
sub put_packet
{
    my ($buf) = @_;

    my $csr = $dbh->prepare(qq{
        declare
            buf varchar2($bufsz);
        begin
            buf := :buf;
            dbms_pipe.pack_message(buf);
        end;
    });
    $csr->bind_param_inout(":buf", \$buf, $bufsz);
    $csr->execute;
}

sub put_message
{
    my ($dest) = @_;

    my $csr = $dbh->prepare(qq{
        declare
            rc number;
        begin
            rc := dbms_pipe.send_message('$dest');
        end;
    });
    $csr->execute;
}

sub open_database
{
    my $buf

    $dbh = DBI->connect("dbi:Oracle:$host", $user, $pass);
    my $csr = $dbh->prepare(qq{
```

```

        declare
            buf varchar2($bufsz);
        begin
            :buf := perld.pipeName;
            dbms_pipe.purge(perld.pipeName);
            dbms_pipe.reset_buffer;
        end;
    });
    $csr->bind_param_inout(":buf", \ $buf, $bufsz);
    $csr->execute;
    return $buf;
}

sub close_database
{
    my $csr = $dbh->prepare(qq{
        begin
            dbms_pipe.remove_pipe(perld.pipeName);
        end;
    });
    $csr->execute;
    $dbh->disconnect;
}

sub get_config
{
    open CONFIG, "perld.rc";

    while (<CONFIG>) {
        chomp;
        s/#.*//;
        s/^\s+//;
        s/\s+$//;
        next unless length;
        ($var, $value) = split(/\s+=\s+/, $_, 2);
        $localConfig{$var} = $value;
    }

    close CONFIG;
}

sub get_path
{
    my $path = get_packet();
    $path =~ s/\\\/\//g;
    return $path;
}

```

Conclusion

We've come a long way. And this is certainly more than enough to get started. But, as you can imagine, we've really only scratched the surface of subjects like regular expressions, pattern matching, lists, hashes, references, packages, modules, and object classes. And we haven't even mentioned nested data structures, graphics, networking, security, or the Tk toolkit for building GUIs. More than enough for years of amusement. Enjoy!

Resources

As one might expect, many of the best resources are available on the web. Coupled with the exceptional books published by O'Reilly & Associates, Inc., especially *Programming Perl* and the recent *Programming the Perl DBI*, there is an abundance of information out there. Here are some links to get you started:

- www.perl.com – the official site with links to CPAN (Comprehensive Perl Archive Network) and others.
- www.activestate.com – the source for Microsoft Windows ports of Perl.
- www.oreilly.com – why turn to anyone else for printed documentation?
- www.bookpool.com – some of the best prices on the net and they carry O'Reilly.

