Advanced Perl Programming

Student Workbook

itcourseware

TRAINING MATERIALS FOR IT PROFESSIONALS

ADVANCED PERL PROGRAMMING

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112

Contributing Authors: Brandon Caldwell, Jeff Howell, Jim McNally, Rob Roselius

Editor: Rob Roselius

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Michael Naseef and Richard Raab, and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

CHAPTER 1 - COURSE INTRODUCTION

Course Objectives

- Expertly manipulate lists, arrays, and hashes.
- Exploit code references and closures.
- Use **eval** to run dynamically generated Perl code, and to trap exceptions.
- Create and use object-oriented Perl modules.
- Tie Perl variables to subroutines to customize access and assignment.
- Access UNIX DBM files efficiently from Perl.
- Find and effectively use the thousands of freely available Perl modules.
- Access database management systems from Perl programs using DBI.
- Write GUI application components quickly using Perl/Tk.
- Extend Perl with modules that load C/C++ object code.
- Interface Perl with, and embed Perl in, C/C++ applications.
- Describe the internal representation of Perl's various datatypes and data structures.
- Write reusable Perl modules.
- Avoid coding practices that hurt performance and maintenance.

In this course, it is assumed that you have:

1. Completed training in Perl Programming.
2. Programmed extensively in Perl for at least several months.

Or, if you have not attended Perl Programming training, you have programmed extensively in Perl for at least a year or two. This is a wide-ranging course which addresses topics including Perl internals, coding techniques, use of various Perl modules, and interfacing Perl with C and C++. You will write many programs in this class, large and small.

Some aspects of Perl that you should already be comfortable with include:

- Language syntax fundamentals, including:
  - Commonly used special variables and shortcuts
  - Perl datatypes: scalars, lists, hashes
  - Operators and expressions
  - Quoting and interpolation
- References - these are especially important
- Contexts - list, scalar, string, numeric
- Perl regular expressions
- Anonymous arrays and anonymous hashes
- Input and output
- Packages, use, and require
- CPAN
- Very basic C-language features
- User-level or better UNIX skills

Every programmer takes a unique career path, encounters a unique sequence of programming situations, and has unique strengths and weaknesses; it's possible that you are less than familiar with one or two of these topics (which is by no means exhaustive). Don't worry, you'll still have a great experience in this class, and maybe fill some gaps in your Perl repertoire beyond the contents of this workbook! And it's also possible that you're more than familiar with the material in some chapters. Stay tuned-in though, and don't be surprised if you learn something in an area you thought you knew inside-out.

If more than a few of the topics mentioned above are new to you, you might want to have a chat with the instructor about what you can expect to accomplish in this class — which could still be a great deal; we just want to be sure you don't get frustrated, you learn a lot, and you have fun!

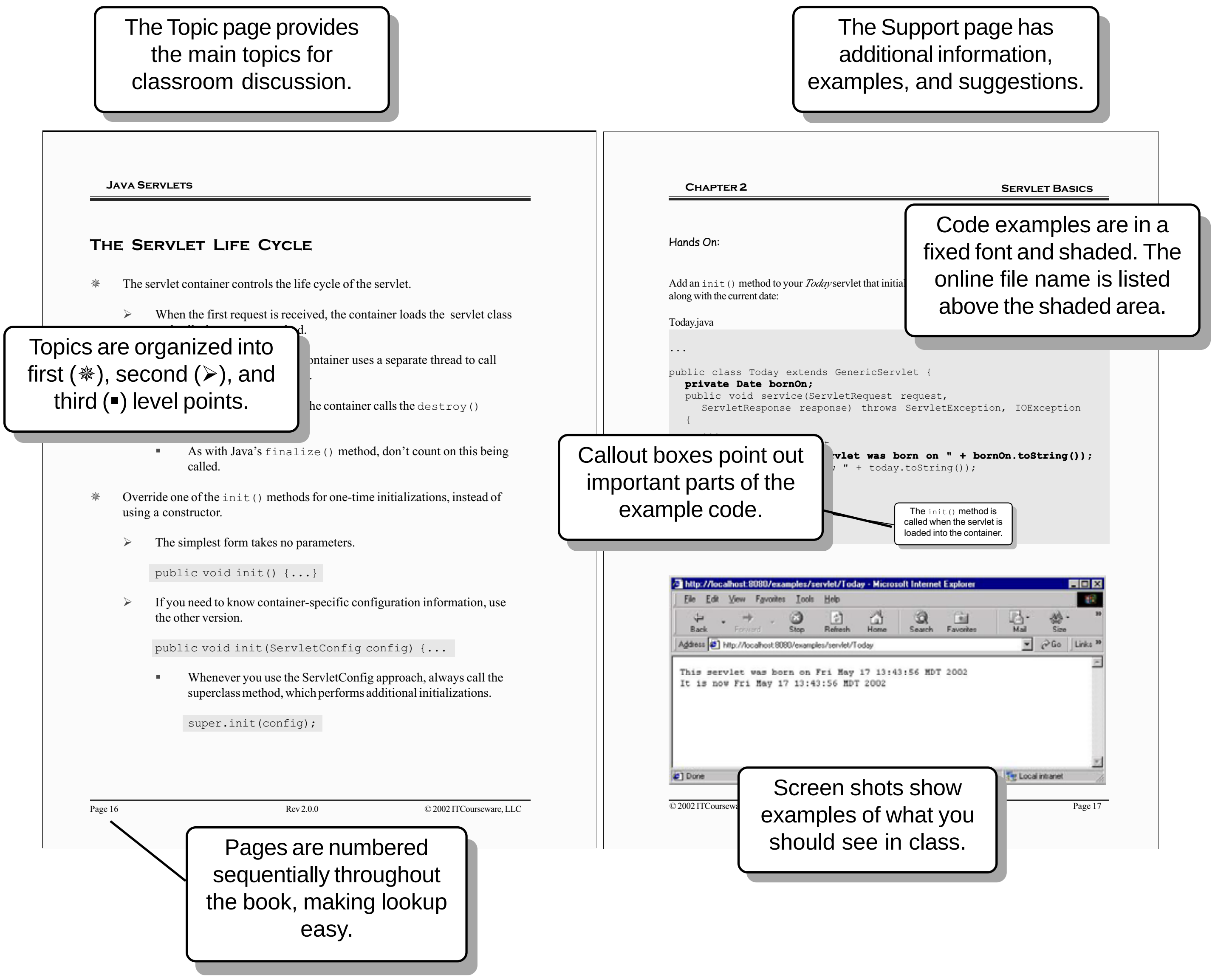
COURSE OVERVIEW

- **Audience**: Application programmers, system administrators, website authors, webmasters, and UNIX/NT power users.
- **Prerequisites**: *Perl Programming on Unix* and at least several months real experience programming in Perl. Comprehension of the extending, embedding, and internals material will require some C or C++ programming experience.
- **Classroom Environment:**
  - UNIX X-Windows workstation per student.
  - Perl5.004+ installed.
  - C compilation system.
  - DBMS with supporting Perl DBI/DBD modules.
  - Perl Tk[*48*]*??.???* module installed and tested.
  - Additional modules: Devel::Peek, Data::Dumper.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. Hands On sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick lookup. Printed lab solutions are in the back of the book as well as online if you need a little help.



The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples, and suggestions.

Topics are organized into first (❋), second (➢), and third (▪) level points.

Code examples are in a fixed font and shaded. The online file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

Screen shots show examples of what you should see in class.

Pages are numbered sequentially throughout the book, making lookup easy.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

❋ The servlet container controls the life cycle of the servlet.

➢ When the first request is received, the container loads the servlet class d.

ontainer uses a separate thread to call .

he container calls the `destroy()`

▪ As with Java's `finalize()` method, don't count on this being called.

❋ Override one of the `init()` methods for one-time initializations, instead of using a constructor.

➢ The simplest form takes no parameters.

```
public void init() {...}
```

➢ If you need to know container-specific configuration information, use the other version.

```
public void init(ServletConfig config) {...
```

▪ Whenever you use the ServletConfig approach, always call the superclass method, which performs additional initializations.

```
super.init(config);
```

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC

CHAPTER 2 SERVLET BASICS

Hands On:

Add an `init()` method to your *Today* servlet that initial along with the current date:

Today.java

```
...

public class Today extends GenericServlet {
  private Date bornOn;
  public void service(ServletRequest request,
    ServletResponse response) throws ServletException, IOException
  {
vlet was born on " + bornOn.toString());
  " + today.toString());
```

The `init()` method is called when the servlet is loaded into the container.

http://localhost:8080/examples/servlet/Today - Microsoft Internet Explorer

File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Mail Size
Address http://localhost:8080/examples/servlet/Today
Go Links
Done Local intranet

```
This servlet was born on Fri May 17 13:43:56 MDT 2002
It is now Fri May 17 13:43:56 MDT 2002
```

© 2002 ITCoursew Page 17

Suggested References

Christiansen, Tom and Nathan Torkington. 2003. ***Perl Cookbook, Second Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596003137.

Friedl, Jeffrey E. F. 2006. ***Mastering Regular Expressions, Third Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596528124.

Hall, Joseph and Randal Schwartz. 1998. ***Effective Perl Programming***. Addison-Wesley, Reading, MA. ISBN 0201419750.

Cozens, Simon. 2005. ***Advanced Perl Programming***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596004567.

Wall, Larry, Tom Christiansen, and Jon Orwant. 2000. ***Programming Perl***. Third Edition. O'Reilly & Associates, Sebastopol, CA. ISBN 0596000278.

http://www.perl.com
http://www.perl.org

news:comp.lang.perl.misc
news:comp.lang.perl.modules
news:comp.lang.perl.tk
news:comp.lang.perl.announce
news:comp.lang.perl.moderated

Perl, the archetype for free/open-source software, was supported for years by its original author Larry Wall. Time was, you'd send an email to Larry, and he'd have a fix or enhancement in the Perl source code distribution in weeks, days, or overnight. Larry has long since ceded the dirty work of maintaining the Perl source code to other dedicated hackers and is unlikely to answer your Perl questions by email (though he might, you never know).

Perl is one of the best documented languages available. Quite apart from the several fine books available, every Perl installation comes with complete, up-to-date documentation including almost 100 pages of FAQs (nearly 300 answers to common questions), and over 600 pages of technical reference and tutorials. Find Perl's Plain Old Documentation (POD) in ***/usr/local/lib/perl5/pod/***.

For many years businesses were reluctant to adopt Perl as a production development and deployment tool due to concerns of "Where's the support?" and, perhaps, "Who's liable (who do we sue)?" Perlers knew all along, of course, that Perl's support is higher quality and more responsive than that for just about any other software you could name. Perl's support department is the Internet, available 24/7, staffed by tens of thousands of experienced coders.

For a long while, the USENET newsgroup ***comp.lang.perl*** (***clp*** to its subscribers) was the point of discussion. This group has been superseded by subgroups of ***comp.lang.perl*** newsgroup hierarchy (***comp.lang.perl.misc*** or ***clpm***, primarily) as the readership grew and specialized. While some news servers haven't fully removed the original ***clp*** group, you should start at ***comp.lang.perl.misc*** these days. Woe unto you if you post without first checking the FAQ.

For those actively working to keep the latest Perl releases compiling on its many supported platforms, there's the ***perl5-porters*** mailing list. Don't pester these folks with basic questions, though; this list is for the busy, dedicated folks keeping Perl going for the rest of us.

As the Web emerged, Tom Christiansen started the Perl home site, ***www.perl.com***. This site includes links to archives of the Perl source code, as well as modules, links, and other resources. He maintained this extremely popular and busy site for years over a 28,800 modem (using clever Perl networking techniques, of course) until O'Reilly & Associates offered to sponsor the site. Give it a visit.

While there are now many consultants and small companies providing Perl support services, and commercial journals are in publication (see archives of ***The Perl Journal***, at ***www.tpj.com***), the Internet is still the official Perl support department, and users (programmers like us) still get to exercise our resourcefulness and curiosity as we pursue solutions and understanding. Happy browsing!

CHAPTER 2 - DEBUGGING

OBJECTIVES

- ❈ Enable, and temporarily disable, warnings in various scopes.
- ❈ Get verbose diagnostics for debugging.
- ❈ Generate appropriate warnings and exits in your own code.
- ❈ Enable strict compiler checking of variables, references, and barewords.
- ❈ Temporarily disable strict checking.
- ❈ View data structures, internal values, and stack traces.

WARNINGS

- **perl -w** prints wisdom that you might otherwise miss, such as:
  - **if ($x = $y)**... instead of **if ($x == $y)**...
  - Non-numeric scalars in a numeric expressions.
  - Variables that are only used once (helps catch typos).
  - Uninitialized variables.
  - Subroutines that recurse more than 100 times.
  - And many more, both compile-time and runtime errors.
- **-w** sets the global **$^W** variable (that's either $<Ctrl>W or $^W (carat-W)) — you can change, and even localize, **$^W** at runtime.

```
$^W = 1;  # Turn on warnings
{  ## I want to do some dangerous stuff here...
   local $^W = 0;   # Set to 0 until end of block
```

- Set **$SIG{__WARN__}** to trap warnings and call your own function.

```
$SIG{__WARN__} = \&callbackFunction;
```

  - Any time a warning is generated, or **warn()** is used, your callback function will be invoked.
  - The warning message is sent to the callback as its first parameter.
  - Do this in a **BEGIN** block to catch compiler warnings.

Hands On:

Run this sample program and examine the output. Make sure you understand the reason for every warning:

warntest.pl

```
print "\$^W is $^W\n";

while ($l = <STDIN>) {1;}

#BEGIN {
#   $SIG{__WARN__} = \&myWarning;
#}

print "x is '$x'\n";
&scopeIt();
print "z is '$z'\n";

warn "Warning: You should rotate your tires periodically.\n";

sub scopeIt {
  local $^W = 0;
  print "y is '$y'\n";
}

sub myWarning {
  print "Got a warning that starts: '",
         substr($_[0],0,40), "...'\n";
}
```

Now, uncomment just the **$SIG{__WARN__}** line, and run it again. Which messages are compile-time warnings and which are runtime warnings? Why?

Finally, uncomment the **BEGIN** block around the **$sig{__WARN__}** line and run it again.

DIAGNOSTIC MESSAGES

- For verbose descriptions of warnings, use **diagnostics;**.
  - This automatically enables the **-w** flag.
  - For each warning, you'll get an additional, descriptive paragraph.
  - You may enable or disable these diagnostic messages during runtime by using **diagnostics::enable()** and **diagnostics::disable()**.
- The **-verbose** option prints the **perldiag** man page introduction, which describes the various levels of warnings, before any other diagnostics.
- To get diagnostic output without modifying your code, you can use **-Mdiagnostics**.

```
perl -Mdiagnostics prog.pl
```

```
perl -M'diagnostics -verbose' prog.pl
```

- Strive to get your code running cleanly under **-w** before putting it into production.

Hands On:

Add a **use diagnostics** statement to the program from the previous page and run it.

warntest.pl

```
print "\$^W is $^W\n";

use diagnostics;
...
```

Change it to **use diagnostics -verbose** and run it again. Peruse the output.

The question of whether to use **-w**, and other diagnostics-producing options, in the production runtime environment is the subject of passionate debate (so don't bring it up in ***comp.lang.perl.misc*** unless you're wearing your asbestos suit). Certainly it can cause failure and explosive logfile growth for "trivial" reasons in CGI scripts, for example (some argue that there are no trivial warnings, of course). New releases of Perl may add disconcerting warning messages that older scripts didn't count on; it may not be practical to retest every script before upgrading your Perl. You can, of course, use a **$SIG{__WARN__}** handler to insulate end users from warnings, and quietly log (or transmit to you) the diagnostic information. On this somewhat religious issue, you must consult your own conscience and experience.

Some warnings are themselves considered bugs. Perl5.004 introduced a new warning when you use constructs like:

```
while ($line = <FH>) { ...
```

This produces the warning:

```
Value of <HANDLE> construct can be "0"; test with defined()...
```

(If the last line of input is a 0 with no trailing newline, **$line** will be false before the true **EOF**.) This breaks a great deal of perfectly good existing code and the warning is removed in Perl5.005.

Carping, Confessing, and Croaking

- You probably use **warn** to print your own warnings to **STDERR**.
  - If you don't provide a message, **warn** issues a default warning, indicating something wrong at the line it was invoked.
  - If you provide a message to **warn** and don't put a new line at the end of it, the line number and current file name are added.
- **die** behaves just like **warn**, but exits after printing.
- **warn** and **die** report the line number and filename of the file in which they occur; if that file was included with **require** or **use**, it reports the included file, not the file that included it.
- The **carp** module provides alternative error routines.
- **confess** behaves just like **die**, but prints a stack trace before exiting.
- Your function can **carp** a message and the code that invokes your function will have its filename and line number shown.
  - **carp** behaves like **warn**, but with the caller's file and line number reported.

    ```
    carp "function expects filename, using STDIN";
    ```

  - **croak** behaves like **die**, but with the caller's information reported.

    ```
    croak "function requires filename";
    ```

- In module code, **carp** and **croak** are preferable to **warn** and **die**.

Hands On:

Run the *carptest.pl* program. Note the file and line numbers reported by the warning and error messages.

Now, uncomment the lines in *myTest.pm* and run the *carptest.pl* program again. (Of course, only the first of **confess** or **croak** will work, so uncomment them individually.)

mytest.pm

```
package myTest;

#use Carp;
sub printIt {
   print "myTest::printIt -> $_[0]\n";
   warn "This is a warning from myTest::printIt";
   die "die called from myTest::printIt";
   #carp "This is a carp message generated by myTest::printIt";
   #confess"confess called from myTest::printIt";
   #croak  "croak called from myTest::printIt";
}
1;
```

carptest.pl

```
use myTest;     #using the module

myTest::printIt("Hi MOM!");
```

STRICT CHECKS

- By default, the **use strict;** pragma tells the compiler to be strict about three kinds of usage:
  - Variables that are not declared, fully qualified, lexical, or imported.
  - References that are symbolic.
  - Subroutines that are not declared, or barewords that aren't hash keys or used with the => operator.
- You may provide an argument that will turn on strict checking for only that usage:

```
use strict 'vars';
use strict 'refs';
use strict 'subs';
```

- You can scope the checking to any block or use it for the entire program.
- You will find that using strict checks from the beginning of your development effort will greatly reduce the number of debugging headaches you might otherwise encounter.
  - It is difficult to retrofit a program with strict checking, so use it from the start.
- Strive to get your code running cleanly under **use strict** before putting it into production.

Hands On:

In the following code, uncomment each **use strict** line one at a time and run the program.

stricttest.pl

```
#use strict;
&strictVars();
&strictRefs();
&strictSubs();

sub strictVars {
  #use strict 'vars';
  @a = (6);
  print "strictVars: \@a is @a\n";    # Undeclared package var
}

sub strictRefs {
  #use strict 'refs';
  @names = ('Fido', 'Spot', 'Kiki');
  my $symRef = "names";
  print "strictRefs: \@\$symRef is @$symRef\n"; # Symref @'names'
}

sub strictSubs {
  #use strict 'subs';
  my $name = Francis; # Bareword Francis used for quoted string
  print "strictSubs: \$name is $name\n";
}
```

COMPILER PRAGMAS

- Using other compiler pragmas will help you write more bulletproof code.

  - When performing strict checks on **'vars**,**'** you can do a pseudo-import on some symbols, which will allow you to use them without fully qualifying them.

    ```
    use strict 'vars';
    use vars qw($grandtotal @employee);
    ```

  - You can override built-in functions or call functions without enclosing their parameter arguments in parentheses.

    ```
    use subs qw(maxCostSub createEmployee);
    ```

- You can use your own Perl modules or those that you've downloaded from CPAN by using the appropriate library:

  ```
  use lib "/home/me/my_perl";
  ```

  - This is preferred over modifying the **@INC** array in your programs to include your module paths.

- To turn off a pragma, use **no**:

  ```
  no strict 'refs';
  no vars;
  no integer;
  ```

Other pragmas include:

```
use sigtrap LIST;
   # print a stacktrace of the program if a signal
   # in the list is generated

use sigtrap;
   # print a stacktrace of signals:
   # ABRT, BUS, EMT, FPE, ILL, PIPE, QUIT,
   # SEGV, SYS, TERM, and TRAP.

use integer;
   # force arithmetic to be performed in the more
   # efficient integer mode
```

useinteger.pl

```
use strict;

my $total = &add(2.5,2.5);
print "\$total is $total\n";

sub add {
  #use integer;     # uncomment to see the effect of use integer;
  my(@args) = @_;
  my $total;
  my $value;
  foreach $value (@args) {
    $total += $value;
  }
  return $total;
}
```

DEBUGGING FLAGS

- Perl's **-D** switch generates many kinds of extra information from the compiler and interpreter.
  - This information is sent to standard error.
  - **-Dx** (or **-D1024**) prints a syntax tree dump — opcodes, et al.
  - **-Ds** (or **-D2**) prints a stack snapshot before each opcode.
  - **-Dt** (or **-D8**) prints an execution trace.
  - You may combine these options, either by summing their numeric values or combining their letter values in a list:

    ```
    perl -Dst myprog.pl 2>dst.out
    perl -D10 myprog.pl 2>dst.out
    ```

- Within your program, the numeric sum of the currently set **-D** options is available in the special variable **$^D**.
- The **-D** switch is available only with Perl interpreters that have been compiled with the **-DDEBUGGING** compiler switch.
  - The **-DDEBUGGING** option creates a slower running *perl* executable, so you should keep both versions available.

    ```
    /usr/local/bin/perl
    /usr/local/bin/debugperl
    ```

Hands On:

Read the **perlrun** manpage to see a listing of all the **-D** flags and their meanings. Run a very simple Perl script with a debug flag:

```
perl -Dx -e 'print "Hello\n"'
```

See how much of the output you can figure out. Try it with some other **-D** options, like **-Dt** and **-Ds**. While using the **-Ds** option, pass some additional arguments to print.

Test the following program with several of the **-D** options, one at a time. Be sure to try **-Dx**, **-Ds**, **-Dl**, **-Dt**, **-Dc**, and **-Dr**. Redirect the standard error output to a file, then examine it (e.g., **perl -Dr dtest.pl 2>dr.out**).

Note that the program wants a valid file name as its first argument and, in the regular expression, searches for lines in the file which begin with **u** or **p** and prints them out.

dtest.pl

```
use strict;
my ($filename) = @ARGV;
open (FH, $filename) or die "Can't open $filename";
my $x = "123";
print "$x\n";
$x += 4;
print "$x\n";
while (my $line = <FH>) {
  print $line if ($line =~ /^p|^u/);
}
```

The following abbreviations will appear in some of the output:

SV	Scalar Value
HV	Hash Value
IV	Integer Scalar Value
CV	Code Value
PV	String Scalar Value
GV	Glob Value (typeglob)
NV	Double Scalar Value
RV	Reference Value
AV	Array Value

YOUR PERL CONFIGURATION

❈ If you've ever built and installed Perl yourself, you know how many questions are asked and answered by the **Configure** script about your particular system.

- ➢ **Configure** puts this information in *config.sh*, which is used for generating **Makefile**s, etc.
- ➢ You may be interested in these details if you're integrating Perl with other applications or seeking help with debugging.

❈ Use **perl -V** to print a summary of the major configuration values.

```
perl -V
```

❈ Methods in the *Config.pm* module can provide you with all of the configuration information programmatically.

- ➢ **Config::myconfig()** returns the same information as the **-V** command-line option, as a single string value.
- ➢ **Config::config_sh()** returns all of the configuration values, as a single and very long string.
- ➢ **Config::config_vars(@*varlist*)** prints the names and values of the variables named in **varlist** to **STDOUT**, suitable for capture by other programs and scripts.

❈ The **ExtUtils** module (used when embedding **perl**, etc.) can provide similar information.

configtest.pl

```
use Config;

#print &Config::myconfig(),"\n";      # Returns "perl -V" info.

#print &Config::config_sh(),"\n";     # Full contents of config.sh!

@cvars=qw(ccflags ldflags perlpath privlib sitearch sitelib);
&Config::config_vars(@cvars);
```

Hands On:

Run the following commands to retrieve the configuration variables, compiler options, and link options used to build your Perl interpreter:

```
perl -MConfig -e 'print Config::config_sh'
perl -MConfig -e 'Config::config_vars(libs)'
perl -MExtUtils::Embed -e ccopts
perl -MExtUtils::Embed -e ldopts
```

The Devel::Peek Module

- **Devel::Peek** lets you look at internal Perl structures and data values.
  - If you understand the contents of the **perlguts** manpage, you'll understand the output of this module (and vice versa).
  - At its simplest, and perhaps most useful level, using the module involves two steps:
    1. Import the **Devel::Peek** module.

       ```
       use Devel::Peek; # or
       use Devel::Peek('Dump'); # only dump values
       ```

    2. Dump the information you're interested in.

       ```
       Devel::Peek::Dump($somevar);
       ```

- **Devel::Symdump** is a primitive package designed to be subclassed, which allows you to dump the symbols in a package.
  1. Create the **Devel::Symdump** object.

     ```
     $sym = new Devel::Symdump('pkg');
     $sym = rnew Devel::Symdump('pkg');#recurse
     ```

  2. Get information from the **Devel::Symdump** object.

     ```
     @symbols  = $sym->scalars();
     print  $sym->as_string(),"\n";
     print FH $sym->as_HTML();
     ```

peektest.pl

```
#use strict;
use Devel::Peek('Dump');
use Devel::Symdump;
### Build a big, horkin' data structure...
my $topref = {
  key1 => [
     [ 'key1Elem1_1', 'key1Elem1_2', 'key1Elem1_3', 'key1Elem1_4' ],
     [ 'key1Elem2_1', 'key1Elem2_2',
       { key1Elem3_3Key1 => 'key1Elem3_3Key1-Value',
           key1Elem3_3Key2 => 'key1Elem3_3Key2-Value',
          key1Elem3_3Key3 =>
            { key1Elem3_3Key3_Key1 =>
              [
                 'key1Elem3_3Key3_Key1-Value1',
                 'key1Elem3_3Key3_Key1-Value2',
                 'key1Elem3_3Key3_Key1-Value3'
              ]
            },
          },
        'key1Elem2_4']
      ],
  key2 => 'toprefKey2-value',
  key3 => 'toprefKey3-value',
  key4sub => sub { my $x = 'hello'; return $x }
};

my $x = "hello";
my $y = \$x;
@pkgB::x_arr = ('pkgB x_arr value 1','pkgB x_arr value 2');
$pkgB::x = "package B x value";
$pkgB::pkgC::z = "pkgC z value";

my $symdumpObj = rnew Devel::Symdump('pkgB');
print $symdumpObj->as_string();
Devel::Peek::Dump($x);
Devel::Peek::Dump($topref);
```

THE DATA::DUMPER MODULE

❈ **Data::Dumper** allows you to dump variable values.

1. Import the **Data::Dumper** module.

```
use Data::Dumper;
```

2. Create a **Data::Dumper** object.

```
$dumpObj = new Data::Dumper(
   [$var1,$var2],['$var1','$var2']);
```

3. Set up any configuration variables.

```
$dumpObj->Quotekeys(0);
```

4. Dump the object.

```
print  $dumpObj->Dump(),"\n";
```

❈ **Data::Dumper**'s output is valid Perl code which can be executed (or **eval**ed) to re-create the dumped data.

❈ A structure with recursive references can't be created in one step.

➢ The **PURITY** flag tells **Data::Dumper** to output additional statements to fully define it when it encounters a self-referencing data structure.

dumptest.pl

```
use Data::Dumper;
### Build a big, horkin' data structure...
my $topref = {
  key1 => [
     [ 'key1Elem1_1', 'key1Elem1_2', 'key1Elem1_3', 'key1Elem1_4' ],
     [ 'key1Elem2_1', 'key1Elem2_2',
       { key1Elem3_3Key1 => 'key1Elem3_3Key1-Value',
           key1Elem3_3Key2 => 'key1Elem3_3Key2-Value',
          key1Elem3_3Key3 =>
            { key1Elem3_3Key3_Key1 =>
              [
                 'key1Elem3_3Key3_Key1-Value1',
                 'key1Elem3_3Key3_Key1-Value2',
                 'key1Elem3_3Key3_Key1-Value3'
              ]
            },
          },
        'key1Elem2_4']
      ],
  key2 => 'toprefKey2-value',
  key3 => 'toprefKey3-value',
  key4sub => sub { my $x = 'hello'; return $x }
};

my $a = "hello";
my $b = \$a;
@pkgB::a_arr = ('pkgB a_arr value 1','pkgB a_arr value 2');
$pkgB::a = "package B a value";
$pkgB::pkgC::c = "pkgC c value";

my $dumper = new Data::Dumper([$topref],['$topref']);
$dumper->Indent(1);
$dumper->Quotekeys(1);
$dumper->Purity(1);
print $dumper->Dump();
```

Labs

❶ Write a program that uses an undeclared variable and one which is used only once. Run your program with the **-w** command-line switch.

Run it again without the **-w** switch, setting the value of **$^W** to **1** inside the program.
(Solution: *warns.pl*)

❷ Copy and modify *warns.pl* to trap the warnings and call your own subroutine to handle the printing of the warning. Be sure to include the original warning text as well as your own.

Hint: Use **$SIG{__WARN__}**
(Solution: *warntrap.pl*)

❸ Write a module with a function that issues warnings with **warn()**.

Write a program that uses the function and examine its output.

Change the **warn** to **carp** and run the program again. Note the differences in line number and filename reporting.

Hint: Don't append a "**\n**" to your warning messages as this disables the output of the filename and line numbers of **warn** and **carp**.
(Solutions: *carps.pl*, *carpmod.pm*)

❹ Write a program which includes the following questionable Perl programming practices:

- An undeclared and unscoped array variable.
- A bareword used as a string in assignments to a scalar.
- A symbolic reference.

Compile and run this program with no warnings.

Once it's working, apply the pragmas **use strict 'vars';**, **use strict 'subs';**, **use strict 'refs';**, and **use strict;** one at a time and in that order.

Compile your program each time and note the kind of warnings and errors you receive.
(Solution: *strictstuff.pl*)

❺ Write a program that declares and initializes a scalar, a simple array, and a simple hash. Create a separate scalar reference to each of these.

Once your program is running, use the **Dump** method of **Devel::Peek** to examine each of these values.

Hint: You might want to redirect the output to a file to look at.
(Solution: *peeker.pl*)

References:
perldoc strict
perldoc perlrun
perldoc perldiag
perldoc perldebug
perldoc Devel::Peek
perldoc Data::Dumper

Resources:
The Perl Journal, Issues #7, #10, #11

CHAPTER 8 - INSTALLING AND USING PERL MODULES

OBJECTIVES

- Find, obtain, install, and use Perl modules.

LAZINESS, IMPATIENCE, AND HUBRIS

- Laziness makes a Perl programmer write labor-saving programs other people will find useful, and document them so as not to have to answer so many questions.
- Impatience makes a Perl programmer write computer programs that anticipate programmers' needs.
- Hubris makes proud Perl programmers write programs they think will withstand the scrutiny of other Perl programmers who use them.
- Perl5 is distributed with a variety of useful library modules.
- In addition, your nearest CPAN site publishes hundreds of other modules that might significantly reduce your application development time.
- There is no need to reinvent the wheel!
  - For most common problems Perlers routinely face, you can find Perl5 modules encapsulating good solutions.
- Among the many advantages of using a published module:
  - It's free.
  - It's (usually) well-documented.
  - It's been tested.
  - Someone else has done the work for you!

Among the scores of modules shipped with Perl5.004_04, you'll find:

▪ **AnyDBM_File**	Provides a framework for multiple DBMs.
▪ **AutoLoader**	Loads subroutines only on demand.
▪ **AutoSplit**	Splits a package for autoloading.
▪ **Benchmark**	Benchmarks running times of code.
▪ **Bundle::CPAN**	Provides a bundle to play with all the other modules on CPAN.
▪ **CGI**	Provides the simple Common Gateway Interface Class.
▪ **CPAN**	Queries, downloads, and builds perl modules from CPAN sites.
▪ **Class::Struct**	Declares **struct**-like datatypes as Perl classes.
▪ **Devel::SelfStubber**	Generates stubs for a **SelfLoading** module.
▪ **DirHandle**	Supplies object methods for directory handles.
▪ **English**	Uses nice English (or **awk**) names for ugly punctuation variables.
▪ **Env**	Imports environment variables.
▪ **Exporter**	Implements default **import** method for modules.
▪ **ExtUtils::Embed**	Provides utilities for embedding Perl in C/C++ applications.
▪ **ExtUtils::MakeMaker**	Creates an extension Makefile.
▪ **File::Compare**	Compares files or filehandles.
▪ **File::Copy**	Copies files or filehandles.
▪ **File::DosGlob**	Provides DOS-like globbing and then some.
▪ **File::Path**	Creates or removes a series of directories.
▪ **FileCache**	Keeps more files open than the system permits.
▪ **FindBin**	Locates the directory of the original perl script.
▪ **Getopt::Long**	Offers extended processing of command-line options.
▪ **I18N::Collate**	Compares 8-bit scalar data according to the current locale.
▪ **IPC::Open2**	Opens a process for both reading and writing.
▪ **Math::BigFloat**	Provides an arbitrary length float math package.
▪ **Math::BigInt**	Provides an arbitrary size integer math package.
▪ **Math::Complex**	Provides complex numbers and associated mathematical functions.
▪ **Math::Trig**	Provides trigonometric functions.
▪ **Net::Ping**	Checks a remote host for reachability.
▪ **Pod::HTML**	Converts pod files to HTML.
▪ **Search::Dict**	Searches for key in dictionary file.
▪ **SelectSaver**	Saves and restores selected file handle.
▪ **SelfLoader**	Loads functions only on demand.
▪ **Shell**	Runs shell commands transparently within perl.
▪ **Sys::Hostname**	Tries every conceivable way to get hostname.
▪ **Term::Cap**	Provides a perl termcap interface.
▪ **Term::Complete**	Provides a perl word completion module.
▪ **Term::ReadLine**	Provides a perl interface to various **readline** packages.
▪ **Test::Harness**	Runs perl standard test scripts with statistics.
▪ **Text::ParseWords**	Parses text into an array of tokens.
▪ **Text::Soundex**	Implements the Soundex Algorithm as described by Knuth.
▪ **Text::Tabs**	Expands and unexpands tabs per the unix **expand(1)** and **unexpand(1)**.
▪ **Text::Wrap**	Provides line wrapping to form simple paragraphs.
▪ **Tie::Hash, Tie::StdHash**	Provides base class definitions for tied hashes.
▪ **Time::Local**	Computes time from local and GMT time.
▪ **UNIVERSAL**	Provides a base class for ALL classes (blessed references).

CPAN

- Comprehensive Perl Archive Network (CPAN) is a central (yet distributed) module repository.
  - Browse it at ***http://www.perl.org/CPAN/*** or via FTP.
- Volunteer CPAN sites around the world maintain mirror copies of all published Perl source code, modules, and scripts.
  - When you visit ***http://www.perl.org/CPAN/***, it will attempt to determine the CPAN site nearest you; it's helpful to memorize the location of your local CPAN site.
- CPAN lists modules by module, category, and author name.
- Authors who submit modules to CPAN provide a standard format and method for installing their modules on your system.
  - These authors, programmers like yourself, submit their modules, accept bug reports and suggestions, and release updated versions.
  - To submit a module to CPAN, find the instructions for PAUSE (Perl Authors Upload SErver).
- Become a frequent visitor to your nearest CPAN mirror site.

The CPAN has many categories of modules. Some of them are:

```
02_Perl_Core_Modules
03_Development_Support
04_Operating_System_Interfaces
05_Networking_Devices_Inter_Process
06_Data_Type_Utilities
07_Database_Interfaces
08_User_Interfaces
09_Interfaces_to_Other_Languages
10_File_Names_Systems_Locking
11_String_Processing_Language_Text_Processing
12_Option_Argument_Parameter_Processing
13_Internationalization_and_Locale
14_Authentication_Security_Encryption
15_World_Wide_Web_HTML_HTTP_CGI
16_Server_and_Daemon_Utilities
17_Archiving_and_Compression
18_Images_Pixmap_Bitmap_Manipulation
19_Mail_and_Usenet_News
20_Control_Flow_Utilities
21_File_Handle_Input_Output
22_Microsoft_Windows_Modules
23_Miscellaneous_Modules
```

USING MODULES

- Instead of putting the **use module** statement in your Perl script, you can use the **-M** command-line option to load a module:

  ```
  perl -MModuleName prog.pl
  ```

- If you can't get your system administrator to install the Perl modules you want, just maintain your own Perl library directory!

- You can specify the library directories in which Perl will search for modules:

  - Modify the **@INC** variable in your script, before calling **use** or **require**.

    ```
    BEGIN {
      unshift(@INC, '/home/francis/my_perl',
                    '/home/jo/perllib');
    }
    ```

  - Or a better alternative is to use the **use lib** pragma in your script:

    ```
    use lib '/home/francis/my_perl';
    use ModuleName;
    ```

  - Use the **-I** command-line switch to push paths onto **@INC**.

    ```
    perl -I/home/francis/my_perl prog.pl
    ```

  - Set the **PERL5LIB** environment variable as a colon-delimited list of directory names.

    ```
    PERL5LIB=/home/francis/my_perl:/home/jo/perllib
    ```

INSTALLING A PERL MODULE

- Installation of a CPAN module is simple:

  1. Unpack the module:

     ```
     gunzip -c ModuleName-1.23.tar|tar xovf -
     ```

  2. Create a **makefile**:

     ```
     perl Makefile.PL
     ```

  3. Execute the **makefile**:

     ```
     make
     ```

  4. Test the module:

     ```
     make test
     ```

  5. Install the module into your Perl directory structure:

     ```
     make install
     ```

- You may run across a module that is just a set of files, with no installation scripts.

  - You'll need to read any *README*s to find out what to do.

  - Most likely you just need to copy a *.pm* file to somewhere where Perl will be able to find it at runtime.

Unpacking the Module Source

- Modules you download from CPAN are usually in *.gzip*/*.tar* format.
  - They are typically named in the style:

    ```
    ModuleName-primaryVersion.subVersion.tar.gz
    ```

- You must unzip the file before going further with the module installation:

  ```
  gunzip -c ModuleName-1.23.tar.gz | tar xovf -
  ```

  - This creates a directory beneath the current directory called ***ModuleName-1.23*** and preserves the original archive file.
  - You may remove this directory, and the *.gz* file, once the installation is complete (not yet, though).
- Don't unpack the module into Perl's standard directory structure.
  - Any temporary directory will do.
- Once you've unpacked the module subdirectory, **cd** into it.
  - This would be a very good time to read any files named *README* or *INSTALL*.

For various reasons, the compressed tar files (sometimes called *tarballs*) that Perl modules are distributed in are usually compressed with *gzip* (GNU zip), not the standard UNIX **compress** command. If your system doesn't have *gzip*/*gunzip*, talk to your system administrator (or download and build it yourself, it's not too tough to do).

```
gunzip -c tarfile.gz
```

leaves the compressed file in place, writing the unzipped data to standard output suitable for input to:

```
| tar xovf -
```

(**x** for eXtract, **o** for assigning your file Ownership to the extracted files, **v** for Verbose, **f -** to read tar archive data from standard input.)

In two steps, this would be:

```
gunzip tarfile.gz
tar xovf tarfile
```

Occasionally, an author will use the extension *.tgz* instead of *.tar.gz*.

THE CONFIGURATION STEP

- Create the **makefile**:

  ```
  perl Makefile.PL
  ```

- *Makefile.PL* is a standard Perl script which has been customized by the author of the module.

  - It checks to see that the module source is complete.
  - It may be interactive.
  - It may also detect the absence of prerequisite modules; if it does, make sure to install those modules first, then start over with this module.

- *Makefile.PL* generates **Makefile**, a standard UNIX **makefile**.

- To configure the module to install itself in your own library directory, use the **PREFIX** option:

  ```
  perl Makefile.PL PREFIX=/home/francis/my_perl
  ```

  - This will replicate the standard Perl library directory structure in a location of your choice.
  - To use modules installed in such a location, you'll have to make sure this directory is known to your scripts.

Module authors use the standard Perl utility script *h2xs* and the standard module **ExtUtils::MakeMaker** to create *Makefile.PL*.

Makefile.PL is a Perl script whose job is to generate a **Makefile** for use with the ubiquitous **make** command. *Makefile.PL* uses information from your local Perl installation to create a **Makefile** customized just for your system.

The Build Step

- Run the **makefile**.

  ```
  make
  ```

- This step compiles any C code necessary for the module.

- You may want to keep the output and any error messages in a log for later perusal.

  ```
  make 2>&1 | tee make_log.1
  ```

  - Most CPAN modules compile cleanly "out-of-the-box."

- If there are any compilation failures, you'll need to solve them before going on.

  - Check the *README* files to see if there are any special options or modifications you can use in *Makefile.PL*.

  - Look for a *README* file specific to your system — *README.win32*, *README.AIX*, *README.VMS*, etc.

  - You shouldn't modify **Makefile** directly; instead:

    - Pass parameters to *perl Makefile.PL*.

    - Modify *Makefile.PL*.

This step creates a directory named *blib/* ("build library") under the module source directory. All module files are built or copied into this subdirectory for testing.

THE TEST STEP

- Once you've built the module, run any tests provided by the author:

  ```
  make test
  ```

- This command either:
  - Runs the script *test.pl* in the current directory.
  - Runs all scripts it finds under a subdirectory named *t/*, if they have the filename extension *.t*.
- You may want to keep the output and any error messages in a log file for later perusal.

  ```
  make test 2>&1 | tee make_log.test.1
  ```

- Be sure that all the tests succeeded before continuing.
  - If they didn't, read the log and *README* files to see if you can determine why.
    - If you can't, contact the author of the module.
  - It may not be necessary for every test to succeed before you can use the module.

This uses the module files under *blib/* to run the test scripts provided by the module's author.

THE INSTALL STEP

- ❈ You usually need to be logged on with system administrator (**root**) privileges to install a module into the existing Perl directory structure.

```
make install
```

  - ➢ When installing into your own library directory, watch for errors; you may need to create a directory or two, then rerun **make install**.

- ❈ When installation is finished, you can remove the module installation directory and the archive file.

  - ➢ If you want to keep the module source code around, to study or to build again later with different options, you can save space by running **make clean** to remove temporary files.

The **make install** step copies the tested module files out of the *blib/* directory structure into their final destination — either the standard Perl installation directory, or the location you specified with the **PREFIX** option:

```
perl  Makefile.PL  PREFIX=/my/own/perl/lib.
```

USING CPAN.PM

- The latest versions of Perl5 ship with a module called **CPAN**, which automates the module download and installation procedure.

```
perl -MCPAN -e shell
```

- When you first use the *CPAN.pm* module, it will ask you many configuration questions.
  - You'll enter the full URL of your local CPAN site, as in:

    ftp://ftp.cs.colorado.edu/pub/perl/CPAN
  - It will create a *.cpan/* directory under your home directory.
  - The *MyConfig.pm* file under *.cpan/CPAN/* contains your configuration information.
    - You can change this file to reflect new preferences.
    - Or use the **o conf** command to set and query options.
- With *CPAN.pm* you can:
  - Search for authors, bundles, distribution files, and modules.
  - Display the *README* of the distribution file.
  - Make, test, install, and clean modules or distributions.
    - You can determine if you have the latest version.
- Type "**h**" at the **cpan>** prompt for a list of commands.

```
cpan> h

command arguments       description
a    string             authors
b    or         display bundles
d    /regex/       info    distributions
m    or         about   modules
i    none               anything of above

r    as         reinstall  recommendations
u    above         uninstalled distributions
See manpage for autobundle, recompile, force, look, etc.

make               make
test    modules,        make test (implies make)
install dists, bundles,    make install (implies test)
clean   "r" or "u"      make clean
readme             display the README file

reload  index|cpan      load most recent indices/CPAN.pm
h or ?             display this menu
o    various       set and query options
!    perl-code       eval a perl command
q               quit the shell subroutine

cpan> m /odbc/
Module     DBD::ODBC     (TIMB/DBD-ODBC-0.20.tar.gz)
Module     RDBAL::Layer::ODBC  (B/BR/BRIAN/RDBAL-1.15.tar.gz)
Module     Win32::ODBC   (Contact Author DAVEROTH (Dave Roth))
Module     html       (BJEPS/iodbc-wwwtools-0.20b.zip)
Module     iodbc      (JMAHAN/iodbc_ext_0_1.tar.gz)
Module     wwwodbc    (BJEPS/iodbc-wwwtools-0.20b.zip)

cpan> o conf
CPAN::Config options:
   commit          Commit changes to disk
   defaults           Reload defaults from disk
   init            Interactive setting of all options
   build_cache        10
   build_dir          /home/francis/.cpan/build
   cpan_home          /home/francis/.cpan
   ftp             /usr/bin/ftp
   ...
```

USING MODULE DOCUMENTATION

- Most of the modules on CPAN have an associated *README*, embedded ***perldoc*** documentation, or manpage.
    - If there is a *README*, by all means read it.
- Use the **perldoc** command to read the embedded POD (Plain Old Documentation).

```
perldoc ModuleName
```

- Use **pod2html** to generate HTML documentation from the ***perldoc*** comments embedded in a module.

```
pod2html ModuleName.pm > ModuleName.html
```

- If the documentation provided with the module doesn't provide you with enough information to use the module, look at the test programs provided with the module for hints on how to use it.

Labs

❶ Create a directory, in your home directory, named *perl5lib/*. You'll use this directory for modules you install for personal use, so set up your environment so Perl will search your *perl5lib/* directory for modules at runtime.

❷ The following modules have been provided for you in this chapter's directory:

Lingua-EN-Inflect-1.84.tar.gz
Games-WordFind-0.02.tar.gz
Roman-1.1.tar.gz

Unpack these archives, read the *READMEs*, then take the appropriate steps to install these modules in your own Perl module directory.

❸ Once you've installed the modules, create HTML documentation for them.

❹ Based on the documentation, choose one of the modules and write a short program to use it.

References:
perldoc perlmodlib
perldoc CPAN

Resources:
http://www.perl.com/CPAN
http://www.perl.com/CPAN/modules
http://www.perl.org/news.html
news:comp.lang.perl.modules

Those* who maintain the integrity of Perl distributions struggle to keep it compact and concise. CPAN has many modules that most people find useful, but including them all would bloat the *perl.tar.gz* file beyond its current 2-3Mb. We may soon see a prepackaged "Perl Developers Kit" containing the most stable versions of the most useful modules.

*At any given time, only one person on Earth is ordained to apply patches to Perl and to cut releases of the Perl source distribution from the master source code tree. For obscure reasons, this person is said to be "holding the patch pumpkin," and is referred to as the "Pumpking." For years it was Larry Wall, but now the honor, or burden, rotates.