Perl Programming on NT

Student Workbook
Perl Programming on NT

Bill Parrette

Published by itcourseware, Inc., 7245 South Havana Street, Suite 100, Englewood, CO 80112

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

Editors: Jeff Howell, Rob Roselius, Rick Sussenbach

Editorial Assistant: Jan Waleri

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Jimmy Ball, Tait Cyrus, Phil Gardner, Denise Geller, Roger Jones, Roger Kobylarczyk, Michael Naseef, Bill Parrette, Richard Raab, and George Trujillo and the many students who have offered comments, suggestions, criticisms, and insights.

Copyright © 1996-2001 by itcourseware, Inc. All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to itcourseware, Inc., 7245 South Havana Street, Suite 100, Englewood, CO 80112. (303) 302-5280.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.
# Contents

Chapter 1 - Course Introduction ........................................................................................................ 9
  Course Objectives .................................................................................................................. 11
  Course Overview ................................................................................................................. 13
  Suggested References ........................................................................................................... 15

Chapter 2 - Overview of Perl ......................................................................................................... 17
  Chapter Objectives .............................................................................................................. 19
  What is Perl? ......................................................................................................................... 21
  Running Perl Programs ....................................................................................................... 23
  Perl Scripts as Executable Programs ................................................................................. 25
  Sample Program ..................................................................................................................... 27
  Another Sample Program ...................................................................................................... 29
  Yet Another Example ............................................................................................................. 31
  Labs ...................................................................................................................................... 33

Chapter 3 - Perl Variables ........................................................................................................... 35
  Chapter Objectives .............................................................................................................. 37
  Three Types of Variables ................................................................................................. 39
  Variable Names and Syntax ............................................................................................... 41
  Variable Naming ................................................................................................................... 43
  Lists ....................................................................................................................................... 45
  Scalar and List Contexts ..................................................................................................... 47
  The Repetition Operator ..................................................................................................... 49
  Labs ...................................................................................................................................... 51

Chapter 4 - Arrays and Hashes ................................................................................................... 53
  Chapter Objectives .............................................................................................................. 55
  Arrays ................................................................................................................................... 57
  Example - The @ARGV Array .............................................................................................. 59
  Array Functions ................................................................................................................... 61
  Array Slices ........................................................................................................................... 63

© 1996-2001 by itcourseware, Inc.  3/01
Chapter 8 - Regular Expressions ................................................................. 131
  Chapter Objectives ................................................................................ 133
  Pattern Matching Overview ................................................................. 135
  The Substitution Operator .................................................................. 137
  Regular Expressions .......................................................................... 139
  Special Characters .............................................................................. 141
  Quantifiers (*, +, ?, {}) ................................................................. 143
  Assertions (^, $, \b, \B) ................................................................. 145
  Labs .............................................................................................. 147

Chapter 9 - Subroutines .......................................................................... 149
  Chapter Objectives ............................................................................ 151
  Overview of Subroutines .................................................................. 153
  Passing Arguments ........................................................................... 155
  Local Variables ................................................................................ 157
  Passing Names ................................................................................ 159
  Returning Values ............................................................................ 161
  Labs .............................................................................................. 163

Chapter 10 - Quoting and Interpolation ..................................................... 165
  Chapter Objectives ............................................................................ 167
  String Literals .................................................................................. 169
  Interpolation ................................................................................... 171
  Array Substitution .......................................................................... 173
  Backslashes and Single Quotes ....................................................... 175
  Command Substitution .................................................................. 177
  Here Documents ............................................................................. 179
  Labs .............................................................................................. 181

Chapter 11 - References .......................................................................... 183
  Chapter Objectives ............................................................................ 185
  References ....................................................................................... 187
  Creating References ....................................................................... 189
  Using References ............................................................................ 191
  Passing References as Arguments to Subroutines ......................... 193

© 1996-2001 by itcourseware, Inc. 3/01
Passing References as Arguments to Subroutines (continued) ........................................... 195
Anonymous Composers ........................................................................................................... 197
Hard References as Hash Keys ............................................................................................... 199
The Symbol Table .................................................................................................................. 201
Labs ......................................................................................................................................... 203

Chapter 12 - Complex Data Structures ................................................................................ 205

Chapter Objectives ............................................................................................................... 207
Two-dimensional Arrays in Perl ............................................................................................ 209
Anonymous Arrays and Anonymous Hashes ........................................................................ 211
Arrays of Arrays .................................................................................................................... 213
Arrays of References ............................................................................................................ 215
A Hash of Arrays .................................................................................................................. 217
A Hash of Hashes .................................................................................................................. 219
And So On . . . ....................................................................................................................... 221
Labs ......................................................................................................................................... 223

Chapter 13 - Packages and Modules .................................................................................... 225

Chapter Objectives ............................................................................................................... 227
Packages ................................................................................................................................. 229
BEGIN and END Routines .................................................................................................... 231
require vs. use ...................................................................................................................... 233
Modules .................................................................................................................................. 235
The bless Function ............................................................................................................... 237
Labs ......................................................................................................................................... 239

Chapter 14 - Introduction to Object-Oriented Programming in Perl .................................... 241

Chapter Objectives ............................................................................................................... 243
What is Object-Oriented? ..................................................................................................... 245
Why Use Object-Oriented Programming? ............................................................................ 247
Classes, Objects, and Methods in Perl ................................................................................. 249
Inheritance, the "is-a" Relationship ....................................................................................... 251
Containment, the "has-a" Relationship .................................................................................. 253
Overloaded Operators .......................................................................................................... 255
Destructors ............................................................................................................................. 257
Labs ......................................................................................................................................... 259
Chapter 15 - Advanced Regular Expressions ................................................................. 261
  Chapter Objectives .................................................................................................... 263
  Substrings .................................................................................................................. 265
  Substrings in List Context ........................................................................................ 267
  RE Special Variables ................................................................................................. 269
  RE Options .................................................................................................................. 271
  Multi-line REs ............................................................................................................ 273
  Substituting with an Expression .................................................................................... 275
  Perl5 RE Extensions .................................................................................................... 277
  Labs ............................................................................................................................ 279

Chapter 16 - Binary Data Structures ............................................................................... 281
  Chapter Objectives .................................................................................................... 283
  Variable-Length (Delimited) Fields ............................................................................. 285
  Variable vs. Fixed ....................................................................................................... 287
  Handling Binary Data .................................................................................................. 289
  The pack() Function ................................................................................................. 291
  The unpack() Function ............................................................................................. 293
  The read() Function ................................................................................................ 295
  C Data Structures ...................................................................................................... 297
  Labs ............................................................................................................................ 299

Chapter 17 - Example Perl Scripts .................................................................................. 301
  Chapter Objectives .................................................................................................... 303
  Introduction to Examples ............................................................................................ 305
  displayNetInfo.pl ...................................................................................................... 307
  displayNetInfo.pl (cont'd) .......................................................................................... 309
  perlGrep.pl .................................................................................................................. 311
  perlGrep.pl (cont'd) ................................................................................................... 313
  perlGrep.pl (cont'd) ................................................................................................... 315
  perlMetrics.pl ............................................................................................................. 317
  perlMetrics.pl (cont'd) ............................................................................................... 319
  perlMetrics.pl (cont'd) ............................................................................................... 321
  perlMetrics.pl (cont'd) ............................................................................................... 323
  perlMetrics.pl (cont'd) ............................................................................................... 325
Chapter 1 - Course Introduction
Notes

This course is intended for experienced programmers with user level skills in the Windows NT environment. Many programs will be written during the class. The lecture topics and lab exercises concentrate on the Perl language and its features, with less emphasis on application-specific subjects.

The particular applications that you will design, write and work on during this class are intended to demonstrate the use of Perl's capabilities and services provided in Windows NT programming environments.
Course Objectives

- Write scripts, programs, and reusable modules in Perl.
- Make effective use of Perl modules written by others.
Notes
Course Overview

- **Audience:** This is a programming course designed for software development professionals. You will write many programs in this class.

- **Prerequisites:** User level skills in the NT environment are required. The ability to write programs in a high level structured language (such as C) is recommended.

- **Student Materials:**
  - Student workbook

- **Classroom Environment:**
  - NT 4.0 or Windows 2000 workstation per student with a Perl development environment, such as ActivePerl 5.6 or later.
Suggested References


http://www.activestate.com
http://www.cpan.org
http://www.perl.com
Chapter 2 - Overview of Perl
Notes
Chapter Objectives

- Define what Perl is.
- Run Perl programs various ways.
- Get a feel for Perl programming by reviewing example Perl programs.
Notes

There is a lot of confusion about whether Perl is a compiled or interpreted language. Most interpreted languages perform poorly due to the time required to interpret and then run each line of a program in turn, as in batch files. Perl runs interpretively; the programs you run are written and run as source Perl code. Perl also has the capability to modify itself programmatically as it runs. Perl runs much faster than most other interpreters, however, due to its capability to pre-compile and tokenize major portions of a program before it actually runs the script.

Since much of the functionality of Perl maps directly onto C function calls, it makes sense for Perl to simply call the C library routines as it needs them. This compilation of Perl code dramatically speeds up Perl programs, making them almost as fast as truly compiled C programs.

There is not, however, a Perl compiler which can create an actual binary executable image (file) which can then be loaded and run. Perl programs always run under the perl executable on your host computer. There’s no way around that.

This means that while Perl programs are fast to execute and very fast to develop (in comparison to C, for example), always remember that when you distribute Perl programs, you are distributing clear text source code which the user can read, modify and easily break.
What is Perl?

- Perl is an interpreted programming language.

- Its strengths are:
  - Scanning and processing large amounts of text efficiently.
  - Formatting and printing reports easily.
  - Providing tremendous built-in parsing functionality to reduce the need for ancillary code in a program.
  - Great extensibility via libraries and modules.

- Perl contains the following:
  - Most of the statements and operators of C.
  - Many of the library functions provided in a C environment.
  - Automatic record input and field splitting.
  - Extended regular expressions found in UNIX, with Perl extensions.
  - The file manipulation capabilities of batch files.
Notes

Several ways to run Perl programs:

1. For short (one-line) programs, you can use the `-e` command line option to write and execute the program on the command line:

   ```
   C:> perl -e "foreach (@ARGV) {print qq/$_ \n/ if /^[A-Z]/;}"
   The quick red fox.
   ```

2. Use an editor such as Notepad and type the program into a file, such as `prog.pl`. Then type the following command at the prompt (the prompt is a `C:`):

   ```
   C:> perl prog.pl The quick red fox.
   ```

   The one-liner above might look like this in `prog.pl`

   ```perl
   foreach (@ARGV) {
      print "$_ \n" if /^[A-Z]/;
   }
   ```

3. Use an editor to type a Perl program into a file, but insert a first line into the file so it looks like this:

   ```perl
   #!perl
   foreach (@ARGV) {
      print "$_ \n" if /^[A-Z]/;
   }
   ```
Running Perl Programs

- Perl programs can be run several ways.
- The simplest way is to put the program in a file, say prog.pl, and type:
  
  C:> perl prog.pl

- This will invoke the Perl interpreter, perl, which will interpret the program in the file prog.pl.
  
  □ The .pl extension is not required; it's just a convention that some Perl programmers follow.

- Some other ways to run Perl programs are described on the facing page.
Notes

How can I run my Perl scripts from the DOS command Line?

If you are using the ActiveState implementation of Perl, the Perl interpreter will modify your registry appropriately so that you can run your scripts directly from the DOS command line without having to specify the perl interpreter yourself. If you are using some other Perl implementation, you will have to enter the data in the registry yourself using regedit — but get someone “in the know” before you try to do it.

In addition, in NT version 4 and higher, you can also run your Perl script without an extension. This is done by setting a new environment variable:

```bash
SET PATHEXT=.pl;%PATHEXT%
```

This variable — like the PATH variable — tells the system which files are allowed to be run as commands from the command line. So, if you have a script called `myprog.pl`, and if the PATHEXT variable is set as above; you can execute the script by simply typing `myprog`.

On the other hand, you could also use a utility called `pl2bat` on your Perl script to create a DOS batchfile from it:

```bash
C:> pl2bat myprog.pl
```

This will create a DOS batch file that contains the code of your Perl script called `myprog.bat`. As you know, when you want to execute a DOS batch file, you do not need to type the `.bat` extension. So, you could run the resulting `myprog.bat` more simply as `myprog.pl2bat` is a Utility that was developed by the Perl community specifically for Microsoft versions of the interpreter and should be in all implementations. But, it can also be downloaded for free at many Internet Web sites.
Perl Scripts as Executable Programs

- To run a Perl script without specifying the Perl interpreter, you must modify your system’s registry.

- The registry (among other things) associates a filename extension with a particular executable program file. i.e., .pl -> perl.

- With ActiveState Perl, you do nothing: the system modifies your registry appropriately.

  □ If you aren’t using ActiveState’s version, then you may have to modify the registry yourself with regedit.

    - Or you can associate .pl files with the Perl interpreter in Explorer by selecting the file, right-clicking and selecting "Open With . . ."

- Using ActiveState’s implementation modifying the registry for you, all you need to do to run your script is to type its name as a command.

  C:> myprog.pl

- An additional feature for command line execution lets you run the script without the extension just like you do for .exe, .bat and similar commands.

  C:> myprog

- You can use the new PATHEXT environment variable or you can wrap a DOS batch file around your Perl script.
Notes

Consider the following Perl program:

```perl
#!/perl
foreach (@ARGV) {  
    print "$_\n" if /^[A-Z]/;
}
```

The program is intended to be run with command line arguments that are words. For each word that starts with an uppercase letter, the word is printed.

1. The second line begins a `foreach` loop that comprises the entire program. For each iteration of the loop, the special variable `$_` will be set (as a string) to the next command line argument. The command line arguments are stored in the special array called `@ARGV`.

2. The loop consists of a single statement, the third line of the program. The statement is a conditional, with the `if` syntactically following a call to the Perl `print()` function. The `print()` function, if called, will print out the word.

Under what condition will the `print()` function be called? Well, the pseudo-code for the entire program looks like this:

```
for each command line argument:
    set the variable $_ to the string value of the argument
    if the word named in $_ starts with an uppercase letter
       call the print() function to display the word on the screen.
```

Or alternatively:

```
for each command line argument:
    set the variable $_ to the string value of the argument
    call the print() function to print the word IF the word starts with an uppercase letter
```

A more verbose version of this program follows. It takes less advantage of Perl's automatic actions and is more explicit:

```perl
#!/perl
foreach $file (@ARGV) {
    if($file =~ /^/[A-Z]/) {
        print "$file\n";
    }
}
```

The second line uses `foreach`; it specifies the variable name that the `foreach` loop is to assign the argument strings to, instead of implicitly assigning them to the special variable `$_`. The variable name is `$file`. The dollar sign in the `$file` variable name is part of the variable name. It is not a dereference operator.

The test for uppercase letters on the third line uses UNIX-like regular expressions. If the `/reg-exp/` operator is not given an argument variable containing a string name, it will test the contents of the `$_` variable.
Sample Program

#!/perl
# File: prog.pl

foreach (@ARGV)
{
    print "$_ \n" if /^[A-Z]/;
}
Notes

This program, *menu.pl*, displays a menu of user-selectable actions. The program executes an infinite loop using the C-style `for(;;)` construct, waiting for the user to type a choice. After performing the action chosen by the user, the program clears the screen and redisplay the menu. The program terminates when the user selects the exit menu choice.

The `system("cls")` function tells the operating system to execute the `cls` command, just as if it had been typed on the keyboard.

The line:

```
$choice = <STDIN>; chop $choice;
```

reads one line into the variable `$choice` from the file identified by the file handle `STDIN`, using the input operator `< >`. The `chop` function chops off the newline character that was read from `STDIN` and stored into `$choice`.

The line:

```
last if $choice eq "e" ;
```

exits the `for` loop using the Perl control flow statement `last` if the user chose `e`. If we had not chopped the newline we would have said:

```
last if $choice eq "e\n" ;
```

After setting the variable `$command` based on `$choice`, the `system()` function is called once again to execute the command.

Finally, the line:

```
<STDIN>;
```

simply reads a line from the standard input. The intent is to wait for the user to hit the `<Enter>` key before clearing the screen and redisplaying the menu. The stand-alone `<STDIN>` actually reads one line and discards it.

Note that a semicolon, `;`, must terminate each simple statement. A compound statement (a sequence of simple statements, enclosed in braces) is not terminated with a semicolon.
Another Sample Program

#!perl
#File: menu.pl

for (; ;) {
    system("cls");  # Clear the screen
    print "\n\n"
    print "  l. List the File Names\n"
    print "  d. Print the Date\n"
    print "  t. Print the Time\n"
    print "  e. Exit Menu\n"
    print "\n\n"
    print "  Enter choice: ";
    $choice = <STDIN>; chop $choice;
    last if $choice eq "e" ;

    if     ($choice eq "l") { $command = "dir/w";  }
    elsif ($choice eq "d") { $command = "date/t";  }
    elsif ($choice eq "t") { $command = "time/t";  }
    else { next;  }

    print "\n\n"
    system($command);
    print "\n\n"

    print 'Hit <Enter> to continue:';

    <STDIN>;  # Wait for the enter key
}
The `@ARGV` array holds the command line arguments. The `foreach` loop assigns the value of the arguments to `$word`, one argument for each iteration of the loop. Note that any quotes or wildcard characters are part of the argument in Windows NT.

The line:

```
open(PIPE, "dir /b |") or die "Pipe open failed" ;
```

assigns the PIPE filehandle to be connected to the output of the `dir` command and each time that the input operator `<>` operates on the filehandle, information is read from the pipe.

The line:

```
@files = <PIPE>;
```

reads the entire output of the `dir` command into the array `@files`, with one line of output assigned to each element of the array. If the line had said:

```
$files = <PIPE>;
```

then only one line would have been read from the pipe into the scalar variable `$files`. This is because of the difference between an array context and a scalar context which we will discuss later in great detail.

The next `foreach` loop will assign the array elements of `@files` to the scalar variable `$filename`, one element at a time. This is doing exactly what the outer loop is doing with the `@ARGV` array. The `$filename` variable will thus be set equal to a pathname (one line of output from the `dir` command) for each iteration of the loop.

The `chop` operator (aka `chop` function) chops off the last character of the string in `$filename`, which will be a newline character, and then the `-T` test checks to see if the file named in `$filename` is a text file. If so, then a file handle named `THIS` is created, the file is opened and `THIS` is associated with the open file.

Next the `while` loop sets the value of the scalar variable `$line` equal to each line of the open file by using the input operator `<>` on `THIS`, which reads a line from the file each time it is executed. Note that `$line` is a scalar, so the context of the `<THIS>` operation is a scalar context.

The line:

```
if ($line =~ /$word/)  
```

checks the `$line` variable for the occurrence of the pattern stored in `$word`, which is the current command line argument. The operator `=~` is a pattern bind operator, and the the `/$word/` is a regular expression.

Finally, note that in the `print` statement, even though the `$filename` and `$line` variables are enclosed in double quotes, their values are still used. Perl calls this interpolation.

A few words on filehandles. In Perl, a file is opened and associated with a filehandle by calling the `open` function. After a file is opened, all file operations on the file are done by referencing the filehandle. For example:

```
open (BOOKFILE, "books");  # Open file "books", associate with file handle BOOKFILE
$book = <BOOKFILE>;         # Use input operator <> to read a line

close(BOOKFILE);
```
Yet Another Example

#!perl
# File: dgrep.pl

# A program to:
#   1) open an input pipe
#   2) find all files in the current directory
#   3) print the names of text files
#   4) that contain words
#   5) specified on the command line

open(PIPE, "dir /b") or die "Pipe open failed\n";
@files = <PIPE>;
close(PIPE);
chop @files;

foreach $word (@ARGV)
{
    foreach $filename (@files)
    {
        if ( -T $filename )
        {
            open(THIS, $filename) || die "File open failed\n";
            while ($line = <THIS>)
            {
                if ($line =~ /$word/)
                {
                    print "$filename: $line\n";
                }
            }
            close(THIS);
        }
    }
}
Labs

1. What is the significance of the special variable \$_?

2. Write a program to echo its command line arguments. (Solution: args.pl)

3. What would happen in the program dgrep.pl if we used the automatically set special variable \$_ in each of the foreach loops instead of the explicit variables $word and $filename?

4. Modify the program menu.pl so that it will prompt the user for a command and then execute as a command whatever line the user types in. (Solution: cmd.pl)

5. Write a program that opens a file and loops to read each line from the file into a variable and prints the variable to the screen, prepending each line with its line number. (Solution: liner.pl)
Notes
Chapter 9 - Subroutines
Notes
Chapter Objectives

- Call subroutines in a number of different ways.
- Pass in arguments to a subroutine.
- Control scoping and lifetime of variables.
- Return data from subroutines.
Example:

```perl
#!perl
# File: ret.pl
$return = 0;
$return = &sub_one;
print "main: $return\n";

sub sub_one {
  98 + 1;
  # or: return 99;
}
```

Example:

```perl
#!perl
# File: global.pl
$avar=5;
&sub_two;
print "main: $avar\n";

sub sub_two {
  print "sub_two: $avar\n";
  $avar = 6;
  print "sub_two: $avar\n";
}
```
Overview of Subroutines

- A subroutine is declared as follows:

```perl
sub NAME BLOCK
```

- The declaration can occur anywhere in a script.

- Subroutines can be called with either the `&` or `do` operators:

```perl
&SUBROUTINE [ ( [LIST] ) ];
do SUBROUTINE ( [LIST] ); # 'do' is frowned # upon now.
```

- The return value of a subroutine is the value of the last statement executed in the subroutine `BLOCK`.

- The `return` statement can be used to explicitly return a value and return from the subroutine.

- Return values can be either scalar values or array values.

- Variables in a subroutine are not local unless made so.

- In Perl5, the `&` and the parentheses are optional if the subroutine has been pre-declared (if Perl knows that a subroutine of that name exists).

- This allows subroutines to look exactly like functions or operators.
Notes

There is a possible problem if you use an array to off-load @_ inside the subroutine. Consider the following example:

```perl
sub foo {
    (@x, @y) = @_;  
    return @y;
}
```

We intend to be able to call the subroutine as follows:

```perl
print (&foo(@array1, @array2));
```

We intend to have the subroutine return the second array, but it doesn't. The problem is that inside the subroutine, @x doesn't know when to stop getting scalars from @_, so it soaks up all of them! We have to add an additional argument at the front of the subroutine to pass the actual size of the first array passed in:

```perl
sub foo {
    ($size, @y) = @_;  
    @x = splice(@y, 0, $size);  
    return @y;
}
```

Shortly, we will see a better way to accomplish this.
Passing Arguments

- Arguments are passed to a subroutine as a list.
  
  &open_all('file1', 'file2', "$other");

- In the subroutine, the arguments are available in the array @_.

  $sum = &addem(1,2,3);
  print "$sum\n";
  print &addem(1,2,3), "\n";

  sub addem {
      ($first, $second, $third) = @_;  
      return $first + $second + $third;
  }

- The array @_ is local, but its values are modifiable references to the arguments.

  $v = 0;
  &foo($v);
  print "$v\n";  # Prints 1

  sub foo {
      $_[0] = 1;  # 0th element of @_ refers to $v
  }
Subroutine Variables

To modify the value of a global scalar or array in a subroutine refer to it by its name without a local declaration.

To modify the value of a passed scalar or array in a subroutine refer to it through @_, as in
$_[0]=99$.

To make a variable local and prevent a possible global name conflict, use the local() operator:

```
sub thing
{
    # Provide a meaningful name, BUT might trash global
    ($possibly_global) = @_;  

    # Get meaningful argument names, AND make them local
    local($passed_in1, $passed_in2) = @_;  

    # Just create a local variable to avoid global name conflict
    local($truly_local);
    ...
}
```

To modify the structure of a global array (i.e., push, pop, or change its size in any way), pass the *name of the array.

The my() function in Perl5 allows you to create a "local" variable which is both truly local to that subroutine, block, or eval, and is completely hidden from the outside world. It is considered safer than a local() as the scope extends only to the subroutine itself, not to other subroutines called from it (a local() variable will be accessible to a called subroutine).
Local Variables

- To create local variables in a subroutine, use the `local` operator.

```
local(VARS);
local(VARS) = (LIST);
```

Example:

```perl
#!perl
# File: passs.pl
$y = 1;
$x = 5;
&sub_one($x);
print "main a: $x\n";
print "main b: $y\n";

sub sub_one {
    #local($y) = @_; # Uncomment NOT to nail y.
    ($y) = @_; # Uncomment to nail y.
    $y = 6;
    #$_[0] = 99;  # Uncomment to nail x.
}
```

- To create super local variables in a subroutine, use the `my` operator.

```
my(VARS);
my(VARS) = (LIST);
```
Notes

Does not refer to @a, only to elements of @a:

arrayEx.pl

#!/perl
# Modifying all array elements in a subroutine
@abc = (1,2,3);
&s1(@abc);
print "main abc: @abc\n";

sub s1 {
    foreach $elem (@_) {
        $elem *= 2;
    }
}

Refers to @abc:

abcex.pl

#!/perl
@abc = (1,2,3);
&s2(*abc);                 # NOTE *abc, not @abc
print "main abc: @abc\n";

sub s2 {
    local(*arr) = @_;
    foreach $elem (@arr) {
        $elem *= 2;
    }
}
Passing Names

- The @_ array contains references to scalar or array arguments.

  - This allows you to modify the values of scalars or array elements that are passed in.

- But if you wish to modify the size of an array, you must pass its name using the * operator.

- *NAME* is a structure (called a typeglob) containing actual symbol table pointers of all data types named NAME: scalars, arrays, filehandles, subroutines, etc.

- Here’s how to use it:

```perl
#!/perl
# File: names.pl

@t = (1,2,3);
&thing(*t);   # Pass everything named 't'
print "main t: @t\n";

sub thing {
    # Catch everything named 't'
    local(*v) = @_;

    # Refer to the array named 't'
    unshift(@v, 'P');  # Modifies $#t
}
```
The expression returning the return value will be evaluated in a scalar context if the subroutine was called in an scalar context, and in an array context if called in an array context.
Returning Values

- Return values can be either scalar values or array values.

```perl
#!/perl
# File: subby.pl

$num = &subby;
@num = &subby;
print "$num : @num\n";

sub subby {
    local(@array) = ('book', 'mag', 'news');

    # What will be returned? Why?
    return @array;
}
```

- Perl returns the array as a list, which creates the problem of a list in a scalar context.
  
  - This may return the first or last element, or the number of elements in the list.

- To avoid this kind of problem, a subroutine can use the `wantarray` operator to determine the context of the caller and return an appropriate value.
Notes
Labs

1. What happens if a reference to a non-local variable in a subroutine is the first reference to that variable in the entire program? (Solution: vref.pl)

2. Write a program that calls a subroutine to open a file where the file name is passed as an argument to the subroutine, and the filehandle is the same name as the file name. Return the filehandle to the main program and verify a successful read from the file in the main. (Solution: fname.pl)

3. Write a modular program with no global variables that:
   a) calls one subroutine to open the files named in the file control,
   b) calls a second to read all the numbers from each file, and calculate several sums: one per file, and total,
   c) calls a third to display a table of the results.

(Solution: sums.pl)
Notes
Chapter 14 - Introduction to Object-Oriented Programming in Perl
Notes
Chapter Objectives

- Describe the benefits of Object-Oriented programming.
- Use the Perl5 syntax for implementing Object-Oriented programming.
- Design an Object-Oriented solution in Perl5 implementing use, inheritance, data encapsulation and operator overloading.
Notes

What would the objects in a Name & Address Book application be? Let's name some:

A **person** is an object. You might describe a **person** as a group of other objects.

A **person** has a **name**, one or more **addresses**, one or more **phone numbers**, a SSN, physical **description**, etc.

A **name** is an object. It has last, first, and middle components. An **address** is an object. It has a street name and number, apt/suite, city, state, and zip.

A **phone number** is an object. It has an area code, prefix, and suffix. It may also include an extension or PIN number, etc.

A SSN is just a number, so would probably not be an object.

A **description** is an object. It may have height, weight, eye color, hair color, etc.
What is Object-Oriented?

- Object-Oriented Analysis and Design (OOA&D) is the process of designing and implementing a solution from a data perspective rather than a process flow (procedural) perspective.

- Think in terms of “What am I manipulating?” rather than “What do I need to do?”

- Objects represent things, such as a radio.
  - Object data (or attributes or fields) are what describe an object such as volume, station, etc.
  - Object methods are what you can do with or to an object, such as turn it on/off, adjust volume, select station, etc.

- A class is the formal description of an object and includes its name, its attributes (data), and what you can do with/to it (methods).
  - A class is not a thing, but the description of a thing.
    - For example, you might describe your radio to a friend by telling them the brand name and all the nifty features it has, but the object (radio), is the thing you ask your friend to help you install in your car.

- Object-Oriented Programming (OOP) is the process of implementing the Object-Oriented (OO) paradigm in a programming language (Perl, in our case) to solve a particular programming task.
Notes

Many applications within your company may require a name object, an address object and all
the rest mentioned earlier. If care has been taken to make these classes general in nature,
downstream development based on them can be much, much, faster and maintenance should
be much easier.

Be sure to publish your classes, document them and get them into a class library so that
others may use them. Providing a repository of these classes/ modules is a basic must in
effectively implementing an OO solution within your company.

Remember!

Reusability is a major thing we’re looking for in the OO model. You may develop a solution
using the OO method without reusability in mind. Developing the initial classes and/or modules
can be a tedious task to do properly and may actually take longer than to solve the current
problem in a procedural manner. Creating classes that can be reused by other developers in
the future or that make maintenance more efficient is where the real payback is.
Why Use Object-Oriented Programming?

- Data encapsulation.

  The data which belongs to each object is accessible only through the “public interface” for that class. The public interface is the documented way (methods) to use the object. Perl does not enforce data encapsulation but the developer is encouraged to provide a well documented interface so that downstream developers may use these objects easily. Don’t publicly document the “private data” or “private methods.”

- Speed application development time.

  If implemented effectively, an Object-Oriented solution will help speed later application development efforts which might use the same objects. Many pre-defined objects are available to the developer in the form of class libraries or, in Perl’s case, modules. The developer may inherit much of what is needed from these classes. This is reusability.

- Ease maintenance overhead.

  If a change needs to be made to all of the objects of a particular class, the change need only be made in the class definition or the implementation of the methods. The user interface need not (should not, if possible) change.

- Intuitive.

  Once you begin learning and implementing solutions using the OO paradigm, you will probably find it more intuitive than procedural development.
#!perl
#File: Name.pl

package Name;  # The name of a class

sub new {  # class method to create and return the reference to the "object"
    my $class = shift @_;  # first argument provided automatically by Perl
    my $this = {};  # create the reference to the anonymous hash
    bless $this, $class;  # bind the reference to the package (class) "Name"
    return $this;  # return the reference to the caller for assignment to a scalar
}

sub set_lastname {  # instance method
    $this = shift @_;  # get the reference to the object which invoked me
    $$this{last_name} = shift @_;  # assign to that objects instance data
}

sub set_firstname {  # instance method
    $this = shift @_;  # get the reference to the object which invoked me
    $$this{first_name} = shift @_;  # assign to that objects instance data
}

sub get_lastname {  # instance method
    $this = shift @_;  # get the reference to the object which invoked me
    return $$this{last_name};  # return that object's instance data
}

sub get_firstname {  # instance method
    $this = shift @_;  # get the reference to the object which invoked me
    return $$this{first_name};  # return that object's instance data
}

1;  # Return an arbitrary true value, so "require" doesn't fail.

We can use this class in any program:

# File: namemain.pl

require "Name.pl";  # Pull in the definition of the name class

$name1 = new Name;  
$name2 = new Name;
$name1->set_lastname('Skywalker'); $name1->set_firstname('Luke');
$name2->set_lastname('Leia'); $name2->set_firstname('Princess');
print $name1->get_firstname, " ", $name1->get_lastname, "\n";
print $name2->get_firstname, " ", $name2->get_lastname, "\n";

The above code generates the following output:

Luke Skywalker
Princess Leia
Classes, Objects, and Methods in Perl

- In Perl, a class is simply a package which provides methods to deal with objects.

```
package Name; # The name of a class
```

- In Perl, an object is a reference to a thing which knows it belongs to a particular package (that is, a class) – the `bless` operator tells it which package it belongs to.

  - The thing may be an anonymous hash, anonymous array, scalar or anonymous subroutine.
  
  - The `new` method returns a reference to a thing, most typically an anonymous hash (an easily extensible data structure).

```
sub new {
    $class = shift @_; # The first argument is provided automatically
    $this = {}; # Creates a reference to an empty anonymous hash
    bless $this, $class; # Binds the reference to the class (package) name
    return $this; # Returns the reference to the caller for assignment
}
```

- A method is a subroutine that expects an object reference or package name as its first argument

  - This argument is provided automatically.
#!perl
#File: FormalName.pl

require "Name.pl";  # Pull in the definition of the Name class.

package FormalName;  # The name of a new class
@ISA = qw( Name );  # We say that a FormalName "is-a" Name

sub new {
  my $class = shift @_;  
  $this = new Name;       # Instead of creating another anonymous hash reference
                          # we use the one from Name
  return bless $this, $class;
}
sub set_title {      # adds a new instance method
  my $this = shift @_;  
  $$this{title} = shift @_; 
}
sub get_title {      # adds a new instance method
  my $this = shift @_;  
  return $$this{title}; 
}
1;  # Return an arbitrary true value, so "require" doesn't fail.

We can now use our new, inherited FormalName class:

#!perl
#File: fnamemain.pl

require "FormalName.pl";  # Pull in the definition of the FormalName class.

$name1 = new FormalName;
$name2 = new FormalName;
$name1->set_lastname('Skywalker'); $name1->set_firstname('Luke');
$name1->set_title("The Honorable");
$name2->set_lastname('Leia'); $name2->set_firstname('Princess');
$name2->set_title("Her Royal Highness");
print $name1->get_title, " ", $name1->get_firstname, " ",
      $name1->get_lastname, "\n";
print $name2->get_title, " ", $name2->get_firstname, " ",
      $name2->get_lastname, "\n";

The above code generates the following output:

The Honorable Luke Skywalker
Her Royal Highness Princess Leia
Inheritance, the "is-a" Relationship

- Inheritance is one of the most important features in the OO model.
- It allows us to use part or all of another class’ methods and then add others to make our new class unique – reusability!

Class: Radio

Radio has buttons, a speaker, a dial, and a receiver.
Radio has controls for volume, select station, on/off.

Class: Stereo Radio (descendant or subclass of the Radio class)

Has all the features of radio, PLUS another speaker.
Stereo Radio has controls for volume, select station, on/off
PLUS speaker balance and FM Stereo select.

Class: Stereo Radio w/Cassette Player

Has all the features of a Stereo Radio PLUS those for the Cassette functions. Stereo Radio w/Cassette Player has controls for volume, select station, on/off, speaker balance and FM Stereo select PLUS select cassette, play, fast forward, rewind, and eject.

Radio is the Base Class and Stereo Radio “is-a” Radio.

Stereo Radio w/Cassette Player “is-a” Stereo Radio.

- In Perl, the descendant class simply names its ancestor classes in a special array named @ISA and Perl will look in those classes for methods not found in the descendant.
  - The descendant should create an ancestor object in its new.
# File: Employee.pl
use Name;  # Need "Name.pm" - perl must find Name methods at compile time.

package Employee;        # The name of a base class

sub new {
    my $class = shift @_;  # first argument provided automatically by Perl
    my $this = {};
    bless $this, $class;
    $this->initialize(@_);  # Initialize - create instance data
    return $this;
}

sub initialize {
    my $this = shift @_;  
    $this->{name} = new Name;
}

sub set_ssn {
    my $this = shift @_;  
    $this->{ssn} = shift @_;  
}

sub get_ssn {
    my $this = shift @_;  
    return $this->{ssn};
}

sub display {
    my $this = shift @_;  
    return $this->{name}->get_lastname"," ", ", $this->{name}->get_firstname, " SSN: ", $this->{ssn}," \n";
}

1;

We can use our Employee class, and the Name object it contains:

# File: empmain.pl
require "Employee.pl";

$emp1 = new Employee;
$emp2 = new Employee;

$emp1->{name}->set_lastname('Skywalker'); $emp1->{name}->set_firstname('Luke');
$emp2->{name}->set_lastname('Leia'); $emp2->{name}->set_firstname('Princess');
$emp1->set_ssn("333-33-3333");
$emp2->set_ssn("999-99-9999");
print $emp1->display();
print $emp2->display();

The above code generates the following output:

Skywalker, Luke    SSN: 333-33-3333
Leia, Princess    SSN: 999-99-9999
Many objects have other objects as data members, called containment.

An Employee record may consist of simple data, like a Social Security Number.

It would also contain more complex constructs like our Name or FormalName class.

In this case, an employee record “has-a” Name; a Name object would be a data member to the Employee object.

Since the Perl compiler needs to check the containing class' use of member object methods at compile time, the contained object should be a Perl module: a .pm, included with use.

In fact, to provide the greatest flexibility and reuse, you'll want to define all your classes as Perl modules.
#!perl
#File: ovloadtest.pl

use FormalName;     # use our FormalName.pm module

$name_1 = new FormalName;
$name_1->set_title("Big Furry");
$name_1->set_firstname("Chew");
$name_1->set_lastname("Bacca");
$name_2 = $name_1;  # Now they reference same object! We want a copy!

print "name_1: ",$name_1->get_title," ",$name_1->get_firstname()," ",
      $name_1->get_lastname(),"\n";
print "name_2: ",$name_2->get_title," ",$name_2->get_firstname()," ",
      $name_2->get_lastname(),"\n";

# Uncomment to reset name_2 to no longer point to same thing as name_1
# $name_2 = new FormalName;

$name_2->set_title("Big Scarey");
#$name_2->set_firstname("Darth"); #Let's see what happens...
$name_2->set_lastname("Vader");

print "name_1: ",$name_1->get_title," ",$name_1->get_firstname()," ",
      $name_1->get_lastname(),"\n";
print "name_2: ",$name_2->get_title," ",$name_2->get_firstname()," ",
      $name_2->get_lastname(),"\n";

The above code generates the following output:

name_1: Big Furry Chew Bacca
name_2: Big Furry Chew Bacca
name_1: Big Scarey Chew Vader
name_2: Big Scarey Chew Vader
Some OO languages (such as C++) allow *operator overloading*.

- You can make a standard language operator (+, =, *, etc.) behave differently depending on the class of its operands.

- For example, one common operator to overload would be the assignment operator (=):

  ```perl
  $obj1 = new SomeClass;
  $obj2 = $obj1;
  ```

- Normally, $obj2 simply becomes another reference to the same data and methods as $obj1, not a copy.

- You would overload the = operator so that if its right-hand operand is a SomeClass object, then it creates a new object which is a copy of the right-hand operand (this is often called a *copy constructor*) and assign a reference to the new object to the variable on the left-hand side.

- Many of Perl’s built-in operators can be overloaded using the overload.pm module.

- The overload module does provide a mechanism to fake overloading the assignment operator; it will be invoked when a mutator (like ++ or +=) is applied to $obj2 (see facing page code).
#!perl
#File: Name2.pl

package Name2; # The name of a class

sub new { # class method to create and return the reference to the "object"
my $class = shift @_; # first argument provided automatically by Perl
my $this = {}; # create the reference to the anonymous hash
bless $this, $class; # bind the reference to the package (class) "Name2"
return $this; # return the reference to the caller for assignment to a scalar
}

sub set_lastname { # instance method
$this = shift @_; # get the reference to the object which invoked me
$$this{last_name} = shift @_; # assign to that objects instance data
}

sub set_firstname { # instance method
$this = shift @_; # get the reference to the object which invoked me
$$this{first_name} = shift @_; # assign to that objects instance data
}

sub get_lastname { # instance method
$this = shift @_; # get the reference to the object which invoked me
return $$this{last_name}; # return that object's instance data
}

sub get_firstname { # instance method
$this = shift @_; # get the reference to the object which invoked me
return $$this{first_name}; # return that object's instance data
}

sub DESTROY {
    print "Goodbye from ", $_->[0]->{first_name}, " ", $_->[0]->{last_name}, "\n";
}

1; # Return an arbitrary true value, so "require" doesn't fail.

DESTROY will be called when the last reference to each Name2 object goes out of scope:

#!perl
#File: namemain2.pl

require "Name2.pl"; # Pull in the definition of the Name2 class

$name1 = new Name2;
$name2 = new Name2;
$name1->set_lastname('Skywalker'); $name1->set_firstname('Luke');
$name2->set_lastname('Leia'); $name2->set_firstname('Princess');
print $name1->get_firstname, " ", $name1->get_lastname, "\n";
print $name2->get_firstname, " ", $name2->get_lastname, "\n";

The above code generates the following output:

Luke Skywalker
Princess Leia
Goodbye from Luke Skywalker
Goodbye from Princess Leia
Destructors

- Destructor methods are called automatically by Perl whenever the last reference to an object goes out of scope.
  - The destructor must be called `DESTROY`.

- They are used for any cleanup above and beyond simple memory cleanup, which Perl does automatically.

- You may do things like close and remove temporary files, remove lock files, close database connections, etc.

- Do not create any new references to the object being destroyed.
Notes
Labs

1. Copy `carsClass.pl` to `carsClass.pm` and modify it to become a full object. You will need to create a subroutine `new` that will perform a `bless` on `$this` and initialize `$this` with the data from the `cars` file. Do not forget to update the other functions to shift off the first parameter into `$this`. Lastly, you will need to copy `carsmain.pl` and modify it to call your new object. (Solution: `carsobj.pl`, `carsClass.pm`)

2. Create a new module `carsFunc.pm` to override the `print` function. You will create a function `new` that will call `CarClass`' `new` and then re-bless this into the called class. Do not forget to add `@ISA`. In the `print` function, you may call `CarClass`' `print` by using `SUPER` as the package name. (Solution: `carsobj.pl`, `carsFunc.pm`)