A Quick, Painless Introduction to the Perl Scripting Language

Norman Matloff
University of California, Davis

May 24, 2004

Contents

1 What Are Scripting Languages?

Languages like C and C++ allow a programmer to write code at a very detailed level which has good execution speed. But in many applications one would prefer to write at a higher level. For example, for text-manipulation applications, the basic unit in C/C++ is a character, while for languages like Perl and Python the basic units are lines of text and words within lines. One can work with lines and words in C/C++, but one must go to greater effort to accomplish the same thing. C/C++ might give better speed, but if speed is not an issue, the convenience of a scripting language is very attractive.

The term *scripting language* has never been formally defined, but here are the typical characteristics:

- Used often for system administration and "rapid prototyping."
- Very casual with regard to typing of variables, e.g. no distinction between integer, floating-point or string variables. Functions can return nonscalars, e.g. arrays, nonscalars can be used as loop indexes, etc.
- Lots of high-level operations intrinsic to the language, e.g. stack push/pop.
- Interpreted, rather than being compiled to the instruction set of the host machine.

Today the matloff many people, including me, strongly prefer Python, as it is much cleaner and more elegant.

Our introduction here assumes knowledge of C/C++ programming. There will be a couple of places in which we describe things briefly in a UNIX context, so some UNIX knowledge would be helpful.[1]

2 Goals of This Tutorial

Perl is a very feature-rich language, which clearly cannot be discussed in full detail here. Instead, our goals here are to (a) enable the reader to quickly become proficient at writing simple Perl programs and (b) prepare the reader to consult full Perl books (or Perl tutorials on the Web) for further details of whatever Perl constructs he/she needs for a particular application.

Our approach here is different from that of most Perl books, or even most Perl Web tutorials. The usual approach is to painfully go over all details from the beginning. For example, the usual approach would be to state all possible forms that a Perl literal can take on.

We avoid this here. Again, the aim is to enable the reader to quickly acquire a Perl foundation. He/she should then be able to delve directly into some special topic, with little or not further learning of foundations.

3 A 1-Minute Introductory Example

This program reads a text file and prints out the number of lines and words in the file:

[1] But certainly not required. Again, Perl is used on Windows and Macintosh platforms too, not just UNIX.

```
# ome.pl, introductory example

# comments begin with the sharp sign

# open the file whose name is given in the first argument on the command
# line, assigning to a file handle INFILE (it is customary to choose
# all-caps names for file handles in Perl); file handles do not have any
# prefixing punctuation
open(INFILE,$ARGV[0]);

# names of scalar variables must begin with $
$line_count = 0;
$word_count = 0;

# <> construct means read one line; undefined response signals EOF
while ($line = <INFILE>) {
   $line_count++;
   # break $line into an array of tokens separated by " ", using split()
   # (array names must begin with @)
   @words_on_this_line = split(" ",$line);
   # scalar() gives the length of any array
   $word_count += scalar(@words_on_this_line);
}

print "the file contains ",$line_count," lines and ",
   $word_count, " words\n";
```

We'd run this program, say on the file **x**, by typing

```
perl ome.pl x
```

Note that as in C, statements in Perl end in semicolons, and blocks are defined via braces.

4 Variables

4.1 Types

Type is not declared in Perl, but rather is inferred from a variable's name (see below), and is only loosely adhered to.

Note that a possible value of a variable is **undef** (i.e. undefined), which may be tested for, using a call to **defined()**.

Here are the main types:

4.2 Scalars

Names of **scalar** variables begin with $.

Scalars are integers, floating-point numbers and strings. For the most part, no distinction is made between these.

There are various exceptions, though. One class of exceptions involves tests of equality or inequality. For example, use **eq** to test equality of strings but use **==** for numbers.

4.3 Arrays

4.3.1 Structure

Array names begin with @. Indices are integers beginning at 0.

Array elements are scalars, so that for example

```
@wt = (1,(2,3),4);
```

would have the same effect as

```
@wt = (1,2,3,4);
```

Thus array elements begin with $, not @, e.g.

```
@wt = (1,2,3,4);
print $wt[2];  # prints 3
```

Arrays are referenced for the most part as in C, but in a more flexible manner. Their lengths are not declared, and they grow or shrink dynamically, without "warning," i.e. the programmer does not "ask for permission" in growing an array. For example, if the array **x** currently has 7 elements, i.e. ends at **$x[6]**, then the statement

```
$x[7] = 12;
```

changes the array length to 8. For that matter, we could have assigned to element 99 instead of to element 7, resulting in an array length of 100.

The programmer can treat an array as a queue data structure, using the Perl operations **push** and **shift** (usage of the latter is especially common in the Perl idiom), or treat it as a stack by using **push** and **pop**.

An array without a name is called a **list**. For example, in

```
@x = (88,12,"abc");
```

we assign the array name **@x** to the list (88,12,"abc"). We will then have **$x[0]** = 88, etc.

One of the big uses of lists and arrays is in loops, e.g.:[2]

```
# prints out 1, 2 and 4
for $i ((1,2,4))  {
   print $i, "\n";
}
```

[2] C-style **for** loops can be done too.

The length of an array or list is obtained calling **scalar()**, or by simply using the array name in a scalar context.

4.3.2 Operations

```
$x[0] = 15;
$x[1] = 16;
$y = shift @x;  # "output" of shift is the element shifted out
print $y, "\n";  # prints 15
print $x[0], "\n";  # prints 16
push(@x,9);  # sets $x[1] to 9
print scalar(@x), "\n";  # prints 2
print @x, "\n";  # prints 169 (16 and 9 with no space)
$k = @x;
print $k, "\n";  # prints 2
@x = ();  # @x will now be empty
print scalar(@x), "\n";  # prints 0
@rt = ('abc',15,20,95);
delete $rt[2];
print "@rt\n";  # prints "abc" 15 95
print "$rt[-1]\n";  # prints 95
```

A useful operation is array **slicing**:

```
@z = (5,12,13,125);
@w = @z[1..3];  # @w will be (12,13,125)
@q = @z[0..1];  # @q will be (5,12)
```

4.3.3 An Ambiguity

The slicing operation leads to a bit of an unfortunate ambiguity.

```
@x = (1,2,3);
@u = @x[2];  # RHS is short for @x[2..2], so @u is a 1-element array
$v = $x[2];  # $v is a scalar
print @u, " ", $v, "\n";
```

The output will be

```
3 3
```

So, in a lot of Perl programs out in the world, one sees statements like

```
open(INFILE,@ARGV[0]);
```

when

```
open(INFILE,$ARGV[0]);
```

would be the more "natural" use.

4.4 Hashes

As a first look, you can think of **hashes** or **associative arrays** as arrays indexed by strings instead of by integers. Their names begin with %, and their elements are indexed using braces, as in

```
$h{"abc"} = 12;
$h{"defg"} = "San Francisco";
print $h{"abc"}, "\n";  # prints 12
print $h{"defg"}, "\n";  # prints "San Francisco"
```

However, a closer look at hashes reveals them to essentially be like C **struct**s. In the above example, for instance, we have set up a hash named %h which is analogous to a C **struct** with **int** and **char []** fields, whose values here are 12 and "San Francisco", respectively. This correspondence is more clear in the equivalent (and more commonly used) alternative code

```
%h = (abc => 12,
      defg => "San Francisco");
print $h{"abc"}, "\n";  # prints 12
print $h{"defg"}, "\n";  # prints "San Francisco"
```

Here the first two lines look rather the declaration of a C **struct**, as in

```
struct ht {
   int abc;
   char defg[20];
};

struct ht h;

h.abc = 12;
strcpy(h.defg,"San Francisco");
```

Note, however, that there is no analog of **ht** in our Perl example above. In fact, there are lots of other differences. For example, unlike C **struct**s, hashes actually store their field names. In the example above, the number 12 and the string "San Francisco" are stored, but not the field names abc and defg. By contrast, Perl stores both! In the code above, if we add the line

```
print %h, "\n";
```

the output of that statement will be

```
abc12defgSan Francisco
```

4.5 References

References are like C pointers. They are considered scalar variables, and thus have names beginning with $. They are dereferenced by prepending the symbol for the variable type, e.g. prepending a $ for a scalar, a @ for an array, etc.:

```
1  # set up a reference to a scalar
2  $r = \3;  # \ means "reference to," like & means "pointer to" in C
3  # now print it; $r is a reference to a scalar, so $$r denotes that scalar
4  print $$r, "\n";  # prints 3
5
6  @x = (1,2,4,8,16);
7  $s = \@x;
8  # an array element is a scalar, so prepend a $
9  print $$s[3], "\n";  # prints 8
10 # for the whole array, prepend a @
11 print scalar(@$s), "\n";  # prints 5
```

In Line 4, for example, you should view **$$r** as **$($r)**, meaning take the reference **$r** and dereference it. Since the result of dereferencing is a scalar, we get another dollar sign on the left.

4.6 Anonymous Data

Anonymous data is somewhat analogous to data set up using **malloc()** in C. One sets up a data structure without a name, and then points a reference variable to it.

A major use of anonymous data is to set up object-oriented programming, if you wish to use OOP. (Covered in Section 9.)

Anonymous arrays use brackets instead of parentheses. (And anonymous hashes use braces.) The $->$ operator is used for dereferencing.

Example:

```
# $x will be a reference to an anonymous array
$x = [5, 12, 13];
print $x->[1], "\n";  # prints 12

# $y will be a reference to an anonymous hash (due to braces)
$y = {name => "penelope", age=>105};
print $y->{age}, "\n";  # prints 105
```

Note the difference between

```
$x = [5, 12, 13];
```

and

```
$x = (5, 12, 13);
```

The former sets **$x** as a reference to the anonymous list [5,12,13], while the latter sets **$x** to the length of the anonymous list (5,12,13). So the brackets or parentheses, as the case may be, tell the Perl interpreter what we want.

4.7 Declaration of Variables

A variable need not be explicitly declared; its "declaration" consists of its first usage. For example, if the statement

```
$x = 5;
```

were the first reference to **$x**, then this would both declare **$x** and assign 5 to it.

If you wish to make a separate declaration, you can do so, e.g.

```
$x;
...
$x = 5;
```

If you wish to have protection against accidentally using a variable which has not been previously defined, say due to a misspelling, include a line

```
use strict;
```

at the top of your source code.

4.8 Scope of Variables

Variables in Perl are global by default. To make a variable local to subroutine or block,[3] the **my** construct is used.[4]

5 Subroutines

5.1 Arguments, Return Values

Arguments for a subroutine are passed via an array @_. Note once again that the @ sign tells us this is an array; we can think of the array name as being _, with the @ sign then telling us it is an array.

Here are some examples:

```
1  # read in two numbers from the command line (note: the duality of
2  # numbers and strings in Perl means no need for atoi()!)
3  $x = $ARGV[0];
4  $y = $ARGV[1];
5  # call subroutine which finds the minimum and print the latter
6  $z = min($x,$y);
7  print $z, "\n";
8
```

[3] This includes, for example, a block within an **if** statement.

[4] There are many other scope possibilities, e.g. namespaces of packages.

```
9  sub min {
10    if ($_[0] < $_[1]) {return $_[0];}
11    else {return $_[1];}
12 }
```

A common Perl idiom is to have a subroutine use **shift** on @_ to get the arguments and assign them to local variables.

Arguments must be pass-by-value, but this small restriction is more than compensated by the facts that (a) arguments can be references, and (b) the return value can also be a list.

Here is an example illustrating all this:

```
$x = $ARGV[0];
$y = $ARGV[1];
($mn,$mx) = minmax($x,$y);
print $mn, " ", $mx, "\n";

sub minmax {
   $s = shift @_;  # get first argument
   $t = shift @_;  # get second argument
   if ($s < $t) {return ($s,$t);}  # return a list
   else {return ($t,$s);}
}
```

A Perl quirk or feature, depending on your point of view, is that any subroutine, even if it has no **return**, will return the last value computed. So for instance in the **minmax()** example above, it would still work even if we were to remove the two **return** statements.

5.2 Alternative Notation

Instead of enclosing arguments within parentheses, as in C, one can simply write them in "command-line arguments" fashion. For example, the call

```
($mn,$mx) = minmax($x,$y);
```

can be written as

```
($mn,$mx) = minmax $x,$y;
```

In fact, we've been doing this in all our previous examples, in our calls to **print()**. This style is often clearer.

On the other hand, if the subroutine, say **x()**, has no arguments make sure to use the parentheses in your call:

```
x();
```

rather than

```
x;
```

In the latter case, the Perl interpreter will treat this as the "declaration" of a variable **x**, not a call to **x()**.

5.3 Passing Subroutines As Arguments

Older versions of Perl required that subroutines be referenced through an ampersand preceding the name, e.g.

```
($mn,$mx) = &minmax $x,$y;
```

In some cases we must still do so, such as when we need to pass a subroutine name to a subroutine. The reason this need arises is that we may write a packaged program which calls a user-written subroutine.

Here is an example of how to do it:

```
1  sub x  {
2     print "this is x\n";
3  }
4
5  sub y  {
6     print "this is y\n";
7  }
8
9  sub w {
10    $r = shift;
11    &$r();
12 }
13
14 w \&x;  # prints "this is x"
15 w \&y;  # prints "this is y"
```

Here **w()** calls **r()**, with the latter actually being either **x()** or **y()**.

6 Confusing Defaults

In many cases, in Perl the operands for operators have defaults if they are not explicitly specified. Within a subroutine, for example, the array of arguments @_, can be left implicit. The code

```
sub uuu {
   $a = shift;  # get first argument
   ...
}
```

will have the same effect as

```
sub uuu {
   $a = shift @_;  # get first argument
   ...
}
```

This is handy for experienced Perl programmers but a source of confusion for beginners.

Similarly,

```
$line = <>;
```

reads a line from the standard input (i.e. keyboard), just as the more explicit

```
$line = <STDIN>;
```

would.

A further "simplification" is that if you read in a line from STDIN but don't assign it to anything, it is assigned to **$_**. For instance, consider the code

```
while ($line = <STDIN>) {
   print "the input was $line";
}
```

(The user terminates his/her keyboard input via ctrl-d.)

This simplifies to

```
while (<>) {
   print "the input was $_";
}
```

Moreover, **$_** is the default value for many function arguments. If you don't supply the argument, the Perl interpreter will take it to be **$_**. You'll see examples of this later.

7 String Manipulation in Perl

One major category of Perl string constructs involves searching and possibly replacing strings. For example, the following program acts like the UNIX **grep** command, reporting all lines found in a given file which contain a given string (the file name and the string are given on the command line):

```
open(INFILE,$ARGV[0]);
while ($line = <INFILE>) {
   if ($line =~ /$ARGV[1]/)  {
      print $line;
   }
}
```

Here the Perl expression

```
   ($line =~ /$ARGV[1]/)
```

checks $line for the given string, resulting in a **true** value if the string is found.

In this string-matching operation Perl allows many different types of **regular expression** conditions.[5] For example,

[5] If you are a UNIX user, you may be used to this notion already.

```
open(INFILE,$ARGV[0]);
while ($line = <INFILE>) {
   if ($line =~ /us[ei]/)  {
      print $line;
   }
}
```

would print out all the lines in the file which contain *either* the string "use" *or* "usi".

Substitution is another common operation. For example, the code

```
open(INFILE,$ARGV[0]);
while ($line = <INFILE>) {
   if ($line =~ s/abc/xyz/)  {
      print $line;
   }
}
```

would cull out all lines in the file which contain the string "abc", replace the first instance of that string in the line by "xyz", and then print out those changed lines.

There are many more string operations in the Perl repertoire.

As mentioned earlier, Perl uses **eq** to test string equality; it uses **ne** to test string inequality.

A popular Perl operator is **chop**, which removes the last character of a string. This is typically used to remove an end-of-line character. For example,

```
chop $line;
```

removes the last character in **$line**, and reassigns the result to **$line**.

Since **chop** is actually a function, and in fact one that has **$_** as its default argument, if **$line** had been read in from STDIN, we could write the above code as

```
chop;
```

A related function is **chomp**.

8 Perl Packages/Modules

In the spirit of modular programming, it's good to break up our program into separate files, or **packages**. Perl 5.0 specialized the packages notion to **modules**, and it will be the latter that will be our focus here. Except for the top-level file (the one containing the analog of **main()** in C/C++), a file must have the suffix **.pm** ("Perl module") in its name and have as its first noncomment/nonblank line a **package** statement, named after the file name. For example the module **X.pm** would begin with

```
package X;
```

The files which draw upon **X.pm** would have a line

```
use X;
```

near the beginning.

The **::** symbol is used to denote membership. For example, the expression **$X::y** refers to the scalar **$y** in the file **X.pm**, while **@X::z** refers to the array **@z** in that file.

The top-level file is referred to as **main** from within other modules.

As you can see, packages give one a separate namespace.

Modules which are grouped together as a library are placed in the same directory, or the same directory tree.

For example, if you do network programming (see Section B), you will probably need to include a line

```
use IO::Socket;
```

in your code. Let's look at this closely.

First, part of your Perl programs environment is the Perl search path, in which the interpreter looks for packages that your code uses. This path has a default value, but you can change it by using the **-I** option when you invoke Perl on the command line.

In the above example, the interpreter will look in your search path for a directory **IO**. At that point, the interpreter will consider two possibilities:[6]

- there is a file **IO/Socket.pm** where the package code resides, or
- there is a directory **IO/Socket**, within which there are various **.pm** files which contain the package code

In our case here, it will be the latter situation. For example, on my Linux machine, the directory **/usr/lib/perl5/5.8.0/IO/Socke** contains the files **INET.pm** and **UNIX.pm**, and the socket code is in those files.

Again, the **package** keyword is needed. For instance, in our example above, the first non-comment line of **INET.pm** is

```
package IO::Socket::INET;
```

Any package which contains subroutines must return a value. Typically one just includes a line

```
1;
```

[6] If you know Java, you may notice that this is similar to the setup for Java packages.

at the very end, which produces a dummy return value of 1.

There are many public-domain Perl modules available in CPAN, the Comprehensive Perl Archive Network, which is available at several sites on the Web. Moreover, the process of downloading and installing them has been automated!

For example, suppose you wish to write (or even just run) Perl programs with Tk-based GUIs. If the Perl Tk module is not already on your machine, just type

```
perl -MCPAN -e "install Tk"
```

to install Tk from the network. You will be asked some questions, but just taking the defaults will probably be enough.

9 OOP in Perl

(This section requires some knowledge of OOP, e.g. from C++ or Java.)

9.1 General Mechanisms

Though object-oriented programming capability came to Perl as a belated add-on, it is done in a fairly simple and easy-to-use (if not clean and elegant) manner. Here is an introduction, with some simple code for a warehouse inventory program.

9.2 Overview

The following overview should be read both before and after reading the example below in Sec. 9.3, as it won't be fully clear until after the reader sees an example.

Note first that instead of adding a lot of new constructs when OOP was added to Perl, the Perl development team decided to cobble together "classes" out of existing Perl constructs. So, in the following, keep in mind that you won't be seeing me say things like, say, "A Perl class starts with the keyword **class**." Instead, we put together various items to get the *effect* of classes.

So, having said that, here is the Perl "kludge" for OOP:

- A Perl class, say **X**, consists of a package file, say **X.pm**.
- An instance of that class **X**, i.e. an object of class **X**, consists of an anonymous hash. The elements in that hash are the instance variables of the class.
- Class variables are stored in freestanding variables in the package file.
- The class' methods are subroutines in the package file. One of them will serve as the constructor of the class.

Here is what the constructor code will consist of:

- We set up an anonymous hash.
- We point a reference variable to the hash.
- We perform a **bless** operation, which associates the hash with the class name, i.e. that package file.

The "bless" operation returns the reference, which then serves as a pointer to the object. Again, that object is simply the hash, but now with the additional information that it is associated with the package/class **X**.

Since the constructor will be called explicitly by the programmer, the programmer can give any name to the constructor method for the class. It is common to name it **new()**, though.

Class and instance methods are not distinguished in any formal way. The methods, being subroutines, of course have an argument vector @_, but the difference will be in the first argument:

- if the method is invoked on the class, **$_[0]**, will point to the class
- if the method is invoked on the object, **$_[0]**, will point to the object

For example, say we have a reference **$r** which points to an object of class **X**, and the class contains a subroutine **s()** with a single integer argument, then

```
X->s(12);
```

would set @_ to consist of a pointer to **X** and the number 12, while

```
$r->s(12);
```

would set @_ to consist of a pointer to the object (i.e. it would be **$r**) and the number 12. Thus **s()** would serve as either a class method or an instance method, according to context.

In other words, if you call a method via an object name, the Perl interpreter will prepend an additional argument to the set of arguments you explicitly supply in the call. That additional argument will be a pointer to the object. Readers with experience with various languages will note the similarity to **this** in C++ and Java, which plays the role of an implicit extra argument, and to **self** in Python, in which the argument is explicit. Here in Perl, the additional argument is explicit in the sense that it is added to the argument list @_, though it is not explicitly there in the call's argument list.

If the method is called via the class name, the Perl interpreter also adds an additional argument, in this case the class name.

Note that without the **bless** operation, the interpreter would not have enough information to make decisions like.

9.3 Example

First we have the file **Worker.pm**:

```
package Worker;

# class variables
@workers = ();  # set of (references to) workers
$lastupdate = undef;  # time database was last updated

# this will be the constructor
sub newworker {
   # note the repeated use of "my" to keep things local
   my $classname = shift;  # class name (Worker) is first argument
   my ($nm,$id,$slry) = @_;  # get the other arguments
   # set up an anonymous hash and point a reference to it; our instance
   # variables will be {\bf name}, {\bf id} and {\bf salary}
   my $r = {name => $nm, id => $id, salary => $slry};
   # add this worker to the database
   push(@workers,$r);
   # set the update time
   $lastupdate = localtime();
   # associate the hash with this class, and return the reference
   bless($r,$classname);
   return $r;
}

# an instance method
sub printworker {
   my $r = shift;  # object is first argument
   print "employee number $r->{id}, named $r->{name}, has a salary of ",
      "$r->{salary}\n";
}

1;  # a quirk in Perl; must include dummy return value as last line

# note:  we have no provision here for connecting the workers into one
# set, but we could have a class variable which is the head of a linked
# list, and have newworker add the new object to that list
```

Here is a file which uses the **Worker** package:

```
use Worker;

# add two workers to the database by creating two new objects
$w1 = Worker->newworker("Cassius",12,50000);
$w2 = Worker->newworker("Penelope",5,90000);

# print the workers
$w1->printworker;
$w2->printworker;

# can access the objects from outside the class
print $$w1{name}, "\n";  # prints "Cassius"
print $w2->{id}, "\n";  # prints 5
print "time of last update: $Worker::lastupdate\n"
```

10 Debugging in Perl

(Before reading this section, if you are not a veteran debugger you may wish to view the author's debugging slide show, at `http://heather.cs.ucdavis.edu/~matloff/debug.html`.)

10.1 Perl's Own Built-in Debugger

Perl has its own reasonably-good debugging tool built right in to the interpreter. One merely runs Perl with the **-d** option on the command line, e.g.

```
perl -d x.pl
```

Here are some of the debugger operations:

```
b k         set a breakpoint at line/subroutine k
b k c       set a breakpoint at line/subroutine k, with boolean condition c
B k         delete a breakpoint at line/subroutine k
n           go to next line, skipping over subroutine calls
s           go to next line, not skipping over subroutine calls
c           continue until breakpoint
c k         continue until line k/subroutine (one-time breakpoint)
L           list all breakpoints/actions
d k         delete breakpoint at line k
a k c       execute Perl command ("action") c each time hit breakpoint at k
a k         delete action at line k
r           finish this subroutine without single-stepping
p y         print y
x y         examine y (nicer printing of y)
T           stack trace
l           list a few lines of source code
!           re-do last command
H           list all recent commands
!n          re-do command n
h           help
|c          do debugger command c, but run output through pager
R           attempt a restart, retaining breakpoints etc.
q           quit
=           set a variable to a specified value
```

For an example of the use of **=** command, the following would set the variable **$z** to 5:

```
DB <1> $z = 5
```

The "DB **<** 1**>** " is the prompt, showing that this is the first command that we've given.

Note that the **p** command, applied to an array or hash, strings together the elements; try using **x** instead. For example:

```
main::(u.pl:2): @u = (1,2,3);
  DB<1> n
```

```
main::(u.pl:3): %v = ("abc" => 12, 8 => 3);
  DB<1> n
main::(u.pl:4): $z = 0;
  DB<1> p @u
123
  DB<2> p %v
83abc12
  DB<3> x @u
0  1
1  2
2  3
  DB<4> x %v
0  8
1  3
2  'abc'
3  12
```

The **a** command is very useful. E.g.

```
a 54 print "$x, $y\n"
```

would result in printing out **$x** and **$y** every time you hit the breakpoint at line 54.

10.2 GUI Debuggers

The built-in Perl debugger is certainly quite serviceable, but of course it is nicer to have a GUI through which to use the built-in debugger. Several debuggers provide this.

If you are on a UNIX system, I recommend DDD.[7] By using DDD, one has the same GUI no matter whether one is debugging C/C++, Java, Perl or Python, so that one doesn't have to remember how to use multiple different GUIs. DDD acts as a GUI front end to text-based debuggers for those languages.

To invoke DDD on, say, **x.pl**, simply type

```
ddd x.pl
```

DDD will infer that this is a Perl script by the **.pl** suffix, and invoke **perl -d** on **x.pl**.

One can set command-line arguments by selecting Run | Run with Arguments.

If you use the Eclipse IDE, there is a Perl plugin available at `http://e-p-i-c.sourceforge.net/`.

11 To Learn More

There are obviously a great many good books on Perl. One which I like is *Perl by Example* (third edition), by Ellen Quigley.

[7] See the above debugging Web page URL for information.

There also are many books which treat various specialized aspects of Perl, such as Perl network programming, Perl/Tk GUI programming and so on.

There are a number of Perl Web pages, the "official" one being `http://www.perl.com/`.

If you are on a UNIX system, type

```
man perl
```

which will lead you to other man pages. For example,

```
man perlfunc
```

will give you the documentation for all of Perl's built-in functions.

CPAN, an archive of many contributed Perl modules, was discussed earlier, in Section 8.

A The Tie Operation

Perl's **tie** operation[8] is an advanced topic. Again, this section may be skipped without loss of continuity.

What tie() does is a form of operator overloading. It will associate a scalar variable[9] with a class object. That class is required to have subroutines named TIESCALAR, FETCH and STORE: TIESCALAR is the constructor function, and FETCH and STORE are called each time the scalar needs to be fetched (e.g. is on the right-hand side of an assignment statement) or stored (e.g. is on the left side). The methods in the class then allow one to intervene in the fetch or store.

For example, suppose we have an integer variable whose value should not go outside a certain range. We can use tie() to do runtime checks for exceeding that range. We first set up the class, say BoundedInt.pm:

```
1  # use of tie() to implement bounds checking on integer variables
2
3  # arguments to tie() are the class name, BoundedInt, the name of
4  # the tied variable, and the lower and upper bounds
5
6  package BoundedInt;
7
8  sub TIESCALAR {
9     my $class = @_[0];
10    my $r = {Name=>@_[1], Value=>0, LBd=>@_[2],UBd=>@_[3]};
11    bless $r, $class;
12    return $r;
13 }
14
15 sub FETCH {
16    my $r = shift;
17    return $r->{Value};
18 }
```

[8] Actually it is a Perl built-in function, **tie**().

[9] Or an array or a hash.

```
19
20  sub STORE {
21     my $r = shift;
22     my $tryvalue = shift;
23     if ($tryvalue < $r->{LBd} || $tryvalue > $r->{UBd})  {
24        print "out-of-bounds value for ",$r->{Name},":  ",$r->{Value}, "\n";
25     }
26     else  {
27        $r->{Value} = $tryvalue;
28     }
29
30  }
31
32  1;
```

Now, here is a test program:

```
use BoundedInt;

package main;

$x;

tie $x,'BoundedInt','$x',1,10;
$x = 5;  # OK
$x = 15;  # causes an error message
$x = 3;  # OK

exit;
```

As seen above, the first two arguments to **tie()** must be the variable, then the class name. Further arguments depend on the application.

B Networking in Perl

(This section requires an elementary knowledge of TCP/IP programming. The section may be skipped without compromising understanding of the remaining material. For a quick but rather thorough introduction to networks and TCP/IP, see `http://heather.cs.ucdavis.edu/~matloff/Networks/Intro/NetIntro.pdf`.)

Perl includes analogs of most of the TCP/IP calls one is accustomed to using from C, including **select()**. The arguments to the calls are similar, though Perl, being at a higher level than C, makes many of these calls more convenient.

Here is an example of code which a client may use to connect to a server:

```
use IO::Socket;

# get server and port from command line
my $srvrip  = $ARGV[0];
my $port = $ARGV[1];
my $svrskt = new IO::Socket::INET(PeerAddr=>$srvrip,
                            PeerPort=>$port, Proto=>'tcp');
```

Nothing more need be done. The call to **new()** incorporates what in C would be calls to **socket()**, **connect()** and so on.

Assume that the data is line-oriented. Then the client would send a string **$x** (which, say, did not already have an end-of-line character) as

```
$x = $x . "\n";  # . is Perl's string-concatenate operator
print $svrskt $x;
```

and would read a line, say **$y**, as

```
$y = <$svrskt>;
```

The server would have code something like

```
$port = $ARGV[0];
$listensocket = IO::Socket::INET->new(Proto=>'tcp', LocalPort=>$port,
      Listen=>1, Reuse=>1);
...
$sock = $listensocket->accept();
$z = <$sock>;
...
print $sock $w, "\n";
```