On Perl

Perl for Students and Professionals

Jugal K. Kalita

On Perl:
Perl for Students and Professionals

Universal Publishers/uPUBLISH.com
USA • 2004

ISBN: 1-58112-550-X

www.upublish.com/books/kalita.htm

In Memory of Loving Mother Nirala Kalita

Preface

During our life time, computers have been responsible for ushering in an unprecedented age of swift technological changes that have permeated every sphere of life. For more than three decades, the speed at which computers perform computations has been doubling every eighteen months following an informal "law" formulated in 1965 by Gordon Moore, a co-founder of the chip-making powerhouse, the Intel Corporation. Along with the phenomenal increase in power, speed and storage capacity of computers, the cost of CPU, memory chips, and peripherals such as printers, monitors and scanners have also been falling rapidly. These trends will continue if not accelerate in the years to come.

Experts agree that for a long time, progress in computer software had lagged behind the enormous strides made in hardware. However, during the past decade, the revolutionary software product called the World Wide Web (WWW) came into our lives and quickly changed the world. The WWW, conceived by the European physicist Tim Berners-Lee in Switzerland in 1989, caught the world by storm. It has accelerated the rate at which the world is becoming a global village. Although many historians and economists believe that the current pace of technological change pales in comparison with the developments that took place in the last five decades of the nineteenth century,[1] it is true that in a few years, the WWW has been able to penetrate into millions of homes and corporations around the world. Information on any conceivable topic is at our finger tips with a few key strokes and mouse clicks. People of all ages are spending countless hours chatting on the WWW, making friends and developing relationships. WWW-based commerce is already worth tens of billions of dollars and is growing by leaps and bounds every day. In a few years, the Web is probably destined to become the dominant medium for conducting business and personal communications.

From its early days, Perl has been one of the dominant programming languages of choice in the context of the World Wide Web. Perl has been used extensively for CGI programming for providing dynamic Web pages. The extensive use of CGI programming encourages a lot of programmers to learn Perl. Perl also provides powerful tools for writing programs that communicate across the World Wide Web and that access network databases. One of Perl's strengths is its extra-ordinarily powerful pattern matching ability that can be used in looking for patterns in text including Web pages, thus making it suitable for data mining in many different areas such as business intelligence gathering and genomic studies.

Origin of Perl

Perl stands for *P*ractical *E*xtraction and *R*eporting *L*anguage. The name reflects its origin. It is an interpreted language with primarily dynamic typing created by Larry Wall for generating textual reports. He was once rifling through a hierarchy of files with the goal of extracting relevant information by scanning for textual patterns. He was working on a Unix machine and the premier pattern searching and processing tool for Unix machines called *awk* was not up to the task. So, Mr. Wall, the avid programmer, wrote a tool for achieving his immediate goal and other tasks he had in mind. This was the first version of Perl.

[1] *The Future Came Faster in the Old Days* by Steve Lohr, *The New York Times*, October 5, 1997

From its humble beginnings, Perl has gone through a large number of revisions and extensions. Perl 1 was released in January 1988, Perl 2 in June 1988, Perl 3 in October 1989, and Perl 4 in March 1991. The currently available major version, Perl 5, was released in October 1994. Perl 6 is in the works as of late 2003. No longer it is a language for pattern matching, but it has developed into a full-fledged language that has excellent facilities for many diverse tasks. Sophisticated handling of regular expressions still remains one of its strong points and it is one reason why Perl is very popular with programmers who create dynamic pages for the World Wide Web (WWW) using the technique called Common Gateway Interface (CGI).

Major Features of Perl

Perl is a multifaceted language with many strengths that make it an ideal practical language for many diverse tasks. Perl is an excellent tool for many tasks such as rapid prototype development, systems administration, and networking in addition to text processing and WWW programming.

Rapid Prototype Development

Perl programs are easy to write and they are portable to many different platforms. One does not have to know a lot of Perl's syntax to start programming. Thousands of programmers around the world have found that Perl is one of the best languages currently available for developing complex application programs quickly. It is a very high-level language and even a small Perl program can do tasks that can take hundreds of line of code in other languages. Programs developed in Perl can be tested and modified quickly. In an age where software packages are becoming bigger day by day and new versions of every software package are released every few months, it is of great advantage to developers if they are able to create a prototype for a sophisticated piece of software quickly.

Systems Administration

Although Perl was initially developed for processing textual files, it has been used by systems administrators in the Unix environment from its early days. The Unix operating system has no standards. With the release of OS X based on BSD Unix for Macintosh computers, Unix is likely to gain substantial additional followers and users as well. Different versions of Unix are produced and sold by various corporations. To fulfill Unix systems administration tasks, such as creating logins for new users, allocating resources and maintaining a secure environment for all users, and installing and uninstalling software packages, systems administrators primarily use programming languages called *shells*. There are many different shells and they work differently on different versions of Unix. The shell programs or scripts as they are called, are not always portable from one version of Unix to another. This is where Perl comes as a blessing. Because Perl has been ported to all versions of Unix and Perl has many capabilities that systems administrators desire and use, Perl scripts have started taking the place of shell scripts for many systems administrators. Perl has also been ported to various flavors of the Windows operating system. As a result, Perl is finding widespread use in systems administration tasks in non-Unix environments also.

Networking

For the Internet and the WWW to work successfully, computers around the world have to be able to communicate with one another with ease. Not only for the Internet, but also for the seamless functioning of telephone and other communications systems, computer networks must work flawlessly and reliably without interruption. Networking software and hardware keep the channels of communication among computers around the world working the way they are expected. Perl has built-in facilities for developing

and maintaining networking software. Since programs can be developed easily and quickly in Perl, a large amount of networking software is being written in Perl.

Modules and Libraries

Perl is a large language. It attempts to do many things. Larry Wall, the creator and maintainer of Perl, and many others who have developed software for Perl have created a large set of reusable modules for Perl. There are hundreds of useful and reusable Perl modules that can do any conceivable task. This availability of a large number of reusable software modules is yet another reason for Perl's continued and increasing popularity.

Free Availability

Perl is a language whose interpreter can be obtained for free. This is because Perl's author, Larry Wall, wanted Perl to be free from the beginning. Perl's core modules and documentation have been written by volunteers. It has been ported to many platforms by volunteer programmers. Using the WWW is the simplest way to obtain Perl source code and executables.

Perl's large number of modules and libraries are also available for free. More modules and libraries are being written all the time. The best way to look for any information on Perl is to visit either of the URLs, `http://www.perl.com` or `http://www.perl.org` and start navigating through the linked documents. Another useful site, particularly for Windows systems is `http://www.activestate.com` where a version of Perl called *ActivePerl* is available for downloading along with many modules. The Comprehensive Perl Archive Network (CPAN, `http://www.cpan.org`) is a large archive containing source code, ports, documentation, scripts, modules and extensions. The modules available at CPAN are a superset of those available at `activestate.com`. `www.perl.org` and `www.perl.com` have links to CPAN which is mirrored in many machines around the world. There are also several USENET newsgroups on Perl. They are *comp.lang.perl.misc*, *comp.lang.perl.modules*, *comp.lang.perl.tk*, *comp.lang.perl.announce*, and *comp.infosystems.www.authoring.cgi*, etc.

Purpose of the Book

The purpose of this book is to introduce the student to the rich features of Perl. The text includes a large number of examples of working Perl programs in each chapter. I feel that well-motivated examples with accompanying brief explanations present the best pedagogical style in describing a programming language. A student who reads the chapters of the book, types in the example programs provided and develops his or her own code in parallel, should become an expert in Perl in a short period of time.

The material covered in this book has been used in several classes in the Computer Science Department at the University of Colorado at Colorado Springs. These are **CS 301**: Web Programming, **CS 509**: Bioinformatics, **CS 582**: Artificial Intelligence, **CS 583**: Advanced Artificial Intelligence, and **CS 592**: Applied Cryptography. In CS 301, the material is used for about 6 to 8 weeks, and then referred to throughout the rest of the class. In CS 301, the emphasis is on getting familiar with Perl and write CGI programs. In CS 509, students use Perl to search genome and protein databases, perform pair-wise and multiple alignments in genomic and protein databases. CS 583 focusses on intelligent Internet Systems and the students start programming on the Internet right away. Perl is useful to building systems that fetch materials from the Internet for analysis. Students working on projects such as building search engines of various levels of sophistication, categorizing and classifying material on the Web, personalizing Web pages, learning programs that distill information from the Web, have used the material in this book. In CS 592, network programming and cryptographic programming are discussed.

The material in the book also has been used by professional programmers who are either new to Perl or have used it before, but not for a long period of time. The book is ideal for a professional programmer who wants to get started with Perl without any delay whatsoever. Some of the examples in the book are smaller versions of the programs the author himself has written while consulting with several companies during summers away from academia. For example, the author has used Perl, while working for a company, to write programs that detect spam in incoming email using Bayesian probabilities as well as evolutionary programming. The author used Perl to write code that crawls the Web and creates personalized newspapers. The author has also used Perl to interact with large databases.

In summary, the author recommends this book to college and university students who want to use Perl for programming for class assignments and projects, but do not have access to a class that teaches Perl in depth. It is also for students who want to use Perl for their thesis or dissertation programming in diverse areas such as Web programming, databases, artificial intelligence, networking, bioinformatics, etc. I also recommend the book to those in industry or those who are self-employed and want to use Perl in of these areas, and have to learn it on their own.

This is my first effort at writing a book. Please send any comments or error reports to the author at the address given below.

Jugal Kalita
Associate Professor
Department of Computer Science
University of Colorado
1420 Austin Bluffs Parkway
Colorado Springs CO 80918
Email: `kalita@pikespeak.uccs.edu`
URL: `http://www.cs.uccs.edu/~kalita`

December 20, 2003

Contents

Chapter 1

On The First Steps

In This Chapter

Perl is a massive language. It provides the basic data structures and syntactic constructs most high level programming languages have. In addition, it provides many extras such as a sophisticated pattern matching facility, many operators for handling files and directories, for network programming, and for interacting with the underlying operating system. On top of the large array of facilities the language itself provides, there are many packages and modules that are freely available for doing any task imaginable. It is both the built-in capabilities of Perl and the availability of a large number of well-written modules that have made Perl a popular language.

Perl is a glue language. The many capabilities that Perl provides may seem to be disconnected at first. However, Perl does an excellent job of putting the seemingly disparate parts into a coherent whole. Mastering Perl takes many years of programming experience. However, it is a language that is surprisingly easy to learn. It is not necessary to know all of its syntax to get started or to even write fairly sophisticated programs.

This chapter promptly immerses us in the immensity of Perl. We get introduced to syntactic features of Perl using simple examples. We illustrate the essential elements of the language in real programs, without getting bogged down in details and fine print. In this chapter, we are not trying to be complete or even precise, except that the examples we provide are correct and functional. We want the reader to get as quickly as possible to the point where the learner can write useful programs, and some fairly complex ones. As a consequence, we get into a few complex topics toward the end of the chapter. We feel that familiarity with some complex features in the beginning will excite a learner by increasing his or her level of curiosity. One does not have to be nervous if one does not understand all of the programs and discussions since we present the same issues at length later in the book. Since we do not use the full power of the language, the examples may not be as concise and elegant as they should be. In addition, later chapters necessarily repeat the ideas discussed in this chapter. We hope that the repetition helps one learn better.

A careful reading of this chapter will give the student a sense of accomplishment in a short period of time. Programmers experienced in one or more high level programming languages should be able to extrapolate from the material to their own programming needs. Beginners should supplement it by writing small, similar programs of their own. The best way to learn any programming language is to understand the motivation behind example programs, read them, understand them, sit at a computer, type in the sample examples, and finally code on one's own. That is why the chapters in this book have many example programs.

In this book, when we talk about Unix or Unix-like systems, we not only mean varieties of Unix available, but also include Unix-like systems such as Linux or Macintosh OS X, or simulation of Unix in Windows systems such as `cygwin`. When we talk about non-Unix systems, we mean varieties of Windows operating systems from Microsoft, Inc., and pre-OS X versions of Macintosh OS from Apple Computer, Inc.

1.1 The First Perl Program

We start by learning to write a very simple Perl program. It is a program that we see used as a first example in most programming language texts. Let us assume we want our program to simply write "Hello, world!" onto the terminal. It is as if we come from an alien planet and start the first step in communicating with the human world using a computer. The complete text of the program is given below.

Program 1.1

```
#!/usr/bin/perl
#file hello.pl

print "Hello, world! \n";
```

Technically, this program has only one line or statement of code. The first two lines are comments. # starts a comment in Perl. When Perl finds # on a line of input, it ignores the line starting at the next character till the end of the line.

The first line is needed *only* on a Unix system to indicate where the Perl interpreter is situated in the system. Although, it is not necessary for a comment line to start in the first column, this particular first line must start in the first column. In the particular Unix system where we run the program, the Perl interpreter is `/usr/bin/perl`. The `#!` is used as a convention in many interpreted languages to tell the underlying Unix system where to find the interpreter. If you are using a Unix system and the Perl interpreter is located somewhere else, you will have to find out the full name from your systems administrator and put it in place of the pathname given here.

On non-Unix systems, the first line of comment is unnecessary. However, having the first line of comment doesn't hurt on non-Unix systems. In fact, it is a good idea, especially if you want your program to be portable across platforms.

The second line is also a comment. Of course, this line is not required on any system. This comment line does not have to start in the first column as shown in the sample code above. It says that the program is stored in the file `hello.pl`, and is useful for a human reader.

The only real line of code in this program is the `print` statement. The `print` command, as used here, takes a string as its only argument. The string is started and ended, i.e., delimited by double quotes. The last character in the string is `\n`. If used inside a string enclosed in double quotes, as we do here, `\n`, literally a sequence of two separate characters, means the newline character. In other words, it ends a line of string and starts a new one. Such a two-sequence character with `\` in the front is considered a single unit and is called a *backslashed escape sequence* or simply an *escape character*. Perl allows strings to be enclosed within single quotes also. If a string is single quoted, a backslashed escape character like `\n`, stands for two individual characters and does not make special sense as a two-character sequence with `\` in the front. A statement is terminated by `;` in Perl.

Let us assume that the program is stored, as stated in the comment line, in the file `hello.pl`. Note that Perl doesn't require program files to have any specific extension at the end of the file's name although we have used the `.pl.` extension here. On a Windows system, the extension `.pl` can be associated with the Perl interpreter. We can create the file in any editor with which we are comfortable.

On any system, whether Unix or non-Unix, we can run the program in a terminal by giving the file name as an argument to the `perl` command as given below.

```
perl hello.pl
```

Here, we assume that `%` is the system prompt. There is no need to compile the program since Perl is an interpreted language.

On a Windows system, one can also run the Perl program by by making a choice on a pop-up menu that says something like `Execute Perl script` and then select the name of the file containing the code to run.

To run the program on a Unix machine, one can also do the following. First, we need to make the file executable, i.e., give it `execute` permission for the user. Then, we run the program by typing its name at the system prompt. On a terminal, we type the following.

```
%chmod u+x hello.pl
%hello.pl
```

Note that changing the file's permission to allow execution needs to be done only once, even if we edit the program file later. In a Unix system, the `chmod` command changes the modes or permissions associated with files. The first argument `u+x` asks the system to add the `execute` permission for the user. `u` stands

for user, and x stands for execute permission. The plus sign (+) adds the permission. The last argument to the `chmod` command is the file name `hello`. We are assuming that `%` is the Unix prompt. We don't have to type the `%` sign.
The output of the Perl program is simply

```
Hello, world!
```

as expected.

1.2 Using Scalar Variables

A *scalar variable* is a placeholder or a name whose value has a single component, unlike a more complex variable such as an array variable or a hash variable which can contain many parts or components. We will see arrays and hashes later in the chapter. Perl allows two types of scalar variables commonly used—numbers and strings. A number is either an integer (such as 10) or a float (such as 2.35, -1.414e-10 or 1.414E10). As mentioned earlier, a string can be enclosed either within single quotes or double quotes.
The next program shows us how to use scalar variables in Perl. We use numbers as well as strings.

Program 1.2

```
#!/usr/bin/perl
#file scalars.pl

$day = "Wednesday";
$date = 28;
$month = 'May';
$year = "2003";
$space = " ";

$you = "justin";
$me = 'jugal';

$high = 65.2; $low = 40.3;

print "Dear \u$you,\n\n";
print $space x 5;
print "How are you doing? ";
print "Today is $day, $month $date, $year.\n";
print "Today's high and low temperature were $high and $low degrees F. ";
$tomorrow = $date + 1;
print "Tomorrow is $month $tomorrow, $year.\n";
print "\n\u$me\n";
```

First, we assign values to a number of scalar variables. It is not necessary to declare the variables and their types before we use them. Declaring a variable means telling the system that we intend to use a variable before we actually use it. A variable is usually of a certain type, such as `scalar`, `array` and such. We can assign a value to a variable wherever in the program we find it convenient. All scalar variables in Perl must have names that start with the dollar sign: `$`.

The program assigns values to a number of string variables. The scalar variables `$day, $year, $space` and `$you` have been assigned double-quoted string values. The scalar variables `$month` and `$me` have been assigned single-quoted string values. Double quoted strings must be used if we want the string to contain backslashed escape characters such as `\n`. In this example, the string assignments do not have any such backslashed two-character sequences although the string arguments to some of the `print` commands have them.

We have also used numeric variables. In Perl, although we can use integers and floats, internally all numbers are stored as double-precision floating point values. Here, `$date, $high` and `$low` are scalar variables that have been assigned numeric values. Later in the program, we see a scalar variable `$tomorrow` that has been assigned the value obtained by adding 1 to `$date`.

In this simple program, we see some interesting operations that Perl allows on strings. The first `print` statement.

```
print "Dear\u$you,\n\n";
```

The string argument is a double-quoted string. As a result, it can contain backslashed escape characters such as `\u` and `\n`. `\u` requires Perl to uppercase the next letter it prints. In this case, the next word to print is obtained by *variable interpolation* or by substituting a variable by its value. Since we have the `\u` escape character in front of `$you`, the first letter in the value of `$you` is be uppercased before printing. That is Perl prints `Justin` instead of `justin`.

The second `print` statement is

```
print $space x 5;
```

The `x` is the string repetition operation. Here the value of the variable `$space` is repeated 5 times before printing. Since the value of `$space` is a single blank space, Perl prints five blank spaces.

The rest of the program is straight-forward.

We can create the text of the program using any text editor. Let the file where the program is stored be called `scalars.pl`. The output of this program is given below.

```
Dear Justin,

     How are you doing? Today is Wednesday, May 28, 2003.
Today's high and low temperature were 65.2 and 40.3
degrees F. Tomorrow is May 29, 2003.

Jugal
```

1.3 Declaring Variables: Pragmas

Unlike many other programming languages, it is not necessary to declare variables before use in Perl. Declaring a variable with a certain name simply means stating to the Perl interpreter that the program needs a variable with the given name. It enables the programming system to perform housekeeping such as allocating the appropriate amount of space for the variable. A variable is declared before we actually use the variable for the first time. In Perl, if a variable is not declared before its use, it just springs into existence as soon as it is used in the program in a statement that provides it with a value. If an undeclared and unassigned variable is used for the first time, it assumes a default value such as 0, the empty string, or the empty list, depending on its type and the context of usage.

Declaration of variables helps the Perl allocate space and makes Perl run a bit faster. It also helps write programs that are easier to debug. Perl can be instructed to complain if we use a variable name that has

not been declared before its use. This means that if we make an error in spelling the name of a variable inside the program, Perl catches the error and lets us know. It is considered a good programming practice to declare any variable used in the program. Perl does not enforce pre-declaration of variables. It leaves the choice to the programmer. If a programmer wants the benefits that accrue from declaration of variables, he or she may choose to enforce this discipline on himself or herself. This is consistent with a fundamental belief of Perl's creator that as few rules as possible should be enforced when writing software to stifle creativity. However, many academics and practitioners disagree with such a position arguing that program maintainability is a casualty.

Perl has several ways for declaring variables. The most common way is by using the `my` declaration. It is sufficient for most simple programs. A variable declared with `my` is a so-called *lexically scoped* variable in that it is available only within the block in which it is declared. For simple programs such as the ones we have seen so far, the whole file is assumed to belong to one block. In the following program which is a rewrite of the immediately preceding program, the scope of each variable is the whole program. A block is usually contained inside curly brackets, { and }. We will see examples of such blocks in the next section.

Even when declaration for variable names is enforced, unlike many languages, Perl does not require that all declarations be made at the top of the block, although it is a good practice. In such situations, Perl is happy as long as the variable is declared anywhere inside the block, but before its first use.

Program 1.3

```
#!/usr/bin/perl
#file scalars-strict.pl

use strict vars;

my $day;
my $date;

#Several 'my' declared variables can be put in a list also.
my ($month, $year, $space, $you, $me);
my ($high, $low,  $tomorrow);

$day = "Wednesday";
$date = 28;
$month = 'May';
$year = "1997";
$space = " ";

$you = "justin";
$me = 'jugal';

$high = 65.2; $low = 40.3;
print "Dear \u$you,\n\n";
print $space x 5;
print "How are you doing? ";
print "Today is $day, $month $date, $year.\n";
print "Today's high and low temperature were $high and $low degrees F. ";
$tomorrow = $date + 1;
print "Tomorrow is $month $tomorrow, $year.\n";
print "\n\u$me\n";
```

In this program, each variable has been declared before usage by using `my`. We can declare each variable in a statement by itself. We can also declare several variables in one statement. If we declare several variables in one statement, the list of variables must be included inside parentheses.

In the program given above, we have made it mandatory that every variable used be pre-declared using `my`. We have ensured this by using a *pragma* or Perl directive at the top of the program or the block in question:

```
use strict;
```

A directive requires Perl to look out for things as asked. If we use any variable in this program that is not declared a priori using `my`, Perl will give an error. The pragma could have been written also as:

```
use strict vars;
```

This is because `use strict;` looks for several things, only one of which is ensuring declaration of variable names. The word `vars` can be quoted also.

1.4 Flow of Control

In the programs we have seen so far, the statements in the program have been run sequentially, one after the other. This is the default behavior. The flow of execution of a program can also be controlled by a programmer. One simple way to change a program's flow of control is to organize a sequence of one or more steps into a group or *block* that works a single unit. The block of statements is executed several times, each time performing slightly different computation. We see examples of such control of program flow in this section.

We want to write a program that converts temperatures given in Fahrenheit to Celsius using the standard formula $C = \frac{5}{9} \times (F - 32)$. Here, F is a temperature specified in the Fahrenheit scale, and C is the equivalent temperature in the Celsius scale. The program starts with 0^o Fahrenheit and goes up to 300^o Fahrenheit with a step size of 20^o. For each Fahrenheit temperature, it prints the corresponding Celsius temperature. Each temperature pair is printed in a line. The output of the program is given below.

```
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
 80   26.7
100   37.8
120   48.9
140   60.0
160   71.1
180   82.2
200   93.3
220  104.4
240  115.6
260  126.7
280  137.8
300  148.9
```

Here, the first column represents a Fahrenheit temperature, and the second column represents the Celsius equivalent.

1.4.1 The `while` construct

In a programming language, usually there are several ways to achieve the same task just like as humans, we can say the same thing in many ways. In this section, we look at two different ways of performing *iteration*, i.e., performing the same computation several times with possibly different inputs in every iteration.

A program that produces the desired output follows.

Program 1.4

```
#!/usr/bin/perl
#file farenheit.pl
use strict vars;
my ($lower, $upper, $step, $fahrenheit, $celsius);

#print Fahrenheit-Celsius table for 0, 20, ..., 300 degrees Fahrenheit

$lower = 0;   #lower limit of temperature table
$upper = 300; #upper limit
$step = 20;   #step size

$fahrenheit = $lower;
              #loop through the Fahrenheit values
while ($fahrenheit <= $upper){
    $celsius = 5/9 * ($fahrenheit - 32);
    printf "%4.0f %6.1f\n", $fahrenheit, $celsius;
    $fahrenheit = $fahrenheit + $step;
}
```

The program initializes the lower and upper Fahrenheit limits of the table we want to print. Next, it initializes the value by which a Fahrenheit degree is increased in each iteration of the loop to be 20 degrees.

We use a `while` loop in the program. The expression inside parentheses following the `while` keyword gives the condition that must be satisfied for the statements inside the loop to be executed. The condition simply states that the value of `$fahrenheit` must be less than or equal to the value of `$upper` for the body of the `while` loop to be executed. The *body* of the `while` loop contains all the statements enclosed within { and }. The body is an example of a block. Since the entry condition is satisfied, the loop is executed at least once.

In the body of the loop, there are three statements. The first statement computes the Celsius temperature for the Fahrenheit temperature given as the value of the scalar variable `$fahrenheit` using the formula we discussed earlier. The value of the scalar `$celsius` is set to the value computed.

The next statement allows us to print formatted output. That is why it is called `printf`.

```
    printf "%4.0f %6.1f\n", $fahrenheit, $celsius;
```

The first argument to `printf` is a doubly-quoted string that tells Perl what to print and how. It contains formatting directives that start with `%`. There can be any number of arguments following the first argument. Here we have two such arguments: `$fahrenheit` and `$celsius`. The first formatting directive inside the first argument is `%4.0f`. The `f` at the end tells us that we are printing a floating point number using a decimal point. There are 4 digits before the decimal point and none after the decimal point. The value of `$fahrenheit`—the second argument to the `printf` command— is printed according to the first formatting directive. The value of the third argument to `printf` is printed according the `%6.1f` directive.