# Perl Programming

Palani Karthikeyan

# Introduction to Perl Programming

# Module 1

# Perl

**P**ractical **E**xtraction and **R**eporting **L**anguage

## What is Perl?

- Perl is a general-purpose programming language, and can be used for practically any programming task any other high-level language can be used for. However, Perl is usually thought of as a “glue” language, so called because it binds things together (such as tying databases to Web pages, converting files from one format to another, and so on).
- Perl is very flexible and is currently available on over two dozen operating system platforms

## Perl

- The name Perl comes from “Practical Extraction and Report Language”. Perl has many features borrowed from other programming languages.
- The Perl system uses an interpreter, called “perl”.  Usually Perl and perl are considered to be the same thing for practical purposes.

## Installing Perl

## Versions of Perl

- The current versions of Perl are all in the 5.X and 6.X series (6.X was released in 2001). If you have an older version of Perl (such as Perl 4.X), you should upgrade it as many changes were made between releases.
- Perl 4.X was a very buggy release of Perl and should not be used. Also, many Perl programs designed for 5.X will not work with 4.X.

## Maybe Perl is already installed

- Many operating systems (Linux and UNIX notably, but also Windows NT Resource Kit) come with Perl installed. You can easily check whether Perl is loaded on your system by opening a console or terminal window and issuing the command:

  ```
  perl –v
  ```

  If you get a version number, Perl is installed.

  If you get an error message about command not found (or something similar), Perl is not installed.

## Where to get Perl

- Perl is available free of charge from many public sites.There are several releases of Perl available for different operating systems, so make sure you get a current release.
- For Linux or UNIX systems, visit perl.com for the latest releases
- For Windows systems, you can compile the Perl source code yourself (a hassle) or download a preconfigured Windows release at activestate.com
- For Macintosh, visit macperl.com for MacPerl

# Perl documentation

- Every release of Perl comes with documentation in a set of files. Most releases have over 1,700 pages of documentation included in reference books, user guides, FAQs, and so on.
- On most operating systems, a utility called perldoc is installed as part of the Perl system. The perldoc utility can search for and format Perl documentation for you. To use perldoc to look up the basic syntax for perl, open a terminal or console and issue the command:

```
perldoc perl
```

# More on perldoc

- The Perl documentation is divided into parts by purpose:
  - perlfunc (Perl functions)
  - perlfaq (Perl FAQs)
  - perlop (Perl operators)
- To search for a particular keyword, use the –tf options. For example to look up the print keyword:

```
    perldoc -f print
```

- To search the FAQs use –q as an option:

```
    perldoc -q free
```

## A first Perl program

## What you need

- When you have installed Perl on your system, all you need to use the language is a text editor that can save ASCII files.  All Perl scripts are written and saved in ASCII characters.
- On some operating systems that do not have a Perl GUI front end, you will need to use a console or terminal window to interact with Perl.  Some GUI-based Perl front ends are available for Linux, UNIX, Macintosh and Windows.

## Comments in Perl

- All comments in Perl are written starting with a # sign.  Anything after the # sign through to the end of the line is ignored by the interpreter.
- Comments can be placed anywhere on the line, but commands cannot follow a comment on the same line
- Multiline comments should have a # symbol as the first character on every line

## The #! directive

- The sole exception to # indicating a comment is on the first line of a Perl program (or “script”). All Perl programs can begin with the line:

  ```
  #!/usr/bin/perl
  ```
- The #! is a hold-over from UNIX that instructs the operating system to use the /usr/bin/perl program to run whatever is in this file
- The path may be different for your system, and many environments such as Windows do not need this line. However, it will not cause errors.

## Semicolons

- All valid Perl command lines end in semicolons. Without a semicolon, Perl continues to read onto the next line and doesn’t assume a carriage-return is the end of a statement.
- You can break Perl commands over multiple lines because of this, as long as a semicolon is the end character in the complete statement.
- Perl uses semicolons in the same way as C/C++ and Java

## Whitespace

- Whitespace is ignored by the Perl intepreter. You can use whitespace (spaces and tabs) anywhere in your programs to make them more readable.
- You should use whitespace to help format your scripts to show loops, logic layout, and continuation of statements, as you will see later in this course

## The print command

- The print function tells Perl to display whatever follows, such as a string, variable name, and so on. You’ll see how to build complex print statements later.
- The print statement allows the C or Java escape characters to be used for line feeds, backspace, tabs, and so on. For example, the command:

```
print “Hello\n”;
```

will print “Hello” followed by a newline.

## A Hello World script

- We can write a simple Perl script for the traditional Hello World application:

```
#!/usr/bin/perl
print “Hello World!\n”;
```

- These two lines can be save in a file as ASCII and then run by perl by issuing the command:

```
perl filename
```

## Perl scalars

## Scalars

- Scalars are the Perl term for basic units, including strings and numbers of different forms, as well as constants (which are often called “literals”)
- There are several types of data supported by Perl, and you will see most of them in this and the next module

## Numeric Scalar Variables

- Perl uses the dollar sign to indicate scalar variables, followed by the name of the variable. For example:

  ```
  $date
  ```

  is a variable called “date”. The dollar sign is a type identifier that tells Perl this is scalar. Arrays use a different identifier, as you will see later.
- Variable names are case sensitive, so $Date and $date are different variables

## Strings

- String types in Perl are like those in other programming language. Strings are treated literally when enclosed in quotation marks (either single or double). Escape sequences can be used with Perl strings. These are the most common:

| | |
|---|---|
| – \n | newline |
| – \r | carriage return |
| – \t | tab |
| – \b | backspace |

## Special escape sequences

- Some escape sequences for strings have special meaning to Perl:

| | |
|---|---|
| – \l | change next character to lower case |
| – \u | change next character to upper case |
| – \’ | literal single quotation mark |
| – \” | literal double quotation mark |
| – \\ | backslash |

# The q and qq operators

- Perl allows you to use these structures:
  - q( )
  - qq( )

  Instead of single and double quotes, respectively. So,

  ```
  qq(This is a test)
  ```
  is the same as
  ```
  "This is a test"
  ```
- These can be handy when embedding marks that would otherwise need escaping:

  ```
  qq(He said "Help!" then "Now!")
  ```

## Single and double quotes

- Double quotation marks allow expansion of variables within them. Single quotes do not.
- For example:

```
"This is from $name1";
```

  is not the same as

```
'This is from $name1';
```

  as the second will literally display '$name1" which the first will substituted the value in the variable name1.

## Declaring variables

- Unlike many programming languages, variables do not need to be declared prior to use with Perl. When the variable is assigned an initial value, Perl can figure out the data type.
- If you try to use an uninitalized variable, Perl will use the value zero for a numeric, or Null for a string.  Avoid uninitialized variables as much as possible, as results can be unpredictable.

# Assigning values

- Variables are assigned values using the equal sign:

```
$string1="This is a test";
$var1=6;
$var2=3.14159;
```

- You can assign operation results, as you would expect:

```
$var3=$var2 + $var1;
```

## The $_ variable

- The $_ variable is used by Perl as a default variable. You can use it in place of variable names in your scripts:
  ```
  $_="This is a test";
  print;
  ```
- This will print the default variable $_ and display the string. Use the default operator carefully as it can easily be confusing, but some operators and functions work best with default variables, as you will see later in this course.

## Perl operators

## Standard operators

- Perl supports the standard mathematical operators +, -, *, /, % (modulus) and ** (exponent)
- Operator order of precedence applies according to standard rules; parentheses group operations
- Operators can be combined in statements:
  ```
  $num1=$num2 * ((3 + 8)*$num3/2);
  ```
- Multiple assignments can be made on one line:
  ```
  $num1=$num2=$num3=7;
  ```

## Exercise

- Write a program that simulates rolling five dice. Assign values between 1 and 6 to five different variables and display all five on the screen as well as the sum of the five numbers.  Later you'll see how to use random numbers, but for now just assign the values.
- If you want to display a string and a variable together in a print statement, separate them with periods:
```
print "The sum is " . $sum;
```
  You'll see this in the next few slides.

## Positive and negative

- Numbers are assumed to be positive unless you specify a negative sign in front:

```
$num1=6;          # positive
$num2=-6;         # negative
$num3=-(-6);      # positive
```

- You can convert positive to negative any time using the minus sign in front of the variable:

```
$num4=-$num4;
```

## Increment and decrement

- Like C/C++ and Java, Perl supports autoincrement and autodecrement operators, which increase or decrease a value by one:
  ```
  $var1++;
  ```
  is the same as
  ```
  $var1=$var1+1;
  ```
  and
  ```
  $var2--;
  ```
  is the same as
  ```
  $var2=$var2-1;
  ```

## Shortform assignment

- As with autoincrement and autodecrement, Perl supports shortform assignments like this:
  ```
  $var1+=5;
  ```
  which is the same as
  ```
  $var1=$var1 + 5;
  ```
- This can be performed for all four basic mathematical operators.

## Operators and strings

- Strings can be used with the “.” operator for concatenation:
```
$str1=“Hello ”;
$str2=“World!”;
$str3=$str1 . $str2;
print $str3;
```
  would print “Hello World!”
- You can also concatenate on output in most cases by specifying the string variables together:
```
print $str1 . $str2;
```

## The x operator

- The repetition operator, x, is used with strings to indicate a number of repeats for a string.  For example, the code:
  ```
  $str1= "x" x 20;
  ```
  will string 20 “x”s together and assign them to $str1.
- You can use the repetition operator with existing strings:
  ```
  $str1="x";
  $str2=$str1 x 20;
  ```

## Other operators

- Perl supports several other operators:
  - int: returns the integer portion
  - cos: cosine
  - sin: sine
  - rand: random number between 0 and argument
  - length: length of argument
  - lc: converts to lowercase
  - uc: converts to uppercase

## Exercise

- Rewrite the last program to randomly assign a number to each of the five dice.  To use rand, you need to specify the upper limit: rand(5) will generate a number between zero and 5.  Remember all random numbers have a lower limit of zero.  Have the program roll five dice and display results between 1 and 6, as well as the sum of the dice.

## Converting strings to numbers

- Perl is flexible when using string types as numbers, as long as the conversion makes sense. For example, this works:

  ```
  $str1="6";
  $num1=10-$str1;
  print $num1;
  ```

  will display the value 4. Perl can convert the string to a number if the string looks like a number. This applies to decimal strings as well.

# Converting numbers to strings

- Perl can also convert numbers to strings when the conversion makes sense:
  ```
  $num1=3;
  $str1="I did it" . $num1 .
  " times";
  print $str1;
  ```
  will display the message "I did it 3 times."
- If the conversion doesn't make sense to Perl, it will use a zero instead when you try to call the number.

## Code blocks

- Perl statements can be grouped together into blocks, each block surrounded by braces (like with Java)
- Code blocks can be nested many deep
- Each code block is treated as a unit by Perl, although execution is still always top to bottom unless moderated by control structures
- Usually blocks will be associated with other statements, such as if conditions

# Exercise

- Modify the last program to display strings explaining that you are going to throw the five dice, generate five numbers, show them one at a time like this:

  ```
  The first dice was X.
  The second dice was Y.
  ```

  and so on, and display the sum on a separate line.

## Module 2

## Control Structures

## Comparison operators

- Perl supports the standard comparison operators:

| | |
|---|---|
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |
| == | exactly equal to |
| != | not equal to |

# True and false

- In Perl, any condition that evaluate to false is assigned a value of zero.  Anything that is non-zero is true.  This applies to conditions in statements (such as the if you’ll see in a moment) as well as for numeric evaluation:

| | |
|---|---|
| 0 | false |
| 3 | true (non-zero) |
| 5-5 | false (evaluates to zero) |
| 0.00 | false (it’s zero with precision) |
| “” | Null string is false |
| “ ” | String with a space or anything not null is true |

## The if statement

## The if statement

- Perl's if statement is similar to those of other high-level languages:

```
    if (condition)
    { do if true }
    else
    { do if false }
```

- The blocks of code can contain many statements. The else and its statements are optional.
- Only one block of code is executed, depending on whether the condition is true or false

## Example of if

- A simple if statement is:
```
if ( $num1 > 5 )
{print “It is greater than 5.”;}
else
{print “It is less than or equal
to 5.”;}
```
- If there is only one statement in a block, you can leave the curly braces off as long as a semicolon is used, however many programmer use curly braces to make the code more readable

## Exercise

- Write a program that rolls six dice, all with values between 1 and 6. Add up the result of the die. If the total is greater than 20, display a message that the user wins the game. If the total is not greater than 20, they lose.
- Modify the program to determine if the total die roll is odd or even and display the result

## Nested if-elses

- The if statement can be nested within other if statements, either inside the blocks or as part of the else:

```
if (cond1)
{ if (cond2)
     {statements}}
else { if (cond3)
          { statements}
          else
          {statements}
       }
```

## Exercise

- Write a program that generates three random numbers between 1 and 10. Display a message telling the user whether the sum of the three numbers is greater than, less than, or equal to 15.  Also, tell the user whether the sum is even or odd.

## Reading input

## Input from the keyboard

- The easiest way to get input into a Perl program is to read from the keyboard. To do this, use the <STDIN> structure (the standard input, or STDIN, is the keyboard by default).
- You use <STDIN> to read a value and store it in a variable like this:

```
    $var1=<STDIN>;
```

## The chomp operator

- When reading input from the keyboard, the entire string entered by the user, including the RETURN is saved in the assigned variable
- If you want to eliminate the newline character at the end of the input, use the chomp operator to remove it:
  ```
  $str1=<STDIN>;
  chomp $str1;
  print “You said ” . $str1 . “.”;
  ```
  without the chomp operator, the print statement would have moved down a line at $str1

## Exercise

- Write a program that randomly chooses a number between 1 and 100. Let the user enter a guess, and tell them whether they got the number correct, or whether the random number is higher or lower than the guessed number, and by how much. Only do this guess once.

## String relationships

- There are relational operators for strings (actually, for non-numbers):

| | |
|---|---|
| eq | equal to |
| ne | not equal to |
| gt | greater than |
| lt | less than |
| ge | greater than or equal to |
| le | less than or equal to |

- Comparisons are left-to-right, using ASCII values

## Example of string comparisons

```
$str1="Hello";
$str2=<STDIN>;
chomp $str2;
if ($str1 eq $str2)
   {print "You guessed the
string!\n";}
else
   {print "Wrong guess!\n";}
```

## Exercise

- Write a program that creates three different variables, all with names of animals assigned to them (one animal per variable). Display the three animals for the user. Have a random number generator pick one of the three animals, and ask the user to guess which animal was chosen. Let the user know whether they guessed correctly or not.

## Booleans

# Boolean operators

- Perl supports to Boolean AND and OR operators in the same way as other high-level languages:

| | |
|---|---|
| && | AND |
| \|\| | OR |

- These operators are often used for conditions:

```
if (($num1 < 10) && ($num1 > 5))
```

- Perl also allows the use of the words “and” and “or”:

```
if ($num1 < 10 and $num1 > 5)
```

## The NOT operator

- Perl allows the negation NOT operator to be specified either as “!” or as the word “not”:

```
if (($x < 5) && !($x > 0))
if ($x < 5 and not $x > 0)
```

- Boolean conditions are always evaluated left to right. Keep in mind some complex combinations may not make sense. Check the logic carefully with compound statements.

## Exercise

- Write a program that asks the user for a number between 1 and 100. Check to see if the number is even or greater than 10.  If it is, display a message to that effect.  Also check to see if the number is between 15 and 25 and display a message showing the result.

## Shortform ifs

## The shortform if

- A shortform if statement is popular in Perl, and is applicable if there is only one statement that would reside inside the code block.  Instead of writing:
  ```
  if (condition)
  {statement;}
  ```
  you can write:
  ```
  statement if (condition);
  ```
- This may look confusing, and many programmers do not use it, but it is legal

## Example of shortform if

- This is an example of the shortform if statement:
```
$flag = 1 if ($num1 > 10);
```
- This is the same as writing
```
if ($num1 > 10)
{$flag = 1;}
```
- Shortform ifs can only be used when a single statement is to be executed

## The elsif construct

- Instead of using if-else structures to nest ifs, you can also use the shortform elsif. This removes the need for a set of curly braces for the else. Instead of writing:

  ```
  else { if (cond) {statements…}}
  ```

  you can write:

  ```
  elsif (cond) {statements…}
  ```

- The use of elsif simplifies the number of braces, but some find it awkward to read easily

## Exercise

- Write a program that asks the user for the outside temperature in degrees Fahrenheit. Display the equivalent in degrees Celsius. The conversion formula is:

  ```
  F=32+9C/5
  ```

  where F is degrees Fahrenheit and C is degrees Celsius. Then, if the temperature is going to be below 40F tell the user to take a coat. If the temperature is above 80F tell them to avoid sunburn. If it's in between, tell them it will be a great day!

# Module 3

# Looping

# Perl loops

- As with all other high-level languages, Perl supports loops. The for and while loops are similar to those in languages like C/C++.
- Loops allow you to run a block of code as many times as you want, as long as some condition evaluates to true. Loops always have some condition attached to them.

## The for loop

# The for loop

- The for loop in Perl is similar to that in C/C++ and Java. It consists of three components:
  `for (initial;condition;increment)`
  where initial is the code to run prior to starting the loop, condition is tested prior to each loop and must be true for the loop to continue, and increment is performed after every loop.
- The three parts of the for loop must exist (although they can be empty) and are separated by semicolons

## The parts of the for loop

- The initial statements in a for loop are always executed, whether the condition is true or not.  The initial statements are executed prior to the condition being examined, and are never executed again (unless inside a larger loop)
- The continuation condition is evaluated prior to the code block execution, and revaluated each loop.  At the end of the execution of the code block and prior to the condition being tested, the increment conditions are executed.

## Example of a for loop

```
for ($x=1; $x<=10; $x++)
  {
  print $x . “\n”;}
```

- This program will print the numbers from one to ten in a column and then exit. The initial condition sets the value of $x to 1, each loop increments it by one, and the condition is true until x in incremented to 11.

## Exercise

- Write a program that throws six dice, generating random numbers between one and six.  Display each number as it is thrown, and after six throws, show the sum. Use a for loop to throw the dice six times, using the same variable for each throw.

## The while loop

## The while loop

- The while loop uses a condition to test whether to loop or not. When the condition is true, the loop executes. When the condition is false, the loop terminates. The syntax is:
  ```
  while (condition)
  {statements…}
  ```
  The condition can be anything that evaluates to true or false.

## Example of a while loop

```
 $num1=6;
while ($num1 < 12)
{
print $num1;
$num1++;
}
```

- This loop will print the values of $num1 starting at 6 and going through to 11. The loop fails when $num1 is incremented to 12.

## Exercise

- Modify the last program you wrote to use a while loop instead of a for loop.

## The last and next statements

## The last statement

- The last statement can be used to terminate a loop, regardless of the condition's value. The last statement is similar to break in other programming languages.  The last statement is usually used with some condition:

```
while ($x < 10)
{  if ($x == 6) {last;}
     statements…}
```

- In this case, when $x is exactly 6, the last statement terminates the loop and execution continues past the closing curly brace.

# The next statement

- The next statement causes execution of a loop to restart. It is similar to the continue statement in some languages. Any statements below the next statement are not executed.  The next statement is usually used with a conditional:

```
while ($x < 10)
{  if ($x == 6) {next;}
     statements…}
```

- When the next is encountered when $x is 6, the loop is restarted again, and the statements below the if are not executed on this pass

## Exercise

- Write a program that prompts the user for numbers, adding the numbers the user enters together as they are entered. If the user enters the number 13, terminate the loop at that point and display the total so far. If the user enters the numbers 9 or 18, ignore the numbers in the running total, but continue asking for other numbers.

## Code block labels

## Labels

- Blocks of code and the for and while statements can all be labeled, and identified by that label. To use a label, place it before the statement block followed by a colon:
  ```
  BLOCK1: {statements…}
  ```
- The name of the label can be any valid ASCII characters.  The convention is to use uppercase for labels so there is no conflict with existing keywords.

## Labels and loops

- When you are labeling a loop, use the same convention as a code block:
```
OUTERLOOP: for ( …;…;…)
BIGCOND: while (cond)
```
- Labels allow you to specify which loop is affected using last and next:
```
 if (cond) { last OUTERLOOP;}
```
  will terminate the loop called OUTERLOOP, even if the statement is nested levels deep inside OUTERLOOP

## The exit statement

## The exit statement

- The exit statement is used to terminate a Perl script at any time. Whenever the exit statement is encountered, the script is terminated. You can send a return status code back to the calling program with exit, if you want, by appending the return code after the exit statement:
```
if ($val1 == 0) { exit 0;}
```
- This will exit the program with a return code of 0 is $val1 is zero.

## Exercise

- One of the traditional exercises to show loops is finding prime numbers. Write a program that displays all the primes between 1 and 1,000. Show each prime as it is found. Also show how many primes you found at the end of the program.

## Some Perl functions and operators

## Perl functions and operators

- Perl is a very flexible language. As you will see, there are several ways to accomplish the same tasks in Perl, sometimes quite easily if you know all the functions or operators.
- There are several useful functions that work with scalars and strings, and we can look at these in the next few slides
- A Perl reference book is a handy resource for looking up functions and operators

## The index and rindex functions

## The index function

- The index function is used to find one string inside another. For example, if you have the string “A stitch in time” and want to find out if the string “itch” occurs inside it, you could use the index function.
- The syntax of index is:
  ```
  index string, search_string;
  ```
  where string is the string to be examined for search_string. Of course, you can use variables for either string component.

# Perl functions and parentheses

- Perl functions can be written with or without parentheses in most cases, so the statements:

  ```
  index string, search_string;
  ```

  and

  ```
  index(string, search_string);
  ```

  are identical. It is up to you whether you use parentheses.

## Example of index

- If index finds the substring, it returns the position of the start of the substring in the string, counting from zero. If not found, -1 is returned.
- To find “itch” in “A stitch in time”, you would issue the command:
  ```
  index “A stitch in time”, “itch”;
  ```
  or you could use variables:
  ```
  $str1 = “A stitch in time”;
  $str2 = “itch”;
  $foo=index $str1, $str2;
  ```
  which will return $foo a value of 4.

# Modifications of index

- You can specify a starting position for the substring search, allowing you to skip a known substring if you want. To specify a starting position, give the number after the substring:
  ```
  index $str1, $str2, 6;
  ```
  this will start looking for $str2 starting at the 6th position of $str1.
- This could also be written:
  ```
  index($str1, $str2, 6);
  ```

# The rindex function

- The rindex function is the same as index, except it starts at the right of a string and works towards to left. As with index, it will return the position of the first match, or –1 if no match is found:

  ```
  $str1 = “A stitch in time”;
  $str2 = “itch”;
  $foo=rindex $str1, $str2;
  ```

  Again, $foo would have the value of 4.
- You can specify a starting position with rindex in the same way as index

## Exercise

- Prompt the user for a long string, followed by a shorter one. Use both index and rindex to locate the short string inside the longer one and compare the results.

## The printf function

# The printf function

- The printf function is a more talented version of print, and is similar to the printf in other languages like C and C++
- The printf function is a “formatted print” and allows better control of the output from a print statement. The syntax of the printf statement is the same as that in C/C++:
  ```
  printf format, list;
  ```
  where format is the format specifier and list is what is to be formatted.

## The printf format specifiers

- The format specifiers for printf have the general format of:
  ```
  %-w.dl
  ```
  where % is the identifier used for a specifier, - is an optional minus sign used for justification, w is the width of the field, the period is an optional part followed by the number of decimals, and l is the field type. The field type must be specified. Examples are:
  ```
  %20s
  %-6.2f
  ```

## Field types

- The most common field types for the printf statement are:

| | |
|---|---|
| c | character |
| s | string |
| d | integer number (no fraction) |
| f | floating number |

- There are some other field types, but they are rarely used in programming.

## Examples

- To display the variable $num1 with 6 digits total, two to the right of the decimal, you would use a printf like this:
```
printf “%6.2f”, $num1;
```
or
```
printf(“%6.2f”, $num1);
```
- To print leading zeros if the number does not have enough digits, add a zero in front of the width:
```
printf “%06.2f”, $num1;
```

## String examples

- Strings can be displayed with format specifiers, too:
  ```
  printf "%10s", "Tim";
  ```
  This will right-justify "Tim" into a field 10 characters wide.
- To left-justify a string, use a minus sign in front of the specifier:
  ```
  printf "%-10s", "Tim";
  ```

## Multiple variables

- If you are displaying more than one value, you need a format specifier for each. They are read from left to right:

```
printf "%6d %5.4f %3d",$x1,$x2,$x3;
```

- The format specifiers are matched up with the variables in order. Multiple format specifiers can be inside the quotations marks, as can any string:

```
printf "Total is %8.4f", $total;
```

## Exercise

- Modify the prime number program you wrote earlier to search primes between 2 and 1000. List the primes in a column, stacked one on top of each other so the column of digits are correct, like this:

```
001
011
111
   etc…
```

# Module 4

## Lists and arrays, Part 1

## Scalars, lists, arrays and hashes

- The last few modules have dealt with scalars: variables with a single value.  There are times you want to work with a collection of data, all the same type of different types.
- With Perl, data that is not a scalar is one of three different types: list, hash, or array. We’ll look at all three types in this module.

## Perl lists

## Lists

- Lists are groups of scalars and usually reflect multiple values of the same type of data. There are several ways to represent lists, but the most common is to put the list in parentheses, separated by commas:
  ```
  ("Tim", 1234567, 16.65)
  ```
- This list has a string value and two numbers, one an integer and one a float (although you don't care about the numeric types). Each part of this list is a "list element" so there are three list elements here.

## List contents

- Perl allows lists to have as many elements as you would like in them: there is no size limitation. Lists can therefore be very large.
- You can use variables inside lists, too:
  ```
  ("Tim", $var1, $num1, 65)
  ```
- Lists can refer to other lists, holding one list inside another list as a variable

# Creating list variables

- List variables are similar to scalar variables except the $ used as the first letter of a scalar variable name is replaced by an @ sign for array variables. Lists are a type of array. The array is the variable that had a list assigned to it.
- The same rules apply for array variable naming as with scalar variables, although $foo and @foo are two different variables as far as Perl is concerned because the first letters are different

## Assigning lists

- You can assign a list to a variable the same way as any other value. This creates an array variable containing the list:
  ```
  @list1=("Tim", 45, 56.0);
  ```
- When one list contains another list, you use the same syntax:
  ```
  @list2=(@list1, "sample", 5.3);
  ```
- You can combine lists and variables:
  ```
  @list2=(@list1, "sample", $num1);
  ```

## The qw operator

- When you are adding several scalars to a list and don’t want to bother with quotation marks, you can use the qw operator like this:
```
qw(red green 45 $x);
```
- The qw operator has a whitespace separated list in parentheses, and it converts everything to literals (with no expansion of variables, so $x is a literal). The line above is equivalent to:
```
(‘red’, ‘green’, ‘45’, ‘$x’);
```

## Getting elements from an array

- You can retrieve list (array) elements by using their element index value.  All list elements are numbered, starting from zero, left to right.  To access an array element, use the element number in square brackets:
```
@test=(0,1,2,3,4,5);
print @test[3];
```
- This will display the fourth element (because of zero-origin) which will be 3

## A slice of a list

- You can retrieve more than one list element at a time. This is subgroup of a list is called a slice. Specify the list elements you want separated by commas:

```
@biglist=qw(a b c d e f g h);
@vowels=@biglist[0,4];
```

- You can subgroup or slice lists many times using this method

## End of a list, method #1

- You can find how many elements there are in a list by using the special symbol $# followed by the list name. For example:
  ```
  @list1=qw(a b c d e);
  print $#list1;
  ```
  will display the value 4 (counting starts at 0, and there are five elements, so the last is @list1[4].
- Can you use this method to assign the size of the array to a variable, like this:
  ```
  $size=$#list1;
  ```

## End of a list, method #2

- A sneakier way to determine the length of a list is to use a scalar conversion:
  ```
  $size = @list1;
  ```
- $size is a scalar and cannot contain the contents of list @list1. It holds the number of elements in @list1. (This works because of Perl contexts, which you will see later.)

## Exercise

- Create an array that holds the numbers from 1 to 10. Create a slice of that list with a different name that holds all the odd numbers, and another slice of that sublist that holds all the primes. Write a program that displays the last slice, one element at a time. Display the size of all three lists.

## The foreach loop and arrays

## Accessing elements

- You already know you can access a list element using its index number, such as
  ```
  @list[4];
  ```
  Often you will have to move along an entire array, working with each element in the list. You can do this with a for loop:
  ```
  for ($x=1; $x<@list; $x++)
  { print @list[$x];}
  ```
- The @list in the for condition evaluates to a scalar, the length of the list

## The foreach loop

- Perl has an easier setup for moving across a list, and that's the foreach loop. The foreach loop creates an index variable that is equal to each element in the list, left to right. Here's the same code as the previous slide:
```
foreach $x (@list)
{ print @list[$x];}
```
- The variable $x is set to each element in @list in turn, so you don't have to set up the for loop. The foreach loop starts at index 1, so the 0th element is not counted.

## The for and foreach loops

- Perl uses the foreach and for loops interchangably: you can use either in most situations
- However, you will find that most Perl programmers use foreach when working with arrays, and for in other situations.  This makes it easier to keep track of the loops and uses.

## Exercise

- Modify the last program you wrote to use a foreach or for loop to display the list elements in each of the three lists.

## The pop and push operators

## Arrays and stacks

- Perl has many ways of doing things.  Another way to move through a list is to use the pop and push operators.
- You can think of an array as a stack, like a stack of paper.  When you add something to the top of the stack, everything else gets moved down one place. This is called a push operation.  When you take something off the top of the stack, everything moves up one place.  This is called a pop operation. (These terms will be familiar to anyone who knows RPN notation.)

# Using pop

- Perl supports the pop and push operators to work through a list. The list is stacked, one on top of the other, from left to right, so the last entry is on top. Here's an example:

```
@list=("A", "B", "C", "D", "E");
$pick=pop @list;
print $pick;         #prints "E"
$pick=pop @list;
print $pick;          #prints "D"
```

- You can work through the stack to the bottom using the pop operator

## Using push

- The push operator will add things onto the top of the stack:

```
@list=("A", "B", "C", "D", "E");
pop @list;          # "E"
pop @list;          # "D"
$x=pop @list;       #holds "C"
push @list, $x;     #puts "C" back
```

  The trick to using pop and push is imagining a pointer in the list that moves with each operation. Once something has been popped off the stack it is not in the stack anymore unless pushed.

## More push

- You can push anything back onto a stack. For example:

```
@list=("A", "B", "C", "D", "E");
pop @list;          # "E"
pop @list;          # "D"
push @list, "F"; #puts "F" back
```

- The stack will now have ("A", "B", "C", "F")
- You can add more than one item to the stack:

```
push @list, "F", "G", "H";
```

# Pushing arrays

- So far we’ve only pushed scalars, but you can push arrays onto a stack as well:
  `push @list @array;`
- When you push a list onto a stack, the new list is a concatenation of the two
- You can push lists of lists, as well. To do this, the list of lists gets flattened into a single list, then added to the stack.

## The shift and unshift operators

- The push and pop operators work on the top element of a stack. The shift and unshift operators do the same task at the bottom of the stack.
- The unshift operator adds an object to the bottom of the stack, and the shift operator removes an object from the bottom of the stack.
- You can use pop/push and shift/unshift at the same time. This is one way of manipulating arrays and list elements, but not all programmers use this method.

Context

## Context

- Context has to do with the way Perl uses variables and values. The two important types of contexts in Perl are scalar context and list context.
- When assigning variables, the expression on the left of the equals sign determines what context the right side is evaluated in. The differences between scalar context and list context indicate whether values are treated as lists or scalars, as you will see.

## Scalar and list contexts

- Scalar context arises when using scalars:
  ```
  $num1=$num2;
  ```
  since the left is a scalar, the right is treated as a scalar. If the left is a list, the right is treated as a list:
  ```
  @list1=@list2;
  ```
- There are times when mixed contexts arise:
  ```
  $num1=@list1;
  ```
  which has a scalar to the left so it evaluates the right as a scalar and returns the number of items in the list.

# Forcing scalar

- There are times when you want to force a list to be treated as a scalar. The print function actually expects a list as an argument, so the command:
  ```
  print @list1;
  ```
  prints the contents of the list @list1.
- You can force the print command to accept a scalar context like this:
  ```
  print scalar(@list1);
  ```
  which will convert the list to a scalar and display the number of elements in the list.

## Reordering array elements

## Reordering array elements

- You will often need to work with array elements in a different order than they appear in a list. You could code variable storage for each element, but there are better ways in Perl.  To organize a list in some order, there is the sort function.
- The syntax of the sort function is:
  ```
  sort list;
  ```
  where list is the name of the list to be sorted.
- By default an ASCII sort is produced. The original list is not modified.

## Using sort

- You can use sort to order a list, and assign the result to another variable:

```
@let1=qw(A F D w U d F l X H);
@let2=sort(@let1);
```

- It is important to remember that the ASCII sort will result in lower case letters appearing after upper case letters. Also, numbers will sort in a different order than you expect.  For example, 12 will be after 102. To handle these issues, different sorting orders are needed.

## Exercise

- Create an array with a list of names in it, and sort them in ASCII order, displaying the result.

## Changing the sort order

- To change the sort order from ASCII, you need to add an argument to the sort function. The argument is a block of code which contains instructions on the new sort order.
- Inside the block of code, two variables ($a and $b) are used to indicate two elements in the list. The logic returns one of three values when comparing $a and $b:-1 if $a is less than $b, 1 if $a is greater than $b, and 0 if they are equal.

## The “spaceship” operator

- To help with changing the sorting order of numbers, Perl has the spaceship operator “<=>” which performs the comparison of $a and $b. The spaceship operator returns –1 if the left is less than the right, 1 if the left is greater than the right, and 0 if they are equal. Here’s how to use it:
  `@newlist=sort {$a <=> $b} @list;`
- This command will sort list based on the argument values. This works with numbers, not strings.

## Exercise

- To see how the spaceship operator and the sort function work, create a list of random numbers (you can generate it or simply enter one on the keyboard). Then, use the sort function with the spaceship operator to sort the numbers in ascending order, and display the result.

## Converting scalars and arrays

## Scalars to arrays

- Perl allows conversion between scalars and arrays in several ways
- You can convert a scalar to an array using the split function, which requires a pattern to be specified that breaks the scalar into an array based on that pattern. The pattern is surrounded by slashes. For example, to break the scalar “This is a script” into an array based on spaces, you would issue the command:

```
@n1=split(/ /,“This is a script”);
```

## Results of split

- The result of the command:
  `@n1=split(/ /,“This is a script”);`
  would be an array called @n1 that has the list (“This”, “is”, “a”, “script”) as the contents.
- If you don’t specify a pattern to break up the scalar, whitespace is used by default. If you do not specify a variable name as an argument to split, the default variable $_ is used.
- A special pattern is null (//) which breaks the scalar up character-by-character

## Using split

- The split function allows you to break up arrays into smaller parts when combined with foreach:

```
@cpu=("Intel, Pentium", "AMD, Athlon",
"Transmeta, Crusoe");
foreach $x (@cpu)
{ ($company, $chip)=split(/,/, $x);}
```

- This uses the foreach to extract the three list elements in @cpu, and then split breaks those list elements into two parts based on the commas. These are then assigned to the scalars $company and $chip respectively.

## Exercise

- Create an array which holds a list of record albums, consisting of album title, the year it was released (guess if you don’t know) and artist (in that order). Add at least five albums to the list. Then, use the split function and a for or foreach loop to display the artist followed by the title for each list element.

# Module 5

# Lists and arrays, Part 2

## Converting scalars and arrays

## Arrays to scalars

- The split function lets you convert scalars to arrays.  To convert arrays to scalars, use the join function. The join function expects a string and a list as arguments, and joins the list together using the string, returning a string scalar.
- The syntax is:

```
join string (list);
```

# Example of join

- To join a list of letters into a single string, separated by spaces, you would issue the command:
  `$string=join(‵ ′,  (‵A′, ‵B′, ‵C′, ‵D′, ‵E′));`
  The result would be $string set to “A B C D E”. (The parentheses around the arguments are optional as with most Perl functions.)
- The joining string can be a null or any valid ASCII character

## Exercise

- Create an array with a list of numbers from 1 to 10, then convert the array into a scalar using commands to separate the numbers. Use a hyphen to separate the numbers. Use the join function to perform the conversion.

## Split and join together

- You can use the split and join functions together to manipulate strings. For example:
  ```
  $str="Hello";
  $temp=join(' ', split(//,$str);
  print $temp;
  ```
- This will result in the output "H e l l o". The split breaks the scalar into an array based on individual characters, and the join adds the parts back together into a list with spaces between.

## The reverse function

- The reverse function is used with strings to reverse the characters in the string. It returns a reversed string. For example:

```
$str1="A B C D E";
$str2=reverse($str1);
print $str2;
```

will display the string "E D C B A".

## Exercise

- Write a program that prompts the user for a string, and then display it back, reversed, to the user with hyphens between each of the letters in the string.  If they entered “This is a test” you would display “t-s-e-t- -a- -s-i- -s-i-h-T”.

## Hashes

## Hashes

- Scalars are single data entities.  Arrays are ordered collections of data entities. A hash is an unordered collection of data which may contain many sets of data, but there is no “first” or “last”, no “top” or “bottom” item in a hash.
- Hash data is always in pairs
- Hashes are treated much like lists, but the hash variable is used. Hash variables all start with %.

- Some times called **associative arrays**, dictionaries, or maps; hashes are one of the data structures available in Perl.
- A hash is an un-ordered group of key-value pairs.
- The keys are unique strings.
- The values are scalar values.
- Each value can be either a number, a string, or a reference.

## Creating a hash

- A hash is created in the same way as a list:

```
%hash1=("Intel", "Pentium", "AMD",
"Athlon", "Transmeta", "Crusoe");
```

- To help readability, the hash is usually laid out in its declaration:

```
%hash1=("Intel",     "Pentium",
        "AMD",       "Athlon",
        "Transmeta","Crusoe");
```

- This shows the pairing of hash data more clearly

## The => operator

- To make relationships in hashes clearer, Perl allows the use of the => operator, which Perl interprets to mean a double-quoted string and a comma. This:

  ```
  %hash1=("Intel",     "Pentium",
          "AMD",       "Athlon",
          "Transmeta","Crusoe");
  ```

  becomes this:

  ```
  %hash1=(Intel     => Pentium,
          AMD       => Athlon,
          Transmeta => Crusoe);
  ```

## Hash keys

- When you have a pair like this:
  ```
  Intel => Pentium
  ```
  the left side of the pair is called the hash key, and the right side is the value.  The hash key is used to look up the pair.
- All hash keys must be unique. If you have more than one hash key the same, the newer entries will overwrite the older ones in the list. Values do not have to be unique. Having unique hash keys leads to the use of hashes as a quick-lookup structure.

## Locating hash entries

- To locate a specific entry in a hash, we use the hash keys. The key is enclosed in curly braces:

```
%hash1=(Intel     => Pentium,
        AMD       => Athlon,
        Transmeta => Crusoe);
print $hash1{AMD};
```

- This will print the value corresponding to the AMD hash key, which is “Athlon”. Note the use of a scalar $hash1.

## Exercise

- Create a hash with a set of names and phone numbers. Populate the hash with five sets of data. Ask the user which name they would like to look up the phone number for, and use that as the key to display the number.

## Showing all entries with keys

- Instead of using a for or foreach loop to show all entries in a hash, the keys function can be used:
```
keys(%hash)
```
- This will show a list of all the keys. To show keys and values, a for loop can be used:
```
for (keys %hash) {
  print “$_ is the value for
$hash{$_}\n”;}
```
- This uses the default variable show the hash key ($hash{$_}) and the value assigned to it as $_.

## Reversing hashes

- You can use the reverse function on a hash to reverse the pairs, converting the value to a hash key and the hash key to a value. This allows you to look up with either value:
  ```
  reverse %hash;
  ```
- Reversing a hash swaps the hash key and value, not the order of the pairs in the hash
- Be careful when using reverse.  Since all hash keys have to be unique, there is the chance of losing some data if the values are the same prior to reversing.

## Exercise

- Set up a hash that contains a few city names and their corresponding zip codes (guess if you don’t know them). Then, display the names of the cities and ask the user to enter one.  Display the corresponding zip code.  Reverse the hash and display the zip codes, asking the user to choose one.  Display the city for that zip code. You can display the lists of cities and zip codes any way you want, or display both at the same time with the keys function.

## Modifying hash contents

## Adding entries to a hash

- You can add entries to a hash by specifying the key and value:

```
%hash1=(Intel       => Pentium,
        AMD         => Athlon,
        Transmeta => Crusoe);
$hash1{HP} = “PARISC”;
```

- This adds the entry HP as the key and “PARISC” as the value to the hash. Again, we use a scalar for the function.

# Changing values

- You can change values in a hash by reassigning them:
```
%hash1=(Intel       => Pentium,
        AMD        => Athlon,
        Transmeta => Crusoe);
$hash1{AMD} = “Thunderbird”;
```
- This will change the value associated with the key AMD. To change the key, you can add a new entry and delete the old one, or reverse the hash and change the value, then reverse back (which may cause problems).

## Deleting hash entries

- To remove an entry from a hash, the delete function is used with the hash key:

```
 %hash1=(Intel       => Pentium,
         AMD         => Athlon,
         Transmeta => Crusoe);
delete $hash1{Transmeta};
```

- This will delete the hash pair starting with the key “Transmeta”.

## Converting hashes to arrays

- You can convert hashes to arrays and vice-versa easily, since they both have the same list structure. To convert, simply reassign using the proper variable name:

  ```
  %hash=(“A”, “B”, “C”, “D”);
  @array=%hash;
  %hash2=@array;
  ```
- The first line creates a hash, the second converts the hash format to an array format, and the third line creates a new hash with the array’s contents.

## Blanking a hash

- To remove all data from a hash (hash keys and values), you could use the delete function in a loop, but an easier way is to simply redefine the hash. Suppose you want to blank:
  `%hash=(“A”, “B”, “C”, “D”);`
  To blank this hash, just redefine it:
  `%hash=();`
  and the old values are deleted and a hash with no contents is left.

## Sorting a hash

- You can use the sort function to sort a hash. This is easily done by using keys to extract the hash keys and then use sort on those keys:
```
foreach ( sort keys %hash1)
{  print "$_ $hash1{$_}\n";}
```
- In this case, we've just printed the sorted hash, but you could store the results in a new has variable in sorted order

## Testing for hash keys

- You can test to see if a hash key exists, preventing problems with other statements. Most people would assume this would work:
  `if ( $hash{$key})…`
  but it doesn't.
- To check whether a hash key exists, use the exists function instead:
  `if (exists $hash{$key}) …`

# Exercise

- Modify the last program you wrote to:
  - display the current hash, and ask the user if they want to add a new entry.  If they do, prompt for the city and zip and add it to the hash
  - display the hash again and ask if the user wants to delete an entry. If they do, ask for the hash key to delete, and remove that entry
  - perform the same lookups for city and zip as before, but check to make sure the keys exist. If not, display an error message.

## The grep function

## The grep function

- The grep function searches for patterns in an array. The syntax for Perl's grep is:
  ```
  grep pattern, list;
  ```
- For example, to find all the elements with the pattern "day" in the array:
  ```
  @days=qw(Monday, Tuesday, Friday);
  ```
  you would issue the command:
  ```
  @result=grep /day/, @days;
  ```
  which will populate the array @result with the matching elements.

## How grep works

- The Perl grep works by proceeding through an array, one element at a time, and assigning the element to the default variable $_. The pattern to be found is then compared against $_.
- If the pattern is found in $_, the expression is true and the element is returned by grep.
- If the pattern is not found, the element is not returned by grep.

## Hash intersections

## Intersections

- A common task with Perl is finding the intersections of two hashes or arrays.  In other words, find out which elements are in common with two sets of data. We can also find the difference, by finding everything not in the intersection set.
- The find an intersection, the grep function is very useful

# Performing the intersection

- Finding an intersection between @array1 and @array2 is surprisingly simple:
```
%temp=();
foreach (@array1)
{  $temp{$_}=1;}
@intersect=grep $temp{$_}, @array2;
```
- This code starts by setting up an empty hash. The foreach stores each element in @array1 in $_, one at a time, and fills the temp list with them and sets the values to 1 (so it is true). The last line examines @array2 one element at a time and sets it to $_, which is grepped in %temp. If there, it is an intersection element and added to @intersect.

## Finding the difference

- Finding a difference is similar to an intersection but using negation:

```
%temp=();
foreach (@array1)
{  $temp{$_}=1;}
@intersect=grep (! $temp{$_}, @array2);
```

## Exercise

- Create two arrays of numbers, both with some identical and some different values. Display the intersection and difference of the two arrays.  Display the two derived arrays in sorted order.

## Module 6

## File and directories

## File handles

## Filehandles

- In order to work with files, you need to use a filehandle. A filehandle is a variable that acts as a reference between your Perl program and the operating system’s file structure.
- Filehandles contain information about the file, the way the file was opened (read-only, etc), where you are in the file, and some other attributes.
- Every file manipulation in Perl is done through filehandles

# Naming filehandles

- Filehandle variables do not have a special character in front of them like scalars, lists, arrays, or hashes.  For that reason, the convention is to use uppercase for filehandle variables to avoid confusion with Perl keywords.
- Filehandle names can be any combination of characters you want, but descriptive names are often easiest to work with and keep track of

## Opening a file

- To open a file to be read by Perl, you need to use the open function with a filehandle. The syntax for open is:
  ```
  open handle, filename;
  ```
  where handle is the filehandle and filename is the file to be opened, which may include a path.
- An example of using an open function is:
  ```
  open (BIGFILE, "file1.dat");
  ```
- If you do not specify a directory path, the current directory is assumed

## Checking an open

- Normally you will embed an open function inside an if statement to make sure the file was opened properly.  Otherwise, commands later in the program would cause errors.  Here’s a typical setup:

```
if (open(BIGFILE, “datafile.dat”))
{  statements to run }
else
{ print “Cannot open the file!\n”;
  exit 0;}
```

# Using pathnames

- If you use a pathname in the file open command, it should conform to the format of directory paths in your operating system. For example:
  `open(BIGFILE, “D:\data\data.fil”);`
  will work for Windows but will cause problems for UNIX and Linux. This format will cause Perl problems because the \ has to be escaped. To prevent problems, Perl allows you to use UNIX-like slashes for Windows paths, which is correctly interpreted by the Windows Perl interpreter:
  `open(BIGFILE, “D:/data/data.fil”);`

# Problems with paths

- You must use double backslashes for Windows paths because of escaping of the backslash if you are not using forward slashes, but this can cause even more problems. For example,
  ```
  open(BFILE, "D:\data\data.fil");
  ```
  should be written as:
  ```
  open(BFILE, "D:\\data\\data.fil");
  ```
  to escape the backslashes properly.
- You can use both absolute and relative pathnames, as well as Windows' UNC names (such as \\machine\sharename)

## Closing a filehandle

- After you have opened a file and done something with it, you should always close the file. Closing the file lets the operating system know the file is not in use anymore and the filehandle is freed.
- To close a filehandle, use the close function with the handle name:

```
open(BIGFILE, “data.txt”;
statements…
close BIGFILE;
```

## Reusing a filehandle

- If you have one file open with a specific filehandle, and then use the same filehandle in another open command, the first file is automatically closed and the filehandle is opened with the new file.
- This can be used to eliminate the opening and closing of file statements in a program, as long as the files are used sequentially

## Reading files

# Reading from a filehandle

- There are a couple of ways to read from an open filehandle. The most common is to use the file input operator, which is a pair of angle brackets around the filehandle name (just like <STDIN> to read from the keyboard). For example:

  ```
  open(BIGFILE, “data.txt”)
  $line=<BIGFILE>;
  ```
- This will read a line from the file data.txt (referred to by the filehandle and not the name) and store it in $line

## Using the file input operator

- The line
  `$line=<MFILE>;`
  will read a whole line of input from the MFILE filehandle. If there is nothing to read, the value "undef" (for undefined) is returned.
- You can use loops to read through an entire file. To test whether the value undef has been detected, use the "defined" keyword:
  `while (defined($line=<MFILE>))` ...

## A shortcut for reading lines

- Perl allows the code on the previous slide to be shortened. Instead of writing:
```
while (defined($line=<MFILE>))
{print $line;}
```
  you can write:
```
while(<MFILE>)
{print $_;}
```
- This works because the shortform stores the line in the default variable $_. The shortform also checks for end-of-file for you.

## Exercise

- Write a program that reads in a file (pick any file from the directory) and display the contents of that file, line by line.  Make sure the end of file is handled properly, and remember to close the filehandle after you are finished. Prompt the user for the filename to be read.

## Reading into a list

- So far, we read file contents into a scalar, one line at a time. You could assign the lines read from a file to a list just as easily:
  ```
  open (MFILE, “data.txt”);
  @list=<MFILE>;
  close <MFILE>;
  ```
- When using a list or array, the entire file is read in. Each line in the file is assigned as one element in the list. (So the first line is @list[0], and so on.)

## Using lists

- If you need to read a lot of data from a file, it is often easiest to use a list or array to hold the contents, instead of assigning a variable for each line, then processing the contents of the line somehow.
- Since the array or list is just a copy of the file’s contents, any changes made to the array will not harm the original file

## Exercise

- Write a program that prompts the user for a filename, then reads that file in and displays the contents backwards, line by line, and character-by-character on each line. You can do this with scalars, but an array is much easier to work with. If the original file is:

  ```
  abcdef
  ghijkl
  ```

  the output will be:

  ```
  lkjihg
  fedcba.
  ```

## The die statement

# The open or die syntax

- Perl has a command called “die” which is often used with file commands. When the die command is encountered, the program stops executing and shows a message such as:
  ```
  Died at fileopen.txt line 165
  ```
- To use the die command with an open function, you can use this format instead of an if statement:
  ```
  open(BIGFILE, “data.txt”) || die;
  ```
- This is read as “open or die”: if the open is successful, execution continues; otherwise, the die statement terminates the program

# Adding messages to die

- To help decipher program exits from the die command, you can use strings to be shown upon exit. For example:
  ```
  die “File won’t open”;
  ```
  will display the message:
  ```
  File won’t open at foo.txt line 52
  ```
  when the die statement causes termination of the program.
- You can use these messages to embed error codes or strings in likely locations for die statements

## The $! variable

- When an error is recorded by Perl, it stores the error number in a special variable called $!. When examined numerically, $! shows a number, but when examined as a string it shows an error message from the operating system. This can be used as part of the die string:
  ```
  die “Can’t open: $! \n”;
  ```
- This statement will display the message “Can’t open” followed by the error string from the operating system when the die is triggered

## Warnings

- Instead of bombing out of a program using die, you may simply want to issue a warning to the user. You can do this with the warn command:
```
warn “message”;
```
- The warn command will display the error message, but the program keeps running. You can use the error codes with warn:
```
warn “message: $!”;
```

## Exercise

- Modify the last program you wrote to incorporate the die statement to handle file open errors.  You can use a custom message if you want.

## Writing data to a file

## Opening a file for writing

- Before you can write data to a file, it has to be opened specifically for writing. You use the open function to open a file for writing, but then use a redirection operator in the filename component:

```
open(MYFILE, ">bigfile.txt");
open(MYFILE, ">>bigfile.txt");
```

- The redirection operators are the same used by UNIX. ">" overwrites any contents already in the file, while ">>" appends to the end of the file.

## Creating new files

- If the file you instruct open to open for writing does not exist. The file is created for you in the current directory unless a path has been specified.
- If the file does exist, it will be overwritten unless you use the append operator
- Most operating systems treat case in filenames as important, but some do not.  Check with your operating system to see if mixed case filenames are significant, or whether everything is converted to uppercase.

## Writing data

- Writing data to a file is done with the print or printf command, but with the filehandle included. The syntax is:
  ```
  print filehandle data;
  ```
- There is no comma between the filehandle and the data!
- For example, to write a single variable $num1 to the file once it is opened for writing, use the command:
  ```
  print MFILE $num1;
  ```
  assuming MFILE is the filehandle.

## Checking for writes

- You can use a logic structure to make sure a write has been performed properly:

```
if (! print MFILE $num1)
{warn “Can’t write to the file!”;)
}
close (MFILE);
```

- If the data value $num1 could not be written to the file, the warning is displayed. We used a warn here, but you can also use a die statement instead.

## Closing after writing

- It is important to issue a close operation after writing data to a file.  This is because most operating systems don’t write to the file immediately, but buffer the data.  The close operation tells the operating system to commit the changes, and mark the file as not in use.
- If you do not issue a close operation, there is a chance you will lose the data you have tried to write, and may corrupt the file

## Exercise

- Write a program that creates a file called “data.dat” in the current directory. Prompt the user for five numbers, and write them, one at a time, on both the screen and into the file. Close the file, then open it again for reading only, and display the contents on the screen. Handle error conditions that may occur.

## Working with multiple files

## Multiple files

- You can have many files open at once.  The limit to the number of files that can be open (for reading or writing) is usually set by your operating system. There is no intrinsic limit imposed by Perl.
- Often you will want to read one file, line by line, and process the output saving it into another file. This requires two files to be open at once.  Keep track of the filehandles and the process will be simple.

## Exercise

- Create a file that has a series of ten strings in it, all accepted from the user through the keyboard. Then, open that file in read mode, and reverse the order of the characters in each line, saving the reversed line to a new file. Display the completed reversed file when done.

## Binary files

## Binary vs. text

- Binary files are files that have to be translated literally, such as a picture file, a sound file, or a binary file.  Text files are any files that contain records that end in end-of-line characters.
- Some operating systems distinguish between binary and text files.  Unix and Linux do not, but Windows does. Perl can’t tell the difference between binary and text files (it has a Unix heritage).

## Handling text files

- When Perl writes data to a file, it does so in text mode. When the newline \n character is encountered in a string to be written to a file, Perl converts it to the appropriate characters for the native operating system:

| | |
|---|---|
| UNIX/Linux: | ASCII 10 (LF) |
| Windows: | ASCII 13/10 (CR/LF) |
| Macintosh: | ASCII 13 (CR) |

# Handling binary data

- When writing binary data to a file you don’t want Perl converting anything, so you have to use the binmode command with the filehandle to tell Perl this is to be written literally:
  ```
  open(BFILE, ">file1.dat");
  binmode(BFILE);
  ```
- You only need to specify binmode for a filehandle once, until you close the file
- On some operating systems (UNIX/Linux and Macintosh) binmode is ignored as there is no distinction between binary and text files

## File tests

## File tests

- Perl allows the UNIX file tests to be performed. This is usually done in a condition like this:
  `if (-r FILE) {..}`
- The condition has one valid option followed by the filehandle to be tested. Alternatively, you can use a filename or full path and filename instead of a filehandle.

# Valid tests

- These tests are all UNIX tests available to Perl:

| | |
|---|---|
| -B | true if a binary file |
| -d | true if directory |
| -e | true if file exists |
| -f | true if regular file |
| -M | returns age in days since last modification |
| -r | true if readable |
| -s | returns size of file in bytes |
| -T | true if text file |
| -w | true if writable |
| -z | true if file exists but is empty |

## Using tests

- You can use tests to verify files when opening or writing. If you are prompting the user for a filename,  you can check to make sure the file exists or has the correct type of data.  You can also use test to make sure you are not overwriting an existing file.

## Exercise

- Modify the last program you wrote to allow the user to enter both the filename to read and the filename to write, and check to make sure that the file to read exists, and the file to write to doesn't (so you don't overwrite a file). Display messages if the tests fail.

## File and directory manipulation

## Renaming files

- To rename a file, you use the rename command with the old filename and the new filename separated by a comma:
```
rename “a.dat”, “b.dat”;
```
- You use the filenames and not the filehandles, since the file cannot be open when you rename it. You can use die and warn to trap failures of the rename command, as you have seen earlier.

## Deleting files

- To delete a file, use the unlink command with the filename(s) in a list:
```
unlink file1.dat;
```
- As with rename, you can’t use a filehandle because you can’t delete an open file. Again, you can trap false returns from the operating system with die and warn.

## Directories

- Almost all operating systems use a hierarchical structure to maintain files in directories. Being able to read the directory structure is important. Perl lets you do this through directory handles.
- A directory handle is used to read the contents of a directory. You can open a directory handle using the opendir function:

  ```
  opendir handle directory;
  ```

  where handle is the directory handle you want to open, and directory is the name of the directory to be read.

## Directory listings

- Once a directory has been opened with a dirhandle, you can read the directory contents with the readdir function:
  ```
  opendir TEMPDIR, “/temp” || die;
  readdir TEMPDIR;
  ```
- After you are finished with a dirhandle, you should close the handle with closedir:
  ```
  closedir TEMPDIR;
  ```

## Storing directory contents in an array

- You will often want to read the directory contents and store the list for future use. You can assign the contents to an array just as you did with file contents:
  ```
  opendir (MDIR, "/temp") || die;
  @filelist=readdir MDIR;
  closedir MDIR;
  ```
- You could then manipulate the contents of @filelist, which will have one directory line per element with most operating systems

# Changing directories

- To change directories, you can use the chdir command. Changes in directory can be specified absolutely or relatively. For example:
  ```
  chdir ../book;
  ```
  will move up one directory level and down into the directory book.
- If you do not specify a directory argument for chdir, it will change to your home directory (if one is defined by your operating system)

## Creating directories

- To create a directory, use the mkdir command with both the new directory name and the permissions are arguments:
  ```
  mkdir newdir, perms;
  ```
- For example:
  ```
  mkdir temp, 0755 || die;
  ```
  will create the directory temp under the current directory with the UNIX permissions 755
- The directory permissions use the same convention as your operating system. Not all operating systems will use permissions.

## Deleting directories

- To delete directories, use the rmdir function with the pathname to the directory to be removed:
  ```
  rmdir temp;
  ```
- If the directory cannot be removed, a false is returned by the operating system and can be handled with die or warn.
- You can only delete empty directories!

# Module 7

# Functions (Subroutines)

## Subroutines

## Functions and Perl

- Almost every high-level programming language supports functions and Perl is no exception. Simply put, a function is a block of code that can be called by name to perform some task, then return to the calling program. Functions can usually accept parameters and return values.
- You’ve seen a lot of functions so far: print, printf, reverse, sort, open and so on are all built-in functions
- Perl calls user-defined functions “subroutines” or “subs”

## Creating a subroutine

- To create a Perl subroutine, you use the keyword sub followed by the subroutine name and the code for that subroutine in curly braces:

```
sub sub_name
{ statements…
}
```

- Subroutine names do not have any specific character first, and naming rules are the same as other variables in Perl

## Running a subroutine

- To run a subroutine in Perl, you can use one of two syntaxes:
  ```
  subname();
  ```
  or
  ```
  &subname();
  ```
- The ampersand is optional in almost all cases, as is most commonly left off. If you are passing parameters to the subroutine, they would be enclosed in the parentheses.
- Subroutines can call other subroutines, or themselves recursively

## Returning values

- Many subroutines will return a value when the subroutine code has been executed.  This value is usually assigned to a variable or used in a statement like print. For example:

```
sub twoplustwo
{ 2+2;}
$num1=8+twoplustwo;
```

- In this case, $num1 will have the value of 12.  In this case there is no return statement in the subroutine as with other languages. The last expression in the code is returned by the sub.

# Using return

- You can use a return statement if you want to specify what to return from a subroutine or want to have the option of terminating the sub early:

```
sub oddeven
{ return(0) if ($num%2 == 0);
  statements…}
```

- In this case, $num will have been defined outside the subroutine. If the mod 2 of that number is 0, the returned number will be zero and the sub terminates at this point.  Otherwise, the program continues executing the statements.

## Returning variables

- You can return any variable from a subroutine, including hashes and arrays:

```
sub somesub
{ statements…
    return @array;
    statements…}
@rev=reverse(somesub(@array));
```

- In this code, an array is returned from the sub and reversed in the body of the Perl script, then stored in a new array variable

## Exercise

- Write two subroutines. Both use a variable defined in the body of the program. The first sub multiplies the variable by ten, while the second sub divides the number by two. Prompt the user for the number, and then call both subs and display the results from the Perl script (not the subs).

## Arguments

## Arguments

- The code we wrote so far depends on a variable being defined outside a subroutine, and that’s not good because this reduces code reuse. Ideally, subroutines should be stand-alone and portable.
- To pass data to a subroutine as part of the subroutine call is to use arguments.  You’ve seen arguments passed to functions with several built-in functions, such as sort. To pass arguments to a subroutine, you specify them as part of the subroutine call.

## Specifying arguments

- Arguments for a subroutine can be specified in two ways, either with or without parentheses (most Perl functions don’t care about parentheses as you have seen already in this course):
```
sub(args…);
sub args…;
```
- You can only leave off the parentheses if the subroutine has been defined earlier in the script that the subroutine call. This is a common error for programmers who define subroutines at the end of their code.

# Multiple arguments

- The args list can be composed of many arguments, all separated by commas:
```
sub1 arg1, arg2, arg3;
```
- There is no limit to the number of arguments that can be passed to a subroutine, and they can be of any valid type or size. You can mix argument types in the list passed to the subroutine:
```
sub1 $x1, @array1, $x2, @array2;
```

## The @_ variable

- Inside the subroutine, Perl uses the argument variable @_ to hold all the arguments. You can use @_ in your subroutines:
```
sub showarg
{ print join(" ", @_);}
showarg "This","is","a","test" ;
```
- This program passes the four strings to showarg, which uses the join to paste them together with spaces between each one. The print displays the result.
- The @_ variable is not related to the $_ variable

## Accessing each argument

- Inside a subroutine, each argument is accessed as a part of the @_ array using subscripts, as with any array. The first argument will be element @_[0], the second array element @_[1], and so on.
- You can use loops to access each argument using the array element name

# Using names for arguments

- Inside a subroutine, using @_[x] can be awkward and counter-intuitive. Perl allows you to name arguments being passed into the subroutine, just as with other language function arguments, but you have to do it by assigning the @_ array elements at the top of the subroutine code:

```
sub foo1{
($var1, $var2)=@_;
 statements…}
```

- This will assign the first argument to $var1, and the second to $var2, and so on if more are named

## Exercise

- Write a subroutine that expects three numbers passed as arguments, and multiply the three together, returning the result from the subroutine.  Call the subroutines from inside a Perl script that asks the user for all three numbers. Use names for each argument inside the subroutine itself.

## Passing arrays and hashes as arguments

## Passing an array or hash

- In the last section you passed scalars to a subroutine. Passing an array or hash to a subroutine is done in the same way:
  ```
  sub revarray
  { return reverse(@_);}
  @rev=revarray @array;
  ```
- This will pass the array @array to the subroutine revarray, reverse it, and store the result passed back from the subroutine into @rev. The entire array is read in to the subroutine as @_.

## Passing multiple arrays or hashes, Part 1

- You might think you could pass more than one array or hash in the same manner:

  ```
   sub printarrays
  { (@array1, @array2)=@_;
    print @array1, @array2;}
  ```

  however this will not work as @_ will hold the two arrays together and there is no way to break them into two separate arrays in the subroutine. There is no indication where one array ends and the next starts. In this case, @array1 would have the entire @_ contents and @array2 is empty.

## Passing multiple arrays or hashes, Part 2

- There is no easy way to solve this problem of passing arrays directly. However, they can be passed by reference (which is seen in Module 8).
- You can pass one array and a mix of scalars without problem as long as the array or hash is the last assigned element. The scalars will be properly assigned from the @_ array and everything else goes in the array:

```
 sub passarrays
{ ($x1, $x2, $x3, @array1)=@_;
   statements…;}
```

# Exercise

- Write a program that contains a subroutine that accepts one letter and an array of letters. Prompt the user in the main part of the Perl script for the letter, then for five strings to be sent as an array. Inside the script, combine the five elements in the array into a scalar string, using the letter as a joining character, and pass the result back to the script.  Display the resulting string.

## Subroutine prototypes

## Subroutine prototypes

- A subroutine prototype allows you to specify at the top of your code the name of a subroutine and the number of arguments it expects. Then, later in the code after you have used the subroutine by name in statements, you can define the subroutine properly. This allows you to use the shorter forms of calling subroutines, and allows for code reuse more often.
- A prototype’s role is to tell the interpreter what type of arguments are to be used by a subroutine

## Defining a prototype

- To define a subroutine prototype, use the same syntax as when defining the subroutine itself. You do not have to specify variable names, but you can indicate the presence of variables by using the first symbol (such as $ or @) of the variable:
```
sub mysub1 ($ $ $);
```
- This defines a subroutine called mysub1 which will expect three scalars.  The actual code for mysub1 is defined later in the program.

## Exercise

- Modify the last program you wrote so the body of the subroutine definition is at the end of the script. Add a subroutine prototype to the top of the script and try rerunning the program.

## Subroutines and scope

## Scope

- If you have programmed in any other language that supports functions, you have seen scope. Scope refers to the block of code where a variable has meaning.
- If you define variables inside a Perl script they are available to any subroutines inside that script. Variables created inside a subroutine are available outside the subroutine, as well (which is not how most programming languages behave). This is an important difference in language! In this case, the variables have scope in the entire program.

## Keeping it private with “my”

- To help code reuse and portability, you want to be able to define variables inside a subroutine that have no meaning outside the subroutine. This prevents conflicts with other script variables when you move the subroutine to new programs. To define a local variable, you use the my operator:
```
 my $var1;
 my @array1;
```
- Any variable defined with the word my is considered private to the block of code it is defined in. That is where the variable has scope.

## Private variables

- Any variable defined with the keyword my to make it private is released as soon as the code block in which it was defined is terminated
- You can have private and global variables with the same name, and they are treated differently by the compiler:

```
$num1=6;
sub something
{ my $num1;  # different variable
   statements…}
```

- Both $num1s are treated as different in this case

## Strict

## Using strict

- Perl has a keyword called strict. When used in a program, it tells the interpreter to use much more care when evaluating statements and to display warnings and error messages for everything it finds questionable (usually Perl will let you get away with quite a bit before complaining about something). Using strict is a good way to enhance your programming abilities. To do this, simply put:
  ```
  use strict;
  ```
  at the top of your code.

# Module 8

## References

# References

In this last module of the course we will look at references. If you have programmed in C, C++, or other high level languages, you may be familiar with pointers, which are the same as Perl’s references. If you have not seen these capabilities before, the learning curve is a little steep. To help show the abilities of references we have used very simple exercises throughout. This should show the basics of the subject without overwhelming you.

# References

# References

- Many high-level languages like C and C++ have the concept of a pointer, a variable that points to the memory address of another variable. Perl has pointers too, but they are called references.
- References do not hold a variable value, but hold the memory address of another variable. For example, if you have a variable called $num1, you could have a reference called $refnum1 which holds the memory address when $num1 has its data stored.

## What is a reference

- Every variable in Perl has a value assigned in memory. Any value assigned to that variable is contained in that memory. For example, the statement:
  ```
  $num1=10;
  ```
  will have a memory location assigned with the name $num1, and a value of 10 is stored in that memory location.
- A reference is another variable, and has an assigned memory location, but holds the address of the $num1 memory location instead of a value.

# Why use references

- Why bother using a reference to a memory location with a value in it?  There are many reasons when you get into complex coding, but the simplest reason is it allows you to change the variable the reference points to (and hence the value it points to in that variable’s memory location). This is very handy in some programs, as you will see.
- References are especially handy when dealing with arrays and lists

## Creating a reference

- References are created exactly the same way as other variables, and have no special naming convention. To assign a value to the reference, you use the backslash:
  `$refnum1=\$num1;`
- This will create a reference variable called $refnum1which will hold the memory address of the variable $num1. Creating a reference to a variable doesn’t affect the variable in any way.

## Dereferencing

- To use the value a reference is pointing to, you have to tell the interpreter that you don’t want to know the memory address it holds, but the value inside the memory address it points to. This is done with the dereferencing operator. For example:
  ```
  print $refnum1;
  ```
  will print the memory address $refnum1 holds, but
  ```
  print $$refnum1;
  ```
  will print the value in the memory address of $num1 (if that’s what it points to).

## Using references to change values

- You can use dereferencing to change the value of a variable the reference points to. This is done in the same way as looking up the value:
  ```
  $$refnum1=15;
  ```
- This command will change the value in the memory location $refnum1 points to and set the value of 15 there. You have to use two $ signs here: if you had written
  ```
  $refnum1=15;
  ```
  you would be setting the value 15 into the memory location of $refnum1, not the variable it points to.

## Exercise

- Write a program that create a variable with a user-supplied value. Create a reference to that variable. Display the memory address of the reference and the value stored in the dereferenced variable.

## Using reference values

- When a reference has been given a value (a memory location of a variable), the reference can be used with other variables and references.  For example:

  ```
  $num1=10;
  $refnum1=\$num1;
  $refnum2=$refnum1;
  print $$refnum2;
  ```

  will have $refnum1 point to $num1. $refnum2 then is set to the same value, so the last line shows the dereferenced value of $refnum2, or 10.

## References to references

- You can set up a reference to a reference, although you won’t need this type of ability until you get into complex coding:

  ```
  $num1=10;
  $refnum1=\$num1;
  $refnum2=\$refnum1;
  ```

  the last line sets $refnum2 to the value of $refnum1 (the memory location of $num1), and not to $num1 directly.  To dereference $refnum2 here and see the value of $num1, use:

  ```
  print $$$refnum2;
  ```

## Exercise

- Write a program that sets up five scalars filled with numbers supplied by the user. Then, set up a reference variable that points to the first scalar. Display the dereferenced values of that variable, as well as the memory location it uses. Change the reference to each of the other four variables in turn and repeat the display process.

## References to arrays

## References to arrays

- You can set up a reference to an array in the same way as a reference to a scalar. Since the reference holds a memory address, it is a scalar itself and defined with a $:
```
@array1=("1", "2", "3");
$refarray1=\@array1;
```
- The variable $refarray1 will have the memory address of the first element of the array @array1. It does not point to the entire array, just the start of the memory for @array1.

## Dereferencing array references

- To dereference array references, you can reference any element in the array pointed to with the usual element subscript:
  `$$refarray1[2];`
  This shows the value of the third element in whatever $refarray points to.
- If you want to see the whole array, use:
  `@$refarray1;`
- You can see a range of elements, too:
  `@$refarray1[0-3];`
  shows the first four element in the array.

## Exercise

- Write a program that prompts the user for five strings, and save them as elements in an array. Then, set a reference to that array. Use a loop to show each of the elements in that array using the reference, one element at a time.

## References to hashes

## References to hashes

- References to hashes are set up the same way as arrays:
```
$refhash1=\%hash1;
```
- You access single elements in the hash through the hash key, and get the value associated with that key back:
```
$$refhash1{key};
```
- To see the whole hash the reference points to, use:
```
%$refhash1;
```

## Exercise

- Create a hash and a reference to that hash. You can either prompt the user for hash keys and values, or simply hardcode them to save time. Use a loop to display all the values associated with each hash key in the hash. You may have to refer back to Module 5 for the hash functions.

## References and subroutine

## Passing references to subroutines

- One of the strengths of references is the ability to pass references to arrays and hashes to subroutines. This allows more than one array or hash to be passed properly to a subroutine. Since the code:

  ```
   sub twoarrays
  { (@array1, @array2)=@_;…}
  ```

  does not work, as both arrays are joined into one array @_, references provide a way to pass scalars which reference more than one array.

## Passing arrays

- To pass two arrays to a subroutine, you could do this:

  ```
  @array1=(…);
  @array2=(…);
  $refarray1=\@array1;
  $refarray2=\@array2;
  passarray($refarray1, $refarray2);
  sub passarray
  { statements…}
  ```

  and both arrays can be used inside the passarray subroutine by dereferencing the references.

## Exercise

- Create two arrays, one holding vowels and the other holding consonants.  Pass both arrays into a subroutine using references. Inside the subroutine, display all the elements of each array.

## Regular Expressions

## What is a regular expression?

- A regular expression (*regex*) is simply a way of describing text.
- Regular expressions are built up of small units which can represent the type and number of characters in the text
- Regular expressions can be very broad (describing everything), or very narrow (describing only one pattern).

## Why would you use a regex?

- Often you wish to test a string for the presence of a specific character, word, or phrase
  - Examples
    - "Are there any letter characters in my string?"
    - "Is this a valid accession number?"

## Constructing a Regex

- Pattern starts and ends with a / **/*pattern*/**
  - if you want to match a /, you need to escape it
    - \/ (backslash, forward slash)
  - you can change the delimiter to some other character, but you probably won't need to
    - m|*pattern*|
- any 'modifiers' to the pattern go after the last /
    - *i* : case insensitive **/[a-z]/i**
    - *o* : compile once
    - *g* : match in list context (global)
    - *m* or *s* : match over multiple lines

## Looking for a pattern

- By default, a regular expression is applied to $_ (the default variable)
  - if (/a+/) {die}
    - looks for one or more ‘a’ in $_
- If you want to look for the pattern in any other variable, you must use the *bind* operator
  - if ($value =~ /a+/) {die}
    - looks for one or more ‘a’ in $value
- The bind operator is in no way similar to the ‘=‘ sign!! = is assignment, =~ is bind.
  - if ($value = /[a-z]/) {die}
    - Looks for one or more ‘a’ in $_, **not** $value!!!

## Regular Expression Atoms

- An ‘atom’ is the smallest unit of a regular expression.
- Character atoms
    - 0-9, a-Z match themselves
    - . (dot) matches everything
    - [atgcATGC] : A character class (group)
    - [a-z] : another character class, a through z

## More atoms

- \d - All Digits
- \D - Any non-Digit
- \s - Any Whitespace (\s, \t, \n)
- \S - Any non-Whitespace
- \w - Any Word character [a-zA-Z_0-9]
- \W - Any non-Word character

## An example

- if your pattern is /\d\d\d-\d\d\d\d/
  - You could match
    - 555-1212
    - 5512-12222
    - 555-5155-55
  - But not:
    - 55-1212
    - 555-121
    - 555j-5555

## Quantifiers

- You can specify the number of times you want to see an atom. Examples
  - \d* : Zero or more times
  - \d+ : One or more times
  - \d{3} : Exactly three times
  - \d{4,7} : At least four, and not more than seven
  - \d{3,} : Three or more times
    - We could rewrite /\d\d\d-\d\d\d\d/ as:
      - /\d{3}-\d{4}/

## Anchors

- Anchors force a pattern match to a certain location
  - ^ : start matching at beginning of string
  - $ : start matching at end of string
  - \b : match at word boundary (between \w and \W)
- Example:
  - /^\d\d\d-\d\d\d\d$/ : matches only valid phone numbers

## Grouping

- You can group atoms together with parentheses
  - /cat+/ matches cat, catt, cattt
  - /(cat)+/ matches cat, catcat, catcatcat

- Use as many sets of parentheses as you need

## Alternation

- You can specify patterns which match either one thing *or* another.
  - /cat|dog/  matches either ‘cat’ or ‘dog’
  - /ca(t|d)og/ matches either ‘catog’ or ‘cadog’

## Precedence

- Just like with mathematical operations, regular expressions have an order of precedence
  - Highest : Parentheses and grouping
  - Next : Repetition (+,*, {4})
  - Next : Sequence (/abc/)
  - Lowest : Alternation ( | )

## Examples of precedence

- If we represent sequence with a '·'
  - in other words : /abc/ becomes /a·b·c/
    - **/a·b*·c/ matches abc, abbc, ac, etc.**
    - **/a·b·c*/ matches ab, abcc, abccc, etc.**
    - **/(a·b·c)+/ matches abc, abcabc, etc.**
    - **/c·a·t|d·o·g/ matches cat or dog**
    - **/(c·a·t)|(d·o·g)/ matches cat or dog**
    - **/c·a·(t|d)·o·g/ matches catog or cadog**

## Variable interpolation

- You can put variables into your pattern.
  - if $string = ‘cat’
    - /$string/  matches ‘cat’
    - /$string+/ matches ‘cat’, ‘catcat’, etc.
    - /\d{2}$string+/ matches ‘12cat’, ‘24catcat’, etc.

## Remembering Stuff

- Being able to match patterns is good, but limited.
- We want to be able to keep portions of the regular expression for later.
  - Example:  $string = ‘phone: 353-7236’
    - We want to keep the phone number only
    - Just figuring out that the string contains a phone number is insufficient, we need to keep the number as well.

## Memory Parentheses (pattern memory)

- Since we almost always want to keep portions of the string we have matched, there is a mechanism built into perl.
- Anything in parentheses within the regular expression is kept in memory.
  - ‘phone:353-7236’ =~ /^phone\:(.+)$/;
    - Perl knows we want to keep everything that matches ‘.+’ in the above pattern

# Getting at pattern memory

- Perl stores the matches in a series of default variables. The first parentheses set goes into $1, second into $2, etc.
  - This is why we can't name variables ${digit}
  - Memory variables are created only in the amounts needed. If you have three sets of parentheses, you have ($1,$2,$3).
  - Memory variables are created for each matched set of parentheses. If you have one set contained within another set, you get two variables (inner set gets lowest number)
  - Memory variables are only valid in the current *scope*

## An example of pattern memory

```
my $string = shift;
if ($string =~ /^phone\:(\d{3}-\d{4})$/){
   $phone_number = $1;
}
else {
   print “Enter a phone number!\n”
}
```

## Some tricky bits

- You can assign pattern memory directly to your own variable names:
  - ($phone) = $value =~ /^phone\:(.+)$/;
    - Read from right to left. Bind (apply) this pattern to the value in $value, and assign the results to the list on the left
  - ($front,$back) = /^phone\:(\d{3})-(\d{4})/;
    - Bind this pattern to **$_ (!!!)** and assign the results to the list on the left

## List or scalar context?

- A pattern match returns 1 or 0 (true or false) in a scalar context, and a list of matches in array context.
- There are a lot of functions that do different things depending on whether they are used in scalar or list context.
  - $count = @array                    # returns the number of elements
  - $revString = reverse $string   # returns a reversed string
  - @revArray = reverse @array # returns a reversed list

## Practical Example of Context

- $phone = $string =~ /^.+\:(.+)$/;
  - $phone contains 1 if pattern matches, 0 if not
  - scalar context!!!
  - This is why this worked!

    ```
    unless (/^\d+$/){
       die}
    ```
- ($phone) = $string =~ /^.+\:(.+)$/;
  - $phone contains the matched string
  - list  context!!!

## Finding all instances of a match

- Use the ‘g’ modifier to the regular expression
  - @sites = $sequence =~ /(TATTA)/g;
  - think ***g*** for ***g***lobal
  - Returns a list of all the matches (in order), and stores them in the array
  - If you have more than one pair of parentheses, your array gets values in sets
    - ($1,$2,$3,$1,$2,$3...)

## Perl is Greedy

- In addition to taking all your time, perl regular expressions also try to match the largest possible string which fits your pattern
  - /ga+t/ matches gat, gaat, gaaat
  - ‘Doh! No doughnuts left!’ =~ /(d.+t)/
    - $1 contains ‘doughnuts left’
- If this is not what you wanted to do, use the ‘?’ modifier
  - /(d.+t)/  # match as few ‘.’s as you can and still make the pattern work

# Making parenthesis forgetful

- Sometimes you need parenthesis to make your regex work, but you don’t actually want to keep the results. You can still use parentheses for grouping.
- /(?:*group*)/
  - yet another instance of character reuse.
    - \d? means 0 or 1 instances
    - \d+? means the fewest non zero number of digits (don’t be greedy)
    - (?:group) means look for the group of atoms in the string, but don’t remember it.

## Substitute function

- s/*pattern1*/*pattern2*/;
- Looks kind of like a regular expression
  - Patterns constructed the same way
- Inherited from previous languages, so it can be a bit different.
  - Changes the variable it is bound to!

# Using *s*

- Substituting one word for another
  - $string =~ s/dogs/cats/;
    - If $string was “I love dogs”, it is now “I love cats”
- Removing trailing white space
  - $string =~ s/\s+$//;
    - If $string was ‘ATG    ‘, it is now ‘ATG’
- Adding 10 to every number in a string
  - $string =~ /(\d+)/$1+10/ge;
    - If string was “I bought 5 dogs at 2 bucks each”, it is now:
      - “I bought 15 dogs at 12 bucks each”
    - Note pattern memory!!
    - ***g*** means ***g***lobal (just like a regex)
    - ***e*** is special to *s*, ***e***valuate the expression on the right

## tr function

- ***tr***anslate or ***tr***ansliterate
- tr/*characterlist1*/*characterlist2*/;
- Even less like a regular expression than *s*
- substitutes characters in the first list with characters in the second list

```
$string =~ tr/a/A/;  # changes every ‘a’ to an ‘A’
```

  – No need for the ***g*** modifier when using *tr*.

## Using *tr*

- Creating complimentary DNA sequence
  - $sequence =~ tr/atgc/TACG/;
- Sneaky Perl trick for the day
  - *tr* does two things.
    - 1. changes characters in the bound variable
    - 2. Counts the number of times it does this
  - Super-fast character counter™
    - $a_count = $sequence =~ tr/a/a/;
    - replaces an ‘a’ with an ‘a’ (no net change), and assigns the result (number of substitutions) to $a_count

## Perl debugger

## The debugger

- Part of the Perl interpreter is a debugger that you can use to examine the execution of your Perl scripts. The debugger allows step-by-step execution of scripts, examination of variable values, and the use of breakpoint.
- The debugger is built into every Perl interpreter and is activated with the –d option when launching the interpreter:

```
perl –d myprog.txt
```

## Debugger output

- When you first launch a program with the debugger option you will see version information for the perl interpreter, then a help prompt:
  ```
  Enter h or 'h h' for help.
  ```
  then the first line of the script. You will also see a message showing which filename the statement is in, and what the line number was.
- Finally, the debugger prompt
  ```
  DB<1>
  ```
  is shown, indicating the debugger is waiting for your first debug command.

## The statements

- When the debugger shows you a statement, it is in the cache ready to be executed but has not yet been executed.
- Each statement read by the debugger can be examined and manipulated prior to it being run. This allows for some changes or examination of the environment before each statement is executed, which is ideal for debugging the script.
- Any valid Perl command can be used at the debugger prompt

## Debugger help

- You can get help from within the debugger at any time using the h command, usually followed by the command you want information about. For example, for help on the b (breakpoint) command, type:
  ```
  h b
  ```
- The command ‘h h’ shows a summary of the available commands and their syntax
- To page the output from the help system, put a | in front of the command (such as |h h)

# Listing the program

- To list the next ten lines in your Perl script, use the l command. Every time you issue an l command, the next ten lines will be shown.
- Listing the lines does not affect the line that is being executed: it simply shows you the next ten lines of the script. The next line to be executed is shown in the listing like this:
  ```
  ===>
  ```
- You can specify which lines to show by using a range: `l 10-15` shows lines 10 through 15 inclusively in the script.

## Stepping through statements

- To run each line, one at a time, in the debugger, use the n (next) command. Each line is shown on the screen before it is executed.
- To see the value of any variable, use the print command at the prompt:
  ```
  print $var1
  ```
  and the current value will be shown without affecting the program
- You can keep using the n command to step through each line in the program until termination

## Stepping into subroutines

- When a subroutine call is encountered by the debugger in the script, it executes the subroutine as a single call and doesn’t show the lines in that subroutine.  To jump into the subroutine and move through it one line at a time, use the s (step) command.
- When you issue the step command the debugger shows each line, one at a time, executed inside the subroutine and all valid debugger commands can be used inside the subroutine

## Breakpoints

- You can use the n command step through a program line by line, or let the debugger run all the lines until some condition is met. This is a breakpoint and is set with a b command.
- You can set a breakpoint at any line number by specifying the line number.  To set a breakpoint at line 10, you would use the command:
  ```
  b 10
  ```
- The c (continue) command lets you continue executing after a breakpoint has been triggered

## Using breakpoints

- You can set a breakpoint on any line in a script except those that have:
  - Just a curly brace or closing parentheses
  - A blank line
  - A comment
- Usually, breakpoints are used after a loop, subroutine return, or complex command so you can verify the actions taken. You can set a breakpoint anywhere except those listed above.

## Showing and removing breakpoints

- To show all the breakpoints that are set in your script, use the L command
- To remove a breakpoint, use the d command followed by the line number (or the subroutine number, if a breakpoint is set to a subroutine). For example:
  ```
  d 37
  ```
  deletes the breakpoint set at line 37.

## The reset command

- You can reset the debugger to clear all breakpoints and variables, and restart the execution of the script from the top with the R command
- When reset is executed, any defined variables lose their value, and the first line of the script is the to-be-executed line

## GUI debuggers

- The built-in debugger is acceptable for simple tracing and debugging, but is not suitable for very complex debugging tasks.  Also, it is not graphical.
- There are many GUI-based debuggers for Perl available on the market, some bundled with Perl distributions.  The ActiveState Perl distribution has a Windows debugger in the package, for example, and there are several available for UNIX and Linux.

## Split & Join

Split function is used to split a string into smaller sections

Join function is the opposite

Examples:

Split a string into words:

```
s/^\s+//;
@array = split;
```

Or:

```
$line =~ s/^\s+//;
@array = split(/\W/, $line);
```

Split a string into characters:

```
@array = split(//);
```

Split a string based on a delimiter sequence of characters:

```
@array = split(/:/);
```

**There are two different types of errors:**

syntax errors

logic errors

**Syntax errors** are made as you type your script into an editor.

**Example:-**you might not add a closing quote or might misspell a filename.

**Logic errors** are more insidious and difficult to find.

**Example:-** you might place an assignment statement inside an if statement block that belongs outside the block. Or you might have a loop that runs from 0 to 100 when it should run from 10 to 100. Accidentally deleting the 1 or not entering it in the first place is very easy.

**Syntax errors** are usually easy to fix.

**Logic errors** can be very hard to fix

A debugger is an environment that lets you execute your program line by line. This is also called single-stepping through your program

**use -w option**

```
$foo = { };

$bar = 5;

print("$foa\n");

print("$bar\n");
```

You probably can't see anything wrong at first glance. In fact, this program compiles and runs without complaint.

However, running this program with the -w option (perl -w test.pl) results in these error messages:

```
Identifier "main::foa" used only once: possible typo at test.pl line 4.

Identifier "main::foo" used only once: possible typo at test.pl line 1.

Use of uninitialized value at test.pl line 4.
```

With these error messages, the problem becomes obvious. Either the variable name $foo is misspelled in the assignment statement or the variable name $foa was misspelled in the print statement.

## Stepping Through Your Script

If, after using the -w and the strict pragma, you still have a problem, it's time to use the debugger

**What is the debugger?**

It is an interactive environment that allows you to execute your script's statements one at a time.

If necessary, you can display the lines of your script, view or alter variables, and even execute entirely new statements.

You start the debugger by using the -d command-line option. The following line

starts the debugger and loads the script called 08lst08.pl.

```
perl -w -d 08lst08.pl
```

If you want to invoke the debugger with no script, you need to perform a small bit of magic, like this

```
perl -d -e "1;"
```

To start debugger without any program.

The -e option tells Perl to execute a single Perl statement.

In this case the statement is 1;, which basically means do nothing.

It does, however, stop the interpreter from looking for the name of a script file on the command line.

```
main::(08lst08.pl:3):   my($codeRef);
 DB<1>
```

This message tells you that the debugger (DB) routines have been loaded.

The DB<1> is a prompt that indicates that the debugger is waiting for input. The line number inside the angle brackets is the current execution line.

The current execution line is that line that the debugger waits to execute.

One of the features of the debugger is the capability to insert breakpoints into your script.

A breakpoint is an instruction that tells the debugger to stop, to display a prompt, and to wait for input.

When the debugger first starts, there are no breakpoints defined for your program.

Demonstrates the different breakpoint commands you can use.

```
1:     sub a {
2:   my($foo) = @_;
3:
4:   print("This is function a. Foo is $foo.\n");
5:     }
6:
7:     a(10);
 8:     a(5);
```

When the script is first loaded into the debugger, the current execution line is 7. Using

**A transcript of the debugging session might look like this:**

```
main::(16lst04.pl:7):   a(10);

DB<1>c

This is function a. Foo is 10.

This is function a. Foo is 5.
```

You can force the debugger to stop each time that a() is invoked by using the b a command. This lets you examine the @_ parameter array before the function is started.

**For example:**

```
main::(16lst04.pl:7):   a(10);

  DB<1>b a

  DB<2> c

main::a(16lst04.pl:2):      my($foo) = @_;

  DB<3> p @_

10

  DB<4> c

This is function a. Foo is 10.

main::a(16lst04.pl:2):      my($foo) = @_;

  DB<4> p @_

5

  DB<5> c

This is function a. Foo is 5.
```

## conditional break points

You can also create conditional breakpoints.

**For example:-**

you could tell the debugger to stop inside a() only if $foo is equal to 5 using the command b 4 $foo == 5.

In this instance, you can't use b a $foo == 5 because $foo is a local variable.

When the debugger stops just before executing a function, the parameter array is initialized but not any of the local variables.

**A debugging session using conditional breakpoints might look like this:**

```
main::(16lst04.pl:7):   a(10);

  DB<1> b 4 $foo == 5

  DB<2>L

4: print("This is function a. Foo is $foo.\n");

  break if ($foo == 5)

  DB<2> c

This is function a. Foo is 10.

main::a(16lst04.pl:4):      print("This is function a. Foo is $foo.\n");

  DB<2> c

This is function a. Foo is 5.
```

The debugger did not stop during the first call to a() because $foo was equal to 10.

On the second call, $foo is set to 5 which causes the debugger to stop.

The L debugger command is used to display all breakpoints and their conditions.

If you don't specify any conditions, a default condition of 1 is supplied.

Because 1 is always true, this creates an unconditional breakpoint.

**If you had created an unconditional breakpoint on line 7, the L command would display the following:**

```
4:        print("This is function a. Foo is $foo.\n");
 break if ($foo == 10)
 7:     a(10);
  break if (1)
```

The d command is used to delete or remove breakpoints. Issuing the commands d 4 and then L would result in this display:

```
7:      a(10);
 break if (1)
```

If you want to delete *all* the breakpoints at once, use the D command.

## Examples: Creating Command Aliases

The = command is used to create command aliases. If you find yourself issuing the same long command over and over again, you can create an alias for that command.

**For example, the debugger command**

```
= pFoo print("foo=$foo\n");
```

creates an alias called pFoo. After this command is issued, typing pFoo at the debugger prompt produces the same results as typing print("foo=$foo\n");.

You use the = command without any arguments when you want a list of the current aliases.

If you want to set up some aliases that will always be defined, create a file called .perldb and fill it with your alias definitions.

Use the following line as a template:

```
$DB::alias{'pFoo'} = 'print("foo=$foo\n");';
```

After you create this file and its alias definitions, the aliases will be available in every debugging session.

## Perl Module

## What is a Module?

A Perl module is a set of Perl code that acts like a library of function calls.

The term *module* in Perl is synonymous with the word *package*.

Packages are a feature of Perl 4, whereas modules are prevalent in Perl 5.

## What is a Module?

Modules are an important and powerful part of the Perl programming language.

A module is a **named** container for a group of variables and subroutines which can be loaded into your program.

By naming this collection of behaviours and storing it outside of the main program, you are able to refer back to them from multiple programs and solve problems in manageable chunks.

Modular programs are more easily tested and maintained because you avoid repeating code, so you only have to change it in one place.

Perl modules may also contain documentation, so they can be used by multiple programmers without each programmer needing to read all of the code.

The use of modules in Perl allows variables with the same name to be created at different, distinct places in the same program.

The symbols defined for your variables are stored in an associative array, referred to as a *symbol table*.

These symbol tables are unique to a package.

Therefore, variables of the same name in two different packages can have different values.

Each module has its own symbol table of all symbols that are declared within it.

The symbol table basically isolates synonymous names in one module from another.

The symbol table defines a *namespace*, that is, a space for independent variable names to exist in.

Thus, the use of modules, each with its own symbol table, prevents a variable declared in one section from overwriting the values of other variables with the same name declared elsewhere in the same program.

The variables in a Perl program belong to the main package.

All other packages within a Perl program either are nested within this main package or exist at the same level.

There are some truly global variables, such as the signal handler array %SIG, that are available to all other modules in an application program and cannot be isolated via namespaces.

All other symbols names STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, Inc, and SIG are forced to be in package _main.

Switching between packages affects only namespaces.

All you are doing when you use one package or another is declaring which symbol table to use as the default symbol table for lookup of variable names.

Only dynamic variables are affected by the use of symbol tables.

Variables declared by the use of the **my** keyword are still resolved with the code block they happen to reside in and are not referenced through symbol tables.

In fact, the scope of a package declaration remains active only within the code block it is declared in.

Therefore, if you switch symbol tables by using a package within a subroutine, the original symbol table in effect when the call was made will be restored when the subroutine returns.

Switching symbol tables affects only the default lookup of dynamic variable names.

You can still explicitly refer to variables, file handles, and so on in a specific package by prepending a *packageName*:: to the variable name.

Modules can be nested within other modules.

The nested module can use the variables and functions of the module it is nested within.

For nested modules, you would have to use *moduleName*::*nestedModuleName* and so on.

Using the double colon (::) is synonymous with using a back quote (`).

However, the double colon is the preferred, future way of addressing variables within modules.

The symbol table for a module is actually stored in an associative array of the module's names appended with two colons.

The symbol table for a module called Bond will be referred to as the associative array %Bond::.

The name for the symbol table for the main module is %main::, and can even be shortened to %::.

Similarly, all nested packages have their symbols stored in associative arrays with double colons separating each nesting level.

For example, in the Bond module that is nested within the Investment module, the associative array for the symbols in the Bond module will be named %Investment::Bond::.

## typeglob

A typeglob is really a global type for a symbol name.

You can perform aliasing operations by assigning to a typeglob.

One or more entries in an associative array for symbols will be used when an assignment via a typeglob is used.

The actual value in each entry of the associative array is what you are referring to when you use the **variableName* notation.

Thus, there are two ways of referring to variable names in a package:

```
*Investment::money = *Investment::bills;

    $Investment::{'money'} = $Investment::{'bills'};
```

In the first method, you are referring to the variables via a typeglob reference.

The use of the symbol table, %Investment::, is implied here, and Perl will optimize the lookup for symbols money and bills.

This is the faster and preferred way of addressing a symbol.

The second method uses a lookup for the value of a variable addressed by 'money' and 'bills' in the associative array used for symbols, %Investment:: explicitly.

This lookup would be done dynamically and will not be optimized by Perl.

Therefore, the lookup will be forced to check the associative array every time the statement is executed.

As a result, the second method is not efficient and should be used only for demonstration of how the symbol table is implemented internally.

## Another example in this statement

```
*kamran = *husain;
```

causes variables, subroutines, and file handles that are named via the symbol kamran to also be addressed via the symbol husain.

That is, all symbol entries in the current symbol table with the key kamran will now contain references to those symbols addressed by the key husain.

To prevent such a global assignment, you can use explicit references.

For example, the following statement will let you address the contents of

$husain via the variable $kamran:

```
*kamran = \$husain;
```

However, any arrays such @kamran and @husain will not be the same.

To summarize, when you assign one typeglob to another, you affect all the entries in a symbol table regardless of the type of variable being referred to.

When you assign a reference from one variable type to another, you are only affecting one entry in the symbol table.

Modules are the foundation of **the CPAN,** which contains thousands of ready-to-use modules, many of which you will likely use on a regular basis.

## A Perl module file has the following format:

- package Module Name;
  ...
  #### *Insert module code* ####
  ...
  1;
- The filename has to be called ModuleName.pm.
- The name of a module must end in the string .pm by convention.
- The package statement is the first line of the file.
- The last line of the file must contain the line with the 1; statement.
- This in effect returns a true value to the application program using the module.
- Not using the 1; statement will not let the module be loaded correctly.
- The package statement tells the Perl interpreter to start with a new namespace domain.
- Basically, all your variables in a Perl script belong to a package called main.
- Every variable in the main package can be referred to as $main'variable.

# Here's the syntax for references:

$packageName'variableName

The single quote (') is synonymous with the double colon (::) operator.

For the time being, you must remember that the following two statements are equivalent:

```
$packageName'variableName;

$packageName::variableName;
```

The double-colon syntax is considered standard in the Perl world.

Thus, if you are in the package Finance.pm and specify a variable $pv, the variable is actually equal to $Finance::$pv.

## Using Perl Modules: use vs. require

use ModuleName;
require ModuleName;

Note that the .pm extension is not used in the code shown above.

Also note that neither statement allows a file to be included more than once in a program.

The returned value of true (1;) as the last statement is required to let Perl know that a required or used module loaded correctly and lets the Perl interpreter ignore any reloads.

## Exploring Existing Modules

Before you begin writing your own modules, you need to know how to use modules and how perl finds modules to load.

The Perl core distribution comes with many useful modules, so we'll take one called *Digest::MD5*

## An example and see how to use it.

Using your terminal, run the command `perldoc Digest::MD5`.

```
use Digest::MD5 'md5_hex';

 ...

$digest = md5_hex($data);
```

## About the MD5 Digest

The MD5 algorithm is a *cryptographic hash function*, meaning that it reliably generates a fixed-length "fingerprint" given arbitrarily large data as input.

Further, the fingerprint cannot easily be generated from some *other* input, which makes this fingerprint very useful in login systems.

When you implement a login system (such as a user on a website or a multi-user computer), you want the user to know their password, but it's best if that password isn't left lying around somewhere on the computer where others might find it.

A cryptographic hash like the MD5 digest is a way to verify that the user's password is correct without storing it.

You store a *digest* of Jim's password. When Jim tries to login, compare a digest of the password he types to the one you have stored.

Because the digest is irreversible, anyone who gets access to your password file or database will not be able to simply login as Jim.

When your code uses the ***Digest::MD5*** module, you know that it will always create MD5 digests in the same way.

Without modules, different programs might contain duplicate code or various re-implementations of the algorithm.

## Loading Modules

Most module documentation will include an example section like this which shows a typical use statement.

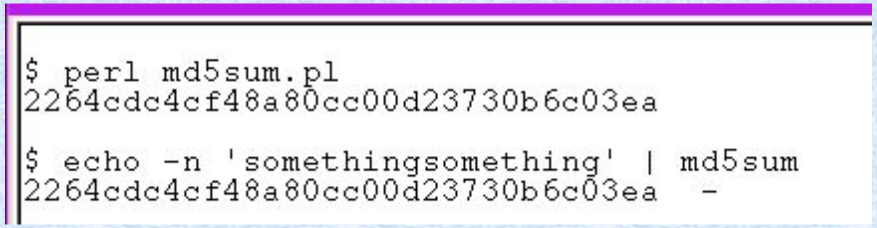
The use command tells perl to find and load the module named *Digest::MD5* and then to invoke Digest::MD5->import('md5_hex').

That import() installs the function *Digest::MD5::md5_hex* into your current program.

Open your text editor and create the following *md5sum.pl* file.

## File:md5sum.pl

```
#!/usr/bin/env perl
use v5.10.0;
use warnings;
use strict;
use Digest::MD5 'md5_hex';
say md5_hex("somethingsomething");
```



```
$ perl md5sum.pl
2264cdc4cf48a80cc00d23730b6c03ea

$ echo -n 'somethingsomething' | md5sum
2264cdc4cf48a80cc00d23730b6c03ea  -
```

When you use a module, perl is following a built-in plan for how to find and load the correct package.

The first thing it does is to search for a *.pm* ("Perl Module") file in the directories listed in the global @INC array.

```
$ perl -E 'say for @INC‘
 /usr/local/lib/perl/5.10.0
 /usr/local/share/perl/5.10.0
 /usr/lib/perl5
 /usr/share/perl5
/usr/lib/perl/5.10
 /usr/share/perl/5.10
/usr/local/lib/site_perl
```

The @INC array is set when perl is compiled, but can be altered by the program, with the -I command-line switch, or by the *PERL5LIB* environment variable.

```
$ PERL5LIB=also perl -Ilib -E 'say for @INC‘
lib also
/usr/local/lib/perl/5.10.0
/usr/local/share/perl/5.10.0
/usr/lib/perl5
/usr/share/perl5
/usr/lib/perl/5.10
 /usr/share/perl/5.10
/usr/local/lib/site_perl
```

When we say use Digest::MD5,

perl starts at the first @INC entry and searches for a *Digest/MD5.pm* file.

Note that module names are always written with '::' separators in code, but on your filesystem this becomes the directory separator (e.g. '/'.)

If there are more *Digest/MD5.pm* files later in @INC, perl will simply ignore them.

This *masking* effect allows you to install a newer version of a module which will take precedence over the one shipped with Perl.

Once perl finds and loads your module, it stores the full filename for the loaded module in the global %INC hash.

Once perl finds and loads your module, it stores the full filename for the loaded module in the global %INC hash. You can look in this hash to see which file was loaded.

```
$ perl -MDigest::MD5 -E 'say $INC{"Digest/MD5.pm"}'
    /usr/lib/perl/5.10/Digest/MD5.pm
```

## Creating a Login System

Use your editor to create the file 'passwords' with a ‘AB' user and our md5_hex output for the password 'somethingsomething'.

Now we need some code to challenge a user to login and check whether they are

1) a valid user and

2) know the correct password.

```
#!/usr/bin/env perl
use v5.10.0;
 use warnings;
use strict;
use Digest::MD5 'md5_hex';
open(my $fh, '<', 'passwords') or die "cannot open passwords file $!";
 my %passwords = map({chomp; split(/:/, $_, 2)} <$fh>);
 my $user;
while (1) {
 print "Username: ";
chomp($user = <>);
print "Password: ";
chomp(my $pass = <>);
# they must be a valid user and
# their digested password must match the stored digest
last if( $passwords{$user} and md5_hex($pass) eq $passwords{$user} );
# otherwise, we're stuck in the loop
say "Sorry!";
}
say "Congratulations $user!";
```

First, your module needs somewhere to live. Create a directory named *lib/TestSite*.

```
$ mkdir -p lib/TestSite
package TestSite::Login;
$VERSION = v0.0.1;
use v5.10.0;
use warnings;
use strict;
use Carp;
use Digest::MD5 'md5_hex';
```

```
open(my $fh, '<', 'passwords') or die "cannot open
    passwords file $!";
my %passwords = map({chomp; split(/:/, $_, 2)} <$fh>);
sub check_password {
my ($user, $pass) = @_;
return(
 $passwords{$user} and
md5_hex($pass) eq $passwords{$user}
);
}
1;
```

The module consists of a few more lines of declaration.

First, we have the package statement, which declares the *namespace* of this module.

The package name should match the file path (with the '/' exchanged for '::'.)

Second, we have a $VERSION number, which every package should have (this helps when it comes time to deploy your code into the world.)

After the version, we have the same set of use lines as our scripts, but we also include the ***Carp*** module, which helps with debugging. Jumping to the very end, every module must end in a true value

(this satisfies a sanity check when it is loaded.)

## Aside from the declarations,

we've also migrated some code from the main program and changed the 'die' statement to croak() (this comes from the *Carp* module.)

The only other change is to put our password-checking code into the check_password() subroutine.

## *login.pl* code to use this new module. *login2.pl*.

- #!/usr/bin/perl
- use v5.10.0;
- use warnings;
- use strict;
- use TestSite::Login;
- my $user;
- while (1) {
- print "Username: ";
- chomp($user = <>);
- print "Password: ";
- chomp(my $pass = <>);
- last if(TestSite::Login::check_password($user, $pass));
- # otherwise, we're stuck in the loop
- say "Sorry!";
- }
- say "Congratulations $user!";

- Notice that we've removed the **use Digest::MD5 ...** line, because our new *TestSite::Login* module takes care of all of that.

When we run the new *login2.pl*,

we need to tell perl to also look for modules in our *lib* directory so it will find *lib/TestSite/Login.pm*.

You can install your module in one of the global @INC directories, but it is more convenient to use the -I switch when developing modules.

Run : perl  -I lib login2.pl

As you take on bigger programming challenges, modularizing behaviours helps to organize and simplify your code,

while also allowing you to reuse it as needed.

## Module for code reuse?

```
package My::Math;
use strict;
use warnings;
 use Exporter qw(import);
 our @EXPORT_OK = qw(add multiply);
 sub add {
  my ($x, $y) = @_;
  return $x + $y;
}
 sub multiply {
  my ($x, $y) = @_;
  return $x * $y;
}
1;
```

Save this in somedir/lib/My/Math.pm (or somedir\lib\My\Math.pm on Windows)

```
#!/usr/bin/perl
use strict;
use warnings;
use My::Math qw(add);
print add(19, 23);
```

Save this in somedir/bin/app.pl (or somedir\bin\app.pl on Windows).

It is going to print an error like this:

```
Can't locate My/Math.pm in @INC (@INC contains:
...
BEGIN failed--compilation aborted at somedir/bin/app.pl line 9.
```

## What is the problem?

In the script we loaded the module with the use keyword.

Specifically with the use My::Math qw(add); line.

This searches the directories listed in the built-in @INC variable looking for a subdirectory called **My** and in that subdirectory for a file called **Math.pm**.

The problem is that your .pm file is not in any of the standard directories of perl: it is not in any of the directories listed in @INC.

You could either move your module, or you could change @INC.

## Change @INC from the command line

Before we try to load the module, we have to make sure the directory of the module is in the @INC array.

Try this:

**perl -Isomedir/lib/ somedir/bin/app.pl**.

This will print the answer: 42.

In this case, the -I flag of perl helped us add a directory path to @INC.

## Change @INC from inside the script

- Because we know that the "My" directory that holds our module is in a fixed place **relative** to the script
- we have another possibility for changing the script:

```
#!/usr/bin/perl
use strict;
use warnings;

use File::Basename qw(dirname);
use Cwd  qw(abs_path);
use lib dirname(dirname abs_path $0) . '/lib';

use My::Math qw(add);

print add(19, 23);
```

- and run it again with this command:
- **perl somedir/bin/app.pl.**
- Now it works.

## How to changed @INC to point to a relative directory

This line: use lib dirname(dirname abs_path $0) . '/lib';

adds the relative lib directory to the beginning of @INC.

$0 holds the name of the current script. abs_path() of Cwd returns the absolute path to the script.

Given a path to a file or to a directory the call to dirname() of File::Basename returns the directory part, except of the last part.

In our case $0 contains app.pl

abs_path($0) returns .../somedir/bin/app.pl

dirname(abs_path $0) returns .../somedir/bin

dirname( dirname abs_path $0) returns .../somedir

That's the root directory of our project.

dirname( dirname abs_path $0) . '/lib' then points to .../somedir/lib

So what we have there is basically

use lib '.../somedir/lib';

but without hard-coding the actual location of the whole tree.

The whole task of this call is to add the '.../somedir/lib' to be the first element of @INC.

Once that's done, the subsequent call to use My::Math qw(add); will find the 'My' directory in '.../somedir/lib' and the Math.pm in '.../somedir/lib/My'.

The advantage of this solution is that the user of the script does not have to remember to put the -I... on the command line.

The first one it finds will be loaded into memory and the import function of My::Math will be called with the parameters after the name of the module.

In our case import( qw(add) ) which is just the same as calling import( 'add' ).

After the use statement is done calling the import function, we can just call the newly imported **add** function of the My::Math module.

## The explanation of the module

A module in Perl is a namespace in the file corresponding to that namespace. The package keyword creates the namespace. A module name My::Math maps to the file My/Math.pm. A module name A::B::C maps to the file A/B/C.pm somewhere in the directories listed in @INC.

## The explanation of the module

As you recall, the
use My::Math qw(add);
statement in the script will
load the module and then
call the import function.

That import function will look at the @EXPORT_OK array in your module and arrange for
on-demand importing of the functions listed in this array.

```
• package My::Math;
• use strict;
• use warnings;
•  use Exporter qw(import);
•  our @EXPORT_OK = qw(add
  multiply);
•  sub add {
•   my ($x, $y) = @_;
•   return $x + $y;
• }
•  sub multiply {
•   my ($x, $y) = @_;
•   return $x * $y;
• }
• 1;
```

The important thing to remember is that Exporter gives you the import.

The last thing I need to mention is the 1; at the end of the module. Basically the use statement is executing the module and it needs to see some kind of a true statement there.

## Object-Oriented Programming

## Libraries

If you have subroutines that you use repeatedly, you will probably be happier if they are kept in a single location and not re-typed or copied with every new use.  You want a library of useful subroutines.

A library consists of just a text file containing a group of subroutine definitions.  It doesn't need "#! /usr/bin/perl" at the top, and it doesn't need to be made executable with chmod.

However, libraries must always return a TRUE value, so the last line in a library is always the line "1;", since 1 is always true. This occurs after the last subroutine definition, outside all subroutines.

## Using Libraries

To use a library, put “require lib_name;” at the top of your program. Use the library file’s name. Once you do this, all subroutines in the library can be called directly from your program. You don’t need to put the actual subroutine definitions into your program.

## @INC

The easiest way to use a library is to have it in the same directory that your program is in.

Alternatively, you can put the full path to the library in the require statement:

```
require '/home/bios546/lib.pl';
```

If the library is not in the current directory, Perl looks in the directories listed in a built-in array called @INC. You can see the contents of @INC by writing a program containing a line like:

```
foreach (@INC) { print "$_ \n"; }
```

@INC contains directories like /usr/lib/perl5/5.6.1. These directories hold the basic Perl program and various added-in modules. You will not be able to write to these directories.

However, you can add your own directory to @INC. The simplest way to do this is to put a line at the top of your program such as:

```
unshift @INC, "path_to_your_lib";
```

This causes the first element of @INC to be your personal library directory, where you store all of your libraries.

# Adding @INC to Your Shell

You can cause Perl to always look for your personal library directory by editing the program that runs when you log in. Note: this isn't necessary; it's just interesting Unix-oriented information.

This program is .bash_profile (for mcahines such as biolinx that give you a bash shell when you log in). If you can't see .bash_profile when you look in your home directory, try "ls –a" or ls –al". By default, files starting with "." are invisible. The –a switch on ls allows them to be seen.

You can edit .bash_profile with pico or any other editor.

Put in 2 lines like:

```
PERL5LIB = /home/bios546
export PERL5LIB
```

You are creating a shell variable called PERL5LIB (note the 5 in it!), which is the path to your library directory. Then, the "export" command makes this visible in all subshells spawned by the original shell (a common event in Unix).

You will need to log out and then back in to get this to work, because .bash_profile onl runs when you log in.

You can see what you have set PERL5LIB to by typing "echo $PERL5LIB" at the command line prompt.

# Packages

Writing a library of useful subroutines and using them with a “require” statement is a good first step, but there’s more.

If you were to have 2 subroutines with the same name, say one in your main program and one in a library, you would have a “namespace collision”. That is, Perl would have to figure out which one you wanted to use, which might not be the same one you actually want.

To prevent namespace collisions, Perl uses “packages”.

A package is declared as in:

```
package MySubroutines;
```

If you want o have more than one package in the same text file, you need to enclose each separate package inside curly braces:

```
{ package MySubroutines;
    sub def1 {}
    sub def2 {}
    1;
}  # end of MySubroutines package
{ package ExtraStuff;
    sub def1  {  }
    sub def2 {  }
    1;
} # end of ExtraStuff package
```

# Using Subroutines in Packages

If you want to use a subroutine that is contained in a package, you need to erfer to it as:

```
PackageName::sub_name(subroutine_parameters);
```

This is called the “fully qualified name”.

Everything in your main program is by default in the package called “main”. Thus if you have subroutines called “mysub” in both your main program and in package “MySubrouines”, you could distinguish between them as:

```
main::mysub();
MySubroutines::mysub();
```

That is, give the package name followed by 2 colons, then the subroutine name.

You rarely need to actually use main::, because this is the default that is used if no specific package is given.

## Object-Oriented Programming

OOP is a programming style that is widely used, especially in large projects. OOP- style programs are easier to maintain than the “function-oriented” programs we have written so far. OOP programs are “encapsulated”: the programmer can alter specific parts without considering (or breaking) the larger program.

It is worth mentioning that OOP programs usually run more slowly than functional programs. Encapsulation has a significant cost. Nevertheless, nearly all high quality programs are OOP style these days.

## OOP Speak

A “class” is a generic object of a particular type: “gene” could be a class, or “student”.

An “instance” is a particular member of a class: “the beta-globin gene”, for example.

Objects contain “methods”, which are just subroutines.

- Some methods are class methods: meant to be invoked by the class. For instance, if you wished to create a new instance of the class “genes”, you would invoke the class method “new” (the constructor).
- Other methods work on instances. For example, if you wished to access data about a specific instance, such as the length or position of the gene, you would invoke an instance method.
- Often methods can be used by either classes or instances.

Objects also contain “data members”: pieces of information specific to that object. Most data members are specific to individual instances rather than to the whole class.

## More OOP Speak

Our goal is to describe how Perl implements various bits of OOP:

encapsulation

invocation of methods

inheritance

constructors

accessors (get and set methods)

destructors

# Classes

A class is defined by a package. All subroutines within that package are defined for that class and all of its instance.

Methods are invoked using the “arrow notation”, similar to that used with references. For example:

```
Class->method(arg1, arg2);
```

Perl translates this into:

```
Class::method(“Class”, arg1, arg2);
```

That is, when the arrow notation is used, Perl interprets the string before the arrow as a package name and the string after the arrow as a subroutine within that package. The subroutine is invoked in its fully qualified form. Also, the class name becomes the first argument passed to the subroutine, with all other arguments coming after it.

Understanding arrow notation is CRITICAL to dealing with object-oriented programming in Perl.

## Class Example

(from Schwartz and Phoenix, “Learning Perl Objects, References and Modules”)

```
{ package Horse;
    sub speak {print “A horse says neigh”; }
}
{ package Cow;
  sub speak { print “A cow says moo”; }
}
{ package Sheep;
    sub speak {print “A sheep says baaa”; }
}

my @pasture = qw(Horse Cow Sheep Sheep Cow);
foreach my $animal (@pasture) {
   $animl->speak;
}
```

## Using Arrow Notation

- Much of the code in the previous slide is unnecessarily duplicated.  We can make a few modifications to start simplifying it.

```
{ package Horse;
      sub sound { “neigh”; }
      sub speak {“
         my $class = shift;
         print “ A $class says $class->sound”; }
   }
  # repeat for Cow and Sheep

my @pasture = qw(Horse Cow Sheep Sheep Cow);
    foreach my $animal (@pasture) {
       $animl->speak;
    }
```

# Inheritance

All the animals are speaking in a similar way, only changing which animal it is and what it is saying. This is a perfect situation to use inheritance: make a super-class of Animal, with a method called “speak”, and have each individual class of animal use this method.

Inheritance in Perl is based on the array @ISA (“is-a”), which is used in each class to define superclasses. @ISA must be declared with the keyword “our”.

## @ISA Example

```
{ package Animal;
      sub speak {
          my $class = shift;
          print “A $class says $class->sound\n”;
      }
}
{package Horse;
   our @ISA = qw(Animal);
   sub sound { “neigh”; }
}
```

## More on @ISA

When a method is invoked, Perl first looks within the invoked package. If the method isn't found there, Perl then goes through the packages listed in @ISA, from left to right, depth first. This means that the first superclass and all of its superclasses are checked before the second superclass listed in @ISA is checked.

Note that if a method is defined in a class, and another method with the same name is defined in a superclass, the lowest level method will be used. This is how methods can be written to override other methods

# Starting at a Higher Level

Sometimes you want to use a method in a superclass without invoking the current package's method of the same name.

To accomplish this, use SUPER as a qualification to the method's name. This will cause the inheritance tree to be searched without searching the invoking class's package.

Say you want to add something to the Horse's speech. "speak" is a method of the superclass Animal.

```
 { package Horse;
     our @ISA = qw(Animal);
     sub sound { "neigh"; }
     sub speak {
         my $class = shift;
         $class->SUPER::speak;
         print " and he gallops off in a cloud of dust.\n};
    }
 }
```

When Horse->speak is invoked, Perl finds the method in Horse and runs it. The Horse speak method uses the "speak" method in Animal ($class->SUPER::speak), then prints an additional message of its own. Note that an infinite regression would result if the Horse "speak" method contained a line like $class->speak--it would invoke itself.

## Summary So Far

A class is defined by a package.  Methods are subroutines defined within the package.

Arrow notation to invoke methods: Class->method(arguments) is read by Perl as: Class::method("Class", arguments).

Inheritance: superclasses are listed in @ISA, which must be declared with "our" in the class package.

Starting the search for a method from a superclass: Class->SUPER::method(args).

## Instances

An instance of a class shares al of the class's methods, defined in the package.

Instances also have specific data members, not shared by other instances of the class.

Instance data is stored in an anonymous hash (using a hash reference) that is "blessed" into its class. (it is possible to use an array reference or even a scalar reference to store instance data, but this is rarely done.)

Using a hash to hold data members means that every instance will have the same hash keys, but different values.

## Constructor

A constructor is a class method that creates a new instance.

A constructor in Perl can have any name, but generally constructors are called "new".

Here's a simple constructor:

```
{  package Horse;
      sub new {
         my $class = shift;
         my $self = {};  # anonymous hash ref
         bless $self, $class;
      }
}

  my $mr_ed = Horse->new;  # invocation, creating a anew instance
```

# Cloning Constructors

Sometimes you want to clone an instance, create a new instance with the same properties as the old one. For instance:

```
my $mr_ed_son = $mr_ed->new;
```

This won’t work with the previous constructor, because in the line “bless $self, $class;”, the variable $class needs to be a string that names a class (package). $mr_ed is actually a reference to an anonymous hash that has been labeled an instance of the class Horse.

To get around this, the “ref” function is used. If the argument of ref is a reference, it returns the name of the class it is blessed into (or just “HASH”, “ARRAY”, etc. if it has not been blessed into a class). If the argument of ref isn’t a reference, ref returns FALSE.

```
{  package Horse;
     sub new {
       my $class = shift;
       $class = ref($class) || $class;
       my $slef = {};
       bless $self, $class;
     }
}
```

## Default Values

Instances hold individual data members that distinguish then from other instances. These data members are often assigned when the instance is created. Default values can be used if necessary.

The “new” method can take arguments, a list of data members and their values. These are passed as the array @_ to the “new” subroutine.

```
 sub new {
     my $class = shift;
     $class = ref($class) || $class;
     my $self = { “color” => “brown”,
                  “legs” => 4;
                  “name” => “An anonymous horse” };
     bless $self, $class;
}
```

my $mr_ed = Horse->new(“color” => “palamino”, “name” => “Mr. Ed”);

Recall that => is just another way of writing a comma. @_ starts out as: (Horse, color, palamino, name, Mr. Ed). After shifting off the class name, there are 4 items in @_, which are interpreted as key-value pairs by putting them insdie the {} of $self.

First default values are set for “color”, “legs”, and “na,e”. Then, color and name are re-set by the values form @_ .

## Accessors

Methods are needed to access and alter (get or set) data members.

Perl doesn’t easily enforce privacy: there are no “private” data members or methods. You are simply expected to use the listed accessor methods instead of directly accessing the instance’s anonymous hash.

Thus if you had an instance $mr_ed, you could set its color by: $mr_ed->{color} = “purple”; This is considered uncouth. Instead you should write and use a “set” method in the Horse package.

```
{ package Horse;
    sub set_color {
        my $self = shift;  # reference to instance anonymous hash
        $self->{color} = shift;
        return $self;   # or some other return value
    }
  sub get_color {
      my $self = shift;
      return $self->{color};
  }
```

The point of this is, you might change the set or get methods, or the hash keys used to store these values (maybe to “hue” instead of “color”). You can then change the set and get mthods without changing th euser interface. No matter what hash key you use in place of “color”, the set_color and get_color methods can be made to work properly.

## Combined Set or Get Method

```
sub color {
    my $self = shift;
    if (@_) {  # arguments exist, so it's a set
        $self->{color} = shift;
    }
   else {   # no arguments, so it's a get
       return $self->{color};
   }
}
```

To use as a “get” function:

```
print “Mr. Ed is colored “, $mr_ed->color, “\n”;
```

To use as a “set” function:

```
$mr_ed->color(“purple”);
```

# UNIVERSAL methods

All classes automatically inherit from the top level superclass UNIVERSAL.

You can write methods in this class if you like:

```
{ package UNIVERSAL;
    sub moan { print “ooooooooo”; }
}
```

Now every class inherits the “whistle method.  This is rarely needed.

UNIVERSAL has 2 built-in methods of great use: “isa” and “can”.

“isa” returns TRUE if the instance or class it is invoked on inherits from the class given in the argument:

```
    if ($mr_ed->isa(“Animal”) { print “He is an animal”; }
```

“can” returns TRUE if the instance has the named method somewhere in its inheritance tree:

```
    if ($mr_ed->can(“moan”) { print $mr_ed->moan; }
```

# AUTOLOAD

If a method is called, Perl first looks within the class's package, then searches the @ISA arrays, then looks in UNIVERSAL. If all else fails, Perl looks for a method called "AUTOLOAD" and runs it.

AUTOLOAD methods automatically contain the variable $AUTOLOAD, which contains the fully qualified name of the method that invoked it. You must declare $AUTOLOAD with "our".

Here's a simple use:

```
{ package Horse;
    sub AUTOLOAD;
        my $self = shift;
        our $AUTOLOAD;
        print "Sorry, the method $AUTOLOAD hasn't been defined yet\n";
    }
```

Invoking it:

```
$mr_ed->fly_to_the_Moon;  # undefined method
```

Results in:

```
"Sorry, the method Horse::fly_to_the_Moon hasn't been defined yet!"
```

## Destructors

Perl automatically destroys objects and variables whenever the last reference to them is removed.  Thus it is almost never necessary to actually write a destructor method.

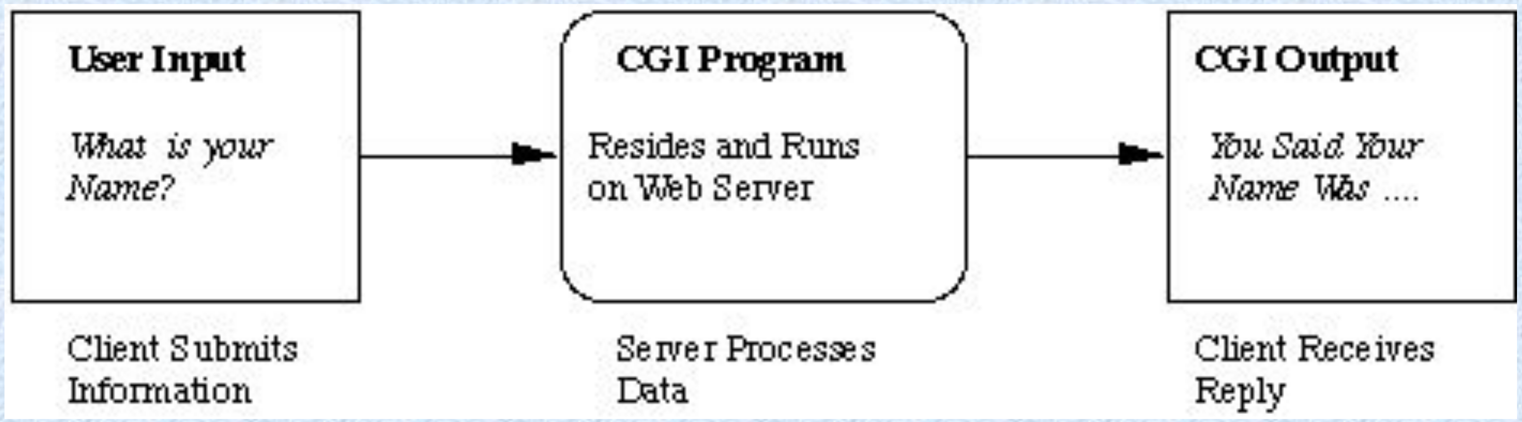
You can see the automatic destructor (called DESTROY) in action using an AUTOLOAD method from the previous slide.

You can even override DESTROY, or at least add to it if you like.

## Perl/CGI

# CGI Programming in Perl

- CGI: the standard programming interface between Web servers and external programs
  - Common: interacts with many different OSs
  - Gateway: provides users with a way to gain access to different programs
  - Interface: uses a well-defined method to interact with a Web server
- CGI standard lets Web browsers pass information to programs written in any language



## Writing and Running CGI Scripts

CGI scripts can be compiled programs or batch files or any executable entity

Typically CGI scripts are written in:

- Perl scripts
- C/C++ programs
- Unix Scripts

The 2 most common ways of running a CGI script are:

- From an HTML form – the "action" attribute of the form specifies the CGI script to be run
- Direct URL reference – A CGI script can be run directly by giving the URL explicitly in HTML

## Why Use Perl for CGI?

Perl is the standard for CGI programming:

- Socket support – create programs that interface seamlessly with Internet protocols
- Pattern matching – ideal for handling form data and searching text
- Flexible text handling – no details to worry
- Advantage of an interpreted language – simplicity in development, debugging, and revision

## How Does CGI Work?

CGI programs are always placed on a disk that the Web server has access to

Web servers are generally configured so that all CGI applications are placed into a cgi-bin directory

# Invoke CGI Programs

There are many ways to invoke CGI programs besides using a web browser to visit the URL

We can start CGI programs from:

- a hypertext link
  - <a href=“cgi-user/ftang/www/cgi-bin/hello.cgi”>Click here to run the hello CGI program</a>
- a button on a HTML form
- a server-side include

We can use a question mark to pass information to a CGI program.

- For example, passing keywords that will be used in a search
  - <a href=“cgi-user/ftang/www/cgi-bin/search.pl?Wine+1993”> Search for 1993 Wine</a>
  - The information follows the question mark will be available to your CGI program through the QUERY_STRING environment variables.

Generally speaking, visitors to your web site should never have to type in the URL for a CGI program.

A hypertext link should always be provided to start the program.

## The Hello World Example

When the web server executes your CGI program, it automatically opens the STDIN, STDOUT, STDERR file handles for you.

- STDIN: the standard input of your CGI program might contain information that was generated by an HTML form. Otherwise, you shouldn’t use STDIN.

- STDOUT: the standard output of your CGI program is linked to the STDIN of the web browser.

- STDERR: The standard output of your CGI program is linked to the web server’s log file.

## CGI Script Output

A CGI script must send information back in the following format:

- The output header
- A blank line
- The output data

For examples:

print(“Content Type: text/html\n\n”);

Note: Between the Header and Data there MUST be a blank line

## Output Header: Content-Type

The output header for a CGI script must specify an output type

3 forms of Header Type:

- Content-Type:
  - text/html, text/plain, image/gif, image/jpeg, application/, postscript, video/mpeg
- Location
- Status

Content-Type is the most popular type

See our hello world example:

- http://www.csupomona.edu/cgi-user/ftang/www/cgi-bin/hello.pl

## Output Header: Location

The **Location** header is used to redirect the client Web browser to another page

For example:

```
print "Location: http://www.google.com/\n\n";
```

## CGI Script Input

A CGI script will often require some form of input in order to operate

We’ll study:

- What form of input a CGI can receive
- How a CGI receives input
- How to process the input in a CGI Perl script
- How a useful Perl library makes this easy

## Accepting Input from the Browser

A CGI script can receive data in many ways:

- Environment variables
- Standard input
  - Data can be passed as standard input through the POST method of an HTML form
- Arguments of CGI script
  - If you call a CGI script directly or use the GET method of a form, arguments are following the “?” after the script URL and multiple arguments are separated by &

## Receiving/Processing Information in CGI

Two basic ways:

- Do it yourself: write Perl code to process the input
- Use pre-written Perl libraries
  - http://stein.cshl.org/WWW/CGI/ a Perl5 CGI Library

## Forms and CGI: What Can They Do?

A CGI program can be used to:

- accept the data which the user inputs
- do something with it
  - email the data to someone
  - add an entry to a database
  - write out a text file
  - create a customized display
  - anything that you can program
- send a reply back to user

## Form Processing

The two most important options with the <form> tag:

- **method**: specifies the manner in which form information is passed to the CGI scripts:
  - POST: contacts the CGI program and once connection established sends data
  - GET: Contacts the CGI program and sends data in a single transaction
- **action**: specifies the URL of the CGI script that will be invoked when the submit button is clicked
  - <from method = “post” action=“/cgi-bin/hello.pl”>

## Handling Form Information

- The GET method
  - <form method="get" action="/cgi-bin/guestbook.pl">
  - The GET method appends all of the form data to the end of the URL used to invoke the CGI script.
  - A question mark is used to separate the original URL and the form information.
    - http://somserver/cgi-bin/foo.pl?fname=Craig&lname=Kelley
  - The GET method can't be used for larger forms

## Handling Form Information

The POST method

```
<form method="post" action="cgi-bin/guestbook.pl">
```

The POST method sends all the form information to the CGI program using the STDIN file handle.

Which to use?

It is  good practice to use the GET method whenever you are able to because the POST method is more difficult for a user to manage, and it doesn’t function well with a browser’s back or history button.

# CGI.pm: a Perl 5 CGI Library

This Perl 5 library uses objects to do many things. It

- Uses objects to create Web fill-out forms on the fly and to parse their contents
- Provides a simple interface for parsing and interpreting query strings passed to CGI scripts
- Offers a rich set of functions for creating fill-out forms

Everything is done through a "CGI" object

The most basic use of CGI.pm is to get the query parameters submitted to your script. To do so, put the following at the top of your Perl/CGI programs:

```
Use CGI;
$query = new CGI;

Or Use CGI qw(:standard);
```

http://stein.cshl.org/WWW/CGI/

## Function Oriented vs. Object Oriented

```
#!/usr/local/bin/perl
# imports standard
   set of functions
use CGI
   qw/:standard/;


print header(),
        start_html(-
   title=>'Wow!'),
```

```
#!/usr/local/bin/perl


use CGI;
$q = new CGI;
print $q->header(),
        $q->start_html(-
   title=>'Wow!'),
        $q->h1('Wow!'),
        'Look Ma, no
```

http://stein.cshl.org/WWW/CGI/#functionvsoo

## What Can You Do with the Query Object?

Fetch the names of all the parameters passed to your script

```
@names = $query->param;
```

Fetch the value(s) of a named parameter

```
@values = $query->param('foo'); # or
$value = $query->param('foo');
```

And many other methods …

Example:

http://www.csupomona.edu/~ftang/www/courses/CS299-S09/examples/simple.html

## Printing HTML

- Instead of printing the quote marks and write in new line characters
- An easier way is to use a special print command in Perl:

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print <<ENDHTML;
<html>
<head>
<title>CGI Test</title>
</head>
<body>
<a href="http://what.com">Click Here</a>
</body>
</html>

ENDHTML
;
```

## Save the Current State of a Form

Save the state to a file

```
$query->save(FILEHANDLE);
```

The contents are written out as TAG=VALUE pairs

Save the state in a self-referencing URL

```
$my_url = $query->self_url;
```

Useful when you want to jump around within a script-generated document using internal anchors, but don't want to disrupt the current contents.

## Three Different Methods

Print out the information after form submission

www.csupomona.edu/~ftang/www/cgi-bin/procform1.cgi

Submit the information to an email address

www.csupomona.edu/~ftang/www/cgi-bin/procform2.cgi

Save the information to a file (will not use this one)

www.csupomona.edu/~ftang/www/cgi-bin/procform3.cgi

To run the above programs:

http://www.csupomona.edu/~ftang/www/courses/CS299-S09/examples/runcgi.html

## Creating the HTTP Header

Creating the standard header

```
print $query->header()
```

default to ‘text/html’

Creating the header for a redirection request

```
print $query->redirect(‘http://www.google.com’);
```

## HTML Shortcuts

Create an HTML header

```
print $query->start_html(many options);
```

End an HTML document

```
print $query->end_html;
```

Shortcut methods for many HTML tags

Importing CGI methods

- Single query object
- import CGI module methods

```
use CGI qw(:standard);
$dinner = param('entrée');
```

## Other Supports

Support for CSS

http://www.csupomona.edu/cgi-user/ftang/www/cgi-bin/style.pl

http://www.csupomona.edu/cgi-user/ftang/www/cgi-bin/bio.pl

Support for JavaScript

Similar as CSS

check http://stein.cshl.org/WWW/CGI/ for details

http://www.csupomona.edu/cgi-user/ftang/www/cgi-bin/contact.pl

Thank You