**Perl Part 1**

|  |  |
| --- | --- |
| **Perl Overview**   * Perl is an **interpreted** language. * **Comments** use **#.** * Often, **.pl** is used as the file extension for Perl files. * The perl shebang (first line and must begin in the first column) can be used to tell sh to invoke Perl. You can also specify **-w** (to display warning messages).   #!/usr/bin/perl -w   * Available on Unix implementations and Microsoft Windows   **Data**   * **No** explicit **declaration** of variables. Variables receive their data type, structure, size, and value from assignment. * Scalars are referenced by $*variable*. * Arrays begin with an @. (e.g., @names) * Array elements that are scalars are referenced by $*array*[*subscript*] (e.g., $names[1]) * Associative arrays are called **hashes** * **Slices** of arrays are provided * Command line arguments are placed in the @ARGV array   **Sequence Control**   * Flow control statements (**if**, **for**, **while**) must be followed by curly braces even when there is only one statement. * A **case** statement is provided.   **Invocation**  To invoke a script named large.pl:   * $ perl large.pl *arguments* * If it has executable permission and contains the Perl shebang: $ large.pl *arguments*   **Reminder:** see my setup section for an approach to more easily insert  #!/usr/bin/perl -w  Create an PerlExamples directory. When logged into a fox server, please cd to the /usr/local/courses/clark/cs3423/PerlExamples directory and copy all the files to your PerlExamples directory. | **Example 1**: To better help you learn perl, a complete example is provided. Contents of large.pl is shown here. It shows which '.pl' files within a specified directory are larger than a specified size.  $ vi large.pl  #!/usr/bin/perl -w  use strict;  my $count = 0;  # See if there are two arguments to the command. Note that  # $#ARGV is the subscript of the last argument. Since subscripts  # begin with 0, it should be one less than the number expected.  if ($#ARGV != 1) {  print "Usage: large directoryName biggerThanSize\n";  die "two arguments expected";  }  # make certain the directory exists  if (not -d $ARGV[0]) {  print "Usage: large directoryName biggerThanSize\n";  die "specified directory $ARGV[0] does not exist";  }  # get all the \*.pl files in the specified directory  my @files = glob("$ARGV[0]/\*.pl");  foreach my $file (@files) {  # see if file size is greater than the specified size  if (-s $file > $ARGV[1]) {  print "$file \t ", -s $file, "\n";  $count++;  }  }  print "\nNumber of files in '$ARGV[0]' larger than $ARGV[1] is $count\n";  $ perl large.pl . 100  ./foreach.pl 391  ./large.pl 868  ./quotient.pl 158  ./scalars.pl 276  ./sumN.pl 436  Number of files n '.' larger than 100 is 5 |
| **Warning Warning Warning**  Some people like/dislike that perl (by default) tries to understand what you specified and doesn't give many errors. For example, if you had coded lowercase $argv[0] in large.pl without specifying **use strict** (or **-w** on the shebang line), the interpreter would not give you a syntax error. Variable names are case sensitive so $argv[0] is not the same as $ARGV[0]. During execution, it would assume the value for $argv[0] is undefined and assume a null value.  You might like the output provided by **use strict** better than the  #!/usr/bin/perl -w  Using use strict:  #!/usr/bin/per  use strict;  # We can be more specific  use strict 'vars'; # We want to require variables to be declared  ...  no strict 'vars'; # We'll go back to normal variable rules now  ...  no strict; # Turn it off. Turn it all off. Go away, strict | |
| **Scalar Variables**  Scalar variables are declared and accessed using a "$". The datatype of the value is dependent on how it is used. When performing an arithmetic operation, it assumes the value of the scalar variable is numeric. Otherwise, it assumes it is a string.  An assignment of a scalar to a scalar will **copy** the value. | **Example 2**: Scalars are numbers and strings  $ vi scalars.pl  #!/usr/bin/perl -w  # scalars  my $total = 10.95;  my $courseNumber = "CS3423";  my $weird = "3423";  $total += $weird;  print "My total is $total\n";  my $course = $courseNumber;  $course = "CS" . 1713;  print "course is ", $course, "\n";  print "courseNumber is ", $courseNumber, "\n";  $ perl scalars.pl  My total is 3433.95  course is CS1713  courseNumber is CS3423 |
| **Constants: Numeric, Boolean, and String**  Numeric constants: 100, 3.14, 45.7e22  Strings:  'literal' unevaluated; even 'hello\n' is a literal that does not contain a line feed. It literally contains the characters '\' and 'n'.  "string" evaluated; "hello $name\n" is 'hello ' followed by the value of the scalar variable $name and a linefeed.  `*command`* Executes the specified Unix (or windows) command. It replaces scalar variables with their values.  **Boolean** **False** Values: undef, 0, '', '0'  **Boolean** **True** Values: 1, 5, '00', 'true', 'false' | **Example 3**: constants, literals, strings, boolean  $ vi const  #!/usr/bin/perl -w  my $pi = 3.14;  my $radius = 3;  my $circumference = 2 \* $pi \* $radius;  my $name = "Ab Normal";  my $sentence = "Hello $name, how are you?";  print '$circumference', " is $circumference\n";  print $sentence, "\n";  if ($radius){  print 'radius is true';  }  # print `dir /b`, "\n"; # this is for windows b=bare  print `ls`, "\n"; # this is for Unix  $ chmod +x const  $ const  $circumference is 18.84  Hello Ab Normal, how are you?  radius is true  const  large.pl  scalar.pl |
| **Important Scalar Operators**  **Arithmetic**  *arg1* + *arg2* addition  *arg1* - *arg2* subtraction  *arg1* \* *arg2* multiplication  *arg1* / *arg2* division  *arg1* % *arg2* modulus  *$var = expr* assignment (data type of *expr* determines the data type of the resulting variable).  *$var += expr* add *expr* to the specified variable  *$var -= expr* subtract *expr* to the specified variable  $var ++ increments  $var -- decrements  rand(*arg*) generates a random number between 0 and *arg.*  **String**  *arg1 . arg2* concatenation  *arg1* x *arg2* returns a string resulting from repeating *arg1 arg2* times.  "duh" x 3 would be "duhduhduh"  substr(*str*, *beg, lgth)* returns a substring of *str* beginning at *beg* (relative to 0) for a length of *lgth.* The length is optional; it assumes to the end of the string if not specified. | **Example 4**: scalar operators  $ vi scalarop  #!/usr/bin/perl  use strict;  print "random is ", rand(5), "\n";  my $rand = rand(5);  print '$rand ', "is $rand\n";  my $abcId = "abc" . 123;  print "abcId is $abcId\n";  my $abc = substr($abcId, 0, 3);  my $num = substr($abcId, 3);  print "parts are '$abc' and '$num'\n";  my $pirateWords = "a" . "r" x $rand;  print "pirate said ", $pirateWords, "\n";  $ perl scalarop  random is 2.67953110895634  $rand is 4.94665668041975  abcId is abc123  parts are 'abc' and '123'  pirate said arrrr |
| **Arrays**  Arrays are declared with an "@". If you access the array (not just an element), it is accessed with an "@". If you access an element of the array, use a scalar reference ("$") and square brackets around the subscript.  Arrays can be initialized:  @*array* = (*valueList*); # e.g., ('cat', 'dog', 'bird')  # e.g., ($one, 'is', "$two is the best")  @*array* = (*range*); # e.g., (50..100)    The subscript of the last element of an array is accessed by $#*arrayName.* If there are three elements in an array, the highest subscript would be 2.  Array assignments  Element assignments  $*array*[*subscript*] = *expr*;  Assigning an array into an array will **copy** it.  my @*targetArray* = *@sourceArray*;  A slice of an array can be accessed by specifying:  *@arrayName*[*subscriptSlice*]  where *subscriptSlice* can be a comma-separated list of subscripts and/or a range of subscripts separated by '..'. | **Example 5**: arrays  $ vi arrays  #!/usr/bin/perl -w  #arrays  my @fruit = ("apple", "grape", "orange");  my @veg = ("broccoli", "green beans", "asparagus", "corn");  print "fruit: @fruit\n";  print "veg: @veg\n";  print "fruit[1] is $fruit[1] \n";  print "last subscript for fruit is ", $#fruit, "\n";  print "last subscript for veg is ", $#veg, "\n";  print "the last element of veg: $veg[$#veg] \n";  print "A slice of fruit:", @fruit[0,2], "\n";  print "A slice of veg:", @veg[1..3], "\n";  print "Another slice of veg:", @veg[0,2..3], "\n";  my @newVeg = @veg[0,2..3];  print "newVeg is @newVeg\n";  my @newFruit = @fruit;  # replace a fruit  $fruit[1] = 'maynard';  # add a fruit to the end of the fruit array  $fruit[$#fruit+1] = "clark";  print "fruit: @fruit\n";  print "newFruit: @newFruit\n";  $ perl arrays  fruit: apple grape orange  veg: broccoli green beans asparagus corn  fruit[1] is grape  last subscript for fruit is 2  last subscript for veg is 3  the last element of veg: corn  A slice of fruit:appleorange  A slice of veg:green beansasparaguscorn  Another slice of veg:broccoliasparaguscorn  newVeg is broccoli asparagus corn  fruit: apple maynard orange clark  newFruit: apple grape orange |
| **Multi-target Assignments**  (*variableList*) = @array;  assigns the first value of the array to the first variable, second value to the second variable, and so on.  (*variableList*) = (*range*);  assigns the first value of the range to the first variable, second value to the second variable, and so on.  (*variableList*) = split( /*pattern*/, *string, limit*);  assigns parts of splitting *string* based on the *pattern* to the specified variables. *limit* is optional. If *limit* is a positive number, it represents how many times the value is split. If 3 is specified, the first two variables receive the first two parts and the remainder is placed in the third variable.  For single character delimiters, we can use vertical bars to separate the alternatives in the pattern. We could also list the alternatives within square brackets. | **Example 6:** multi-target assignments  **$ vi multi**  #!/usr/bin/perl  my @pets = ('cat', 'dog', 'bird', 'bat');  print "@pets\n";  ($x, $y, $z) = @pets;  print "x=$x, y=$y, z=$z\n";  ($a, $b, $c) = (10..12);  print "a=$a, b=$b, c=$c\n";  $line = "clark:x:1000:1000:clark:/home/clark:/bin/tcsh";  ($login, $pwd, $uid, $gid, $name, $home) = split /:/, $line;  print "login=$login, uid=$uid, gid=$gid, home=$home\n";  ($one, $two, $three) = split /:/, $line, 3;  print "one=$one, two=$two, three=$three\n";  my $sentence = "If I wanted bling, I would get gold not perl.";  # Use space, comma, and period as delimiters  my @words = split (/[ ,.]/, $sentence);  print "@words\n";  $ perl multi  cat dog bird bat  x=cat, y=dog, z=bird  a=10, b=11, c=12  login=clark, uid=1000, gid=1000, home=/home/clark  one=clark, two=x, three=1000:1000:clark:/home/clark:/bin/tcsh  If I wanted bling I would get gold not perl |
| **Flow Control Statements**  The **while, if,** and **for** statements are used for flow control. perl also provides **until** and **unless** statements.  **while**, **if**, and **elsif** execute the corresponding *doSomething* statements if the *condExpr* is true (non-zero). Unlike C, you must put the curly braces around *doSomething* even when there is only one statement. See also embedded conditions.  **while loop**  while (*condExpr*) {  *doSomething*  }  **simple if**  if (*condExpr*) {  *doSomething*  }  **if-then-else**  if (*condExpr*) {  *doSomething*  }  else {  *doSomething*  }  **if-then-elsif-...else**  if (*condExpr*) {  *doSomething*  }  elsif (*condExpr*) {  *doSomething*  }  …  else {  *doSomething*  } | **Example 7**: sum the integers from 0 to N  $ vi sumN.pl  #!/usr/bin/perl -w  # Check for wrong number of arguments.  # There should be 1 argument which means the last element is $ARGV[0]  # Note that $#ARGV will be -1 when there are no arguments  if ($#ARGV != 0) {  print "Usage: sumN n\n";  die "must specify a positive integer n\n";  }  my $sum = 0;  my $i = $ARGV[0];  # embedded condition  die "argument must be positive\n" if ($i < 0);  # Calculate the sum using while  while ($i > 0) {  $sum += $i  $i -= 1; }  print $sum, "\n";  $ chmod +x sumN.pl  $ sumN.pl 3  6 |
| **Comparison Operators**  Since Perl doesn't have a datatype for scalar variables, we must have a set of numeric comparison operators and string comparison operators.  Numerically, 3 is less than 10; however, 3 is greater than 10 when using a string comparison.   |  |  |  | | --- | --- | --- | | Numeric Operator | String Operator | Returns | | == | eq | True if equal | | != | ne | True if not equal | | < | lt | True if less than | | > | gt | True if greater than | | <= | le | True if less than or equal to | | >= | ge | True if greater than or equal to | | <=> | cmp | -1 if less than,  0 if equal,  1 if greater than | | **Example 8: if-elsif within input while loop**  $ vi univ  #!/usr/bin/perl -w  my $university;  print "Enter the name of your university:";  while ($university = <STDIN>) {  chomp $university; # remove the linefeed  $university = lc $university; # lowercase  if ($university eq "utsa") {  print "blue and orange\n";  }  elsif ($university eq "ut") {  print "burnt orange\n";  }  elsif ($university eq "tamu") {  print "moron\n";  }  else {  print "unknown university\n";  }  print "Enter the name of your university:\n";  } |
| **for and foreach statement**  The **for** statement is similar to the C **for** statement:  for (*initialize*; *condExpr*; *increment*) {  *doSomething*  }  The **foreach** iterates over a list  foreach *var* (*@array*) {  *doSomething*  }  foreach *var* (*valueList*) {  *doSomething*  }  You can leave off the *var* and it will assign each value into $\_  foreach (*@array*) {  *doSomething*  }  foreach (*valueList*) {  *doSomething*  } | **Example 9:** code sumN.pl using **for**  # We can use the **for** in our sumN example  my $sum = 0;  my $i;  for ($i = $ARGV[0]; $i > 0; $i--) {  $sum += $i;  }  print $sum;  $ perl sumN.pl 5  15  **Example 10**: foreach in an array  $ vi foreach.pl  print '1. List of animals from @animals', "\n";  my @animals = ("monkey", "baboon", "longhorn", "aggie");  foreach my $animal (@animals) {  print $animal, "\n";  }  print '2. List of fruit from a valueList', "\n";  foreach my $fruit ("apple", "grape", "apple") {  print $fruit, "\n";;  }  print '3. List of animals from @animals using $\_', "\n";  foreach (@animals) {  print $\_, "\n";  }  $ perl foreach.pl  1. List of animals from @animals  monkey  baboon  longhorn  aggie  2. List of fruit from a valueList  apple  grape  apple  3. List of animals from @animals using $\_  monkey  baboon  longhorn  aggie |
| **Embedded Conditions**  To reduce coding, embedded conditions are used in Perl.  **Embedded if** executes *statement* only if *condition* is true.  *statement* if (*condition*);  **Embedded unless** executes *statement* only if *condition* is false  *statement* unless (*condition*);  **Embedded or** executes *statement1,* but only executes *statement2* if *statement1* was successful.  *statement1* or *statement2;* | **Example 11:** Embedded conditionals  $ vi quotient.pl  #!/usr/bin/perl -w  # Check for wrong number of arguments.  die "missing argument\n" if ($#ARGV < 1);  # Don't provide quotient if 2nd argument is zero  print $ARGV[0] / $ARGV[1], "\n" unless ($ARGV[1] == 0); |
| **next and last**  The **next** and **last** statements can be used within **for, while,** and **until** statements.  **last** (aka, break in most languages) exits the loop.  **next** (aka, continue in most languages) continues with the **next** iteration skipping the remaining statements within the current iteration. | **Example 12**: using next to skip an invalid file  $ vi showall  #!/usr/bin/perl -w  # for each valid file in the command arguments, output the name  # of the file and its contents  foreach my $file (@ARGV) {  next unless ( -r $file ); #skip an invalid file  print ">>> $file \n";  $output = `cat $file`; #save the stdout of cat in a  #string variable  print $output;  }  print "\n";  $ perl showall junk1 junk2 junk3 junk4  >>> junk1  Inside junk1 ... line1  Inside junk1 ... line2  >>> junk2  Inside junk2 ... line1  Inside junk2 ... line2  Inside junk2 ... line3  >>> junk4  Inside junk4 ... line1  Inside junk4 ... line2  Inside junk4 ... line3 |
| **Exercise**: show code for the Perl program named "examine" which will examine $ARGV[1] and show:   * Whether it is ODD or EVEN * If it is a perfect square of a number between 2 and $ARGV[0], show a message stating what it is a square of; otherwise, show a message that it isn't a perfect square.   Notes:   * Verify that we are passed two arguments. * make "examine" an executable.   $ examine 5 9  ODD  Square of 3  $ examine 5 16  EVEN  Square of 4  $ examine 5 12  EVEN  Not a perfect square | #!/usr/bin/perl -w  die "too few arguments" if ??  # Odd or even?  ??  my $bPerfect = 0;  my $i;  # is it a square of a number <= $ARGV[0] ?  for ??  ??  if ($bPerfect) {  ??  } else {  ??  } |
| **Reading input**  To read from a file:  <*fileHandle*> reads a **line** of input terminated by a linefeed. The value returned will contain the linefeed.  <STDIN> reads a line from STDIN.  <> reads from the files specified as input arguments or STDIN.  Notes:   * When an empty text line is read, the <*fileHandle>* will return "\n". * When end of file is reached, <*fileHandle>* returns a null string which is also treated as false. | **Example 13:** simple input and parsing input  $ vi simpInput  #!/usr/bin/perl -w  print "what is your first name? ";  defined(my $firstName = <STDIN>)  or die "input error from STDIN (likely EOF)";  chomp $firstName; # remove newline  print $firstName, "\n";  print "what is your last name? ";  defined(my $lastName = <STDIN>)  or die "input error from STDIN (likely EOF)";  chomp $lastName;  print $lastName, "\n";  print "what is your major? ";  defined(my $major = <STDIN> )  or die "input error from STDIN (likely EOF)";  chomp $major;  print $major, "\n";  # We can use pattern matching to parse out values.  # More about that later.  print "What is your first name, last name, and major (separate with",  " commas)";  defined(my $line = <STDIN> )  or die "input error from STDIN (likely EOF)";  $\_ = $line;  my ($first, $last, $maj) = /([^,]\*),([^,]\*),([a-zA-Z]\*)/;  print "first name is ", $first, "\n";  print "last name is ", $last, "\n";  print "major is ", $maj, "\n";  $ perl simpInput  what is your first name? bob  bob  what is your last name? wire  wire  what is your major? cs  cs  What is your first name, last name, and major (separate with commas)barb,wire,cs  first name is barb  last name is wire  major is cs |
| **Opening and Closing Files**  The **open** function is used to open files. Syntax:  **open** (*fileHandle*, *mode*, *fileName)*;  Some modes:  < read, error if it doesn't exist  > creation/write; overwrites  >> append/write  For more information, see perldoc -f open.  **Reading until eof**:  open (*fileHandle*, "<", *fileName*);  while (my $*variableName* = <*fileHandle*>) {  *do something with it*  }  **Closing a file:**  close(*fileHandle*);  Notes:   * Older Perl code may use open(MYFILE, "< one.txt"); | **Example 14:** EOF input loop from "one.txt" and write last line to "last.txt".  $ vi lastLine  #!/usr/bin/perl -w  # Writes the last line of one.txt to last.txt.  open(INFILE, "<", "one.txt")  or die "could not open INFILE: $!\n";  my $lastLine;  while(my $line = <INFILE>) {  $lastLine = $line;  }  close(INFILE);  open(OUTFILE, ">", "last.txt");  print OUTFILE $lastLine;  close(OUTFILE);  $ cat one.txt  one  two  three and last line  $ perl lastLine  $ cat last.txt  three and last line |
| **File Tests**  perl provides several operations to test properties about files.  -r *filename* returns true if the specified file is **readable**  -w *filename* returns true if the specified file is **writable**  -e *filename* returns true if the specified file **exists**  -s *filename* returns the **size** of the specified file  -d *filename* returns true if the specified file is a **directory**  -f *filename* returns true if the specified file is a **file**  -T *filename* returns true if the specified file is a **text** file | **Example 15:** find files in a directory that are larger than the second arg.  $ vi example15  # get all the \*.pl files in the specified directory  my @files = glob("$ARGV[0]/\*.pl");  foreach my $file (@files) {  # see if file size is greater than the specified size  if (-s $file > $ARGV[1]) {  print "$file \t ", -s $file, "\n";  $count++;  }  }  print "\nNumber of files larger than $ARGV[1] is $count\n"; |
| **Exit Status**  Like shell, perl programs should provide an exit status when terminating abnormally. **die** exits with a non-zero status. **exit status values**:  0 success  non-zero command failed  This can be tested to see whether something that was invoked actually worked.  Your Perl programs can return a failure by executing:  exit *n;*  To show a failure, the value of *n* will typically be 1 for most programs.  Note that **die** exits with the value of the last command's exit code. This is the value of $? | exit 1; |

**©2017 Larry W. Clark,** UTSA CS students may make copies for their personal use