|  |
| --- |
| Explain Perl. When do you use Perl for programming? What are the advantages of programming in Perl?  **About PERL:**  PERL is Practical Extraction and Reporting language, which is a high level programming language written by Larry Wall. The more recent expansion is Pathologically Eclectic Rubbish Lister .   PERL is a free open source language.  It is simple to learn as its syntax is similar to C  It supports OOP – Object oriented programming like C++  Unlike C/ C++ it is a lot more flexible in usage  When do we use PERL for Programming:  Generally PERL is used to develop web based applications even though libraries are available to program web server applications, database interfaces and networking components. Example: The popular e-commerce site www.amazon.com was developed with PERL.  **Advantages of programming in Perl** As mentioned above, PERL  -is easier to understand due to its simple syntax -is easier to use due to its flexibility -supports OOP  -is easily readable |
| How would you ensure the re-use and maximum readability of your Perl code?  modularize code and include them where required using the “use” command  use subroutines or functions to segregate operations thereby making the code more readable  use objects to create programs wherever possible which greatly promotes code reuse  include appropriate comments as and when required  eliminate any dereferencing operator |
| What is the importance of Perl warnings? How do you turn them on?  Warnings are one of the most basic ways in which you can get Perl to check the quality of the code that you have produced. Mandatory warnings highlight problems in the lexical analysis stage. Optional warnings highlight cases of possible anomaly.  The traditional way of enabling warnings was to use the -w argument on the command line:  perl -w myscript.pl You can also supply the option within the "shebang" line: #/usr/local/bin/perl -w  You can also mention use warnings with all, deprecated and unsafe options. **Eg:** use warnings 'all'; |
| Differentiate between Use and Require, My and Local, For and Foreach and Exec and System  **Use and Require**  Both the Use and Require statements are used while importing modules.  A require statement imports functions only within their packages. The use statement imports functions with a global scope so that their functions and objects can be accessed directly.  **Eg.** Require module;  Var = module::method(); //method called with the module reference  **Eg:** use module;  Var = method(); //method can be called directly  -Use statements are interpreted and are executed during the parsing whereas the require statements are executed during run time thereby supporting dynamic selection of modules.  **My and Local**  A variable declared with the My statement is scoped within the current block. The variable and its value goes out of scope outside the block whereas a local statement is used to temporarily assign a value to the global variable inside the block. The variable used with local statement still has global accessibility but the value lasts only as long as the control is inside the block.  **For and Foreach**  The for statement has an initialization, condition check and increment expressions in its body and is used for general iterations performing operations involving a loop. The foreach statement is particularly used to iterate through arrays and runs for the length of the array.  **Exec and System**  Exec command is used to execute a system command directly which does not return to the calling script unless if the command specified does not exist and System command is used to run a subcommand as part of a Perl script.  i.e The exec command stops the execution of the current process and starts the execution of the new process and does not return back to the stopped process. But the system command, holds the execution of the current process, forks a new process and continues with the execution of the command specified and returns back to the process on hold to continue execution. |
| Situation : You are required to replace a char in a string and store the number of replacements. How would you do that?  #!usr/bin/perl use strict; use warnings; my $mainstring="APerlAReplAFunction"; my $count = ($mainstring =~ tr/A//); print "There are $count As in the given string\n"; print $mainstring; |
| Situation: You want to concatenate strings with Perl. How would you do that?  By using the dot operator which concatenates strings in Perl. **Eg.**  $string = “My name is”.$name |
| Situation - There are some duplicate entries in an array and you want to remove them. How would you do that?  If duplicates need to be removed, the best way is to use a hash. **Eg:**  sub uniqueentr  {    return keys %{{ map { $\_ => 1 } @\_ }}; } @array1 = ("tea","coffee","tea","cola”,"coffee"); print join(" ", @array1), "\n"; print join(" ", uniqueentr(@array1)), "\n"; |
| What is the use of command "use strict"?  Use strict command calls the strict pragma and is used to force checks on definition and usage of variables, references and other barewords used in the script. If unsafe or ambiguous statements are used, this command stops the execution of the script instead of just providing warnings. |
| Explain the arguments for Perl Interpreter.  -a - automatically splits a group of input files -c - checks the syntax of the script without executing it -d - invokes the PERL debugger after the script is compiled -d:module - script is compiled and control is transferred to the module specified. -d - The command line is interpreted as single line script -S - uses the $PATH env variable to locate the script -T - switches on Taint mode -v - prints the version and path level of the interpreter -w - prints warnings |
| What would happen if you prefixed some variables with following symbols?  i.) $ - The variable becomes a scalar variable which can hold one value only ii.) @ - The variable becomes an array variable which can hold a list of scalar variables iii.) % - The variable becomes a hash variable which stores values as key-value pairs |
| What is the use of following?  **i.) –w** When used gives out warnings about the possible interpretation errors in the script.  **ii.) Strict** Strict is a pragma which is used to force checks on the definition and usage of variables, references and other barewords used in the script. This can be invoked using the use strict command. If there are any unsafe or ambiguous commands in the script, this pragma stops the execution of the script instead of just giving warnings.  **iii.) -T.** When used, switches on taint checking which forces Perl to check the origin of variables where outside variables cannot be used in system calls and subshell executions. |
| Explain: a.) Subroutine b.) Perl one-liner c.) Lists d.) iValue  **a.) Subroutine** Subroutines are named blocks of code that accept arguments, perform required operation and return values. In PERL, the terms subroutine and function are used interchangeably. Syntax for defining subroutine: sub NAME or sub NAME PROTOTYPE ATTRIBUTES to be specific where prototype and attributes are optional. PROTOTYPE is the prototype of the arguments that the subroutine takes in and ATTRIBUTES are the attributes that the subroutine exhibits.  **b.) Perl one-liner** One-liners are one command line only programs (may contain more than one perl statements) that are used to accomplish an operation. They are called so because the program can be typed and executed from the command line immediately. **Eg:**  # run program, but with warnings  perl -w my\_file # run program under debugger  perl -d my\_file  **c.) Lists** Lists are special type of arrays that hold a series of values. Lists can either be explicitly generated by the user using a paranthesis and comma to separate the values or can be a value returned by a function when evaluated in list context.  **d.) iValue** An ivalue is a scalar value that can be used to store the result of any expression. Ivalues appear in the left hand side of the expression and usually represents a data space in memory.  Situation: You want to concatenate strings with Perl. How would you do that?  By using the dot operator which concatenates strings in Perl. **Eg.**  $string = “My name is”.$name  Situation: You want to download the contents of a URL with Perl. How would you do that?  - Use use the libwww-perl library, LWP.pm - #!/usr/bin/perl use LWP::Simple; $url = get 'http://www.DevDaily.com/'; |
| Situation: You want to connect to SQL Server through Perl. How would you do that?  Perl supports access to all of the major database systems through a number of extensions provided through the DBI toolkit, a third-party module available from CPAN. Under Windows you can use either the DBI interfaces or the Win32::ODBC toolkit, which provides direct access to any ODBC-compliant database including SQL Server database products. Using DBI:  use DBI; my $dbh = DBI->connect(DSN); Using ODBC: Use Win32::ODBC; $database = new Win32::ODBC("DSN" [, CONNECT\_OPTION, ...]); |
| Situation - You want to open and read data files with Perl. How would you do that?  open FILEHANDLE - used to open data files and filehandle points to the file that is opened read FILEHANDLE, SCALAR, LENGTH - used to read from filehandle of length LENGTH and the result is placed in SCALAR. close FILEHANDLE - closes file after reading is complete. |
| Explain the different types of data Perl can handle.  - Scalars : store single values and are preceded by $ sign - Arrays: store a list of scalar values and are preceded by @ sign - Hashes: store associative arrays which use a key value as index instead of numerical indexes. Use % as prefix. |
| Explain different types of Perl Operators.  -Arithmetic operators, +, - ,\* etc -Assignment operators: += , -+, \*= etc  -Increment/ decrement operators: ++, -- -String concatenation: ‘.’ operator  -comparison operators: ==, !=, >, < , >= etc -Logical operators: &&, ||, ! |
| Situation: You want to add two arrays together. How would you do that?  @sumarray = (@arr1,@arr2); We can also use the push function to accomplish the same.  Situation: You want to empty an array. How would you do that?  -by setting its length to any –ve number, generally -1 -by assigning null list |
| Situation: You want to read command-line arguements with Perl. How would you do that?  In Perl, command line arguments are stored in an array @ARGV. Hence $ARGV[0] gives the first argument $ARGV[1] gives the second argument and so on. $#ARGV is the subscript of the last element of the @ARGV array, so the number of arguments on the command line is $#ARGV + 1.  Situation: You want to print the contents of an entire array. How would you do that?  Step 1: Get the size of the array using the scalar context on the array. Eg. @array = (1,2,3); print ; "Size: ",scalar @array,"\n"; Step 2: Iterate through the array using a for loop and print each item. |
| Explain: Grooving and shortening of arrays and Splicing of arrays  **a.)Grooving and shortening of arrays.** Grooving and shortening of arrays can be done by directly giving a non existent index to which Perl automatically adjusts the array size as needed.  **b.)Splicing of arrays** Splicing copies and removes or replaces elements from the array using the position specified in the splice function instead of just extracting into another array. **Syntax:**  splice ARRAY, OFFSET, LENGTH, LIST |
| Explain: a.) Goto label b.) Goto name c.) Goto expr  **a.) Goto label** In the case of goto LABEL, execution stops at the current point and resumes at the point of the label specified. It cannot be used to jump to a point inside a subroutine or loop.  **b.) Goto name** The goto &NAME statement is more complex. It allows you to replace the currently executing subroutine with a call to the specified subroutine instead. This allows you to automatically call a different subroutine based on the current environment and is used to dynamically select alternative routines.   **c.) Goto expr** Goto expr is just an extended form of goto LABEL. Perl expects the expression to evaluate dynamically at execution time to a label by name |
| There are two types of eval statements i.e. eval EXPR and eval BLOCK. Explain them.  When Eval is called with EXPR, the contents of the expression will be parsed and interpreted every time the Eval function is called.  With Eval BLOCK, the contents are parsed and compiled along with the rest of the script, but the actual execution only takes place when the eval statement is reached. Because the code is parsed at the time of compilation of the rest of the script, the BLOCK form cannot be used to check for syntax errors in a piece of dynamically generated code. |
| Explain returning values from subroutines.  Values can be returned by a subroutine using an explicit return statement. Otherwise it would be the value of the last expression evaluated.  What are prefix dereferencer? List them.  When we dereference a variable using particular prefix, they are called prefix dereference.  (i) $-Scalar variables (ii) %-Hash variables (iii) @-arrays (iv) &-subroutines (v) Type globs-\*myvar stands for @myvar, %myvar. |
| Explain "grep" function.  The grep function evaluates an expr or a block for each element of the List. For each statement that returns true, it adds that element to the list of returning values.  Differentiate between C++ and Perl.  -C++ does not allow closures while Perl does -C++ allows pointer arithmetic but Perl does not allow pointer arithmetic |
| Explain: Chomp, Chop, CPAN, TK  **Chomp** A Chomp function removes the last character from an expr or each element of list if it matches the value of $/. This is considered to be safer than Chop as this removes only if there is a match.  **Chop** A Chop function removes the last character from EXPR, each element of list.  **CPAN** CPAN is the Comprehensive Perl Archive Network, a large collection of Perl software and documentation. CPAN.pm is also a module in Perl which is used to download and install Perl software from the CPAN archive.   **TK** TK is an open source tool kit that is used to build web based applications using Perl. |
| What are the logical operators used for small scale operations? Explain them briefly.  In PERL for small scale operations the user can make the use of bit-wise operators. Some of the commonly used bit-wise operators are: - operation1 & operation2 The AND operator is used to compare two bits and results in 1 if both bits are 1;  or else, it returns 0.  - operation1 | operation2 The OR operator is used to compare two bits and gives a result of 1 if the bits are  complementary; else, it returns 0.  - operation1 ^ operation2 The EXCLUSIVE-OR operator is used to compare two bits and gives a result of 1 if  either or both bits are 1; else, it returns 0.  - ~operation1 The COMPLEMENT operator inverts all of the bits of the operand.  - operation1 >> operation2 The SHIFT RIGHT operator is used to move the bits to the right, it also discards the far right bit,  and assigns the leftmost bit a value of 0.  - operation1 << operation2 The SHIFT LEFT operator simply moves the bits to the left, discards the far left bit, and  assigns the rightmost bit a value of 0. |
| Give an example of using the -n and -p option.  The -n and -p options are used to wrap scripts inside loops. The -n option makes the perl execute the script inside the loop:  while (<>)  {    # your script } The -p option also used the same loop as -n loop but in addition to it, it uses continue. The loop of -p looks like: while (<>)  {    # your script } continue  {    print; }  If both the -n and -p options are used together the -p option is given the preference. The -n and -p option can be changed by the -a and -F options. |
| Give an example of the -i and 0s option usage.  The -i option is used to modify the files in-place. This implies that perl will rename the input file automatically and the output file is opened using the original name. If the -i option is used alone then no backup of the file would be created. Instead -i.bak causes the option to create a backup of the file.  **For ex.**  perl -p -i.bak -e "s/harry/tom/g;" test.dat  This command will change all the occurrences of harry to tom in the test.dat file. The -s option enables the user to create custom switches. Theses custom switches are placed after the script name but always before any filename arguments.  **For ex.**  if ($useTR)  {    # do TR processing.    print "useTR=$useTR\n"; }  The above program can be executed using the command:  perl -s -w 17lst03.pl -useTR. |
| What are the options that can be used to avoid logic errors in perl?  In order to avoid and reduce the number of errors made by a programmer in perl he can make use of the following options: - The use of -w Command-line option: Whenever this option is used it will display the list if warning messages regarding the code. The use of this option is always strongly suggested as it greatly reduces the effort required in debugging a code. - The use of strict pragma: This pragma whenever used forces the user to declare all variables before they can be used. It forces the user to declare all variables using the my() function. It also disables the use of any symbolic dereferencing. - Using the built-in debugger: The built in debugger allows the user to scroll through the entire script / program line by line hence greatly reducing the effort required in debugging errors. |
| How can the user execute a long command repeatedly without typing it again and again?  The user can do so by making the use of the = command. The = command can be used to create aliases in perl. The aliases allows the user to define a short form for the long command which can be used anytime in a program without having the user to type a long command again and again. Instead the user simply needs to use the alias defined by him. **For ex.**  = pFoo print("foo=$foo\n");  The above command would create an alias named pFoo. By applying the above command the user is creating an alias by the name pFoo. Now whenever in the command line the user uses the pFoo at the debugger prompt it will produce the same result as typing:  print("foo=$foo\n");. |
| What are some of the key features of objects in perl?  Some of the key-points of objects that need to be remembered in perl are: - Every object are anonymous hashes: This means that most of the new() methods return a reference to a hash.  - Data type changed by bless() method: The data type of anonymous hashes is changed to the name of the class by the bless() method. - The anonymous hash are blessed: This implies references to the hash are not blessed.  - Objects belong to one class at a time: You can use the bless() function to change the ownership at any time but it is not usually preferred. - The -> operator is used to call a method associated with a class: There are two different ways to invoke or call class methods:  $item = new Invent\_item; $item = Invent\_item->new(); |
| Can inheritance be used in perl? Explain with the help of an example.  Yes perl allows inheritance in order to promote code reusability. Inheritance as the name suggests means the properties and methods of a parent class will be available to their child classes. Inheritance can be done in perl by making the use of the special array @ISA. The names of the parent classes are placed into this array. The elements of the array @ISA are always searched from left to right to check for any missing methods. **For ex.**  package A; sub foo  {    print("Inside A::foo\n"); } package B; @ISA = (A); package main; B->foo(); B->bar(); |
| How does polymorphism work in perl? Give an example.  Polymorphism in perl means that the methods defined in the base class will always override the methods defined in the parent class.  **For ex.**  package A; sub foo  {    print("Inside A::foo\n"); } package B; @ISA = (A); sub foo  {    print("Inside B::foo\n"); } package main; B->foo();  In the above example the foo() method defined in class B overrides the inheritance from class A. Polymorphism is primarily used to extend the functionality of a class that exists without changing the entire class. |
| What rules must be followed by modules in perl.  The module in perl in general terms means a namespace defined in a file. Certain modules are only collections of function. In perl the modules must follow the following guidelines: - The file name of a module must the same as the package name. - The name of the package should always begin with a capital letter. - All the file name should have the extension of "pm". - In case no object oriented technique is used the package should be derived from the Exporter class. - Also if no object oriented techniques are used the module should export its functions and variables to the main namespace using the @EXPORT and @EXPOR\_OK arrays.  The use directive is used to load the modules. |
| Write an example explaining the use of symbol tables.  In perl the symbol table is a hash that contains the list of all the names defined in a namespace. It contains all the function and variable. For every namespace the hash is named after the namespace followed by colons. **For ex.**  sub dispSymbols  {    my($hashRef) = shift;    my(%symbols);    my(@symbols);    %symbols = %{$hashRef};    @symbols = sort(keys(%symbols));    foreach (@symbols)     {        printf("%-10.10s| %s\n", $\_, $symbols{$\_});    } } dispSymbols(\%Foo::); package Foo; $bar = 2; sub baz  {    $bar++; } |
| Write the program to process a list of numbers.  The program stated below would ask the user to enter numbers upon being executed. Once the numbers are entered the average of the numbers is shown as the output.;/  $sum = 0; $counted = 0; print "Enter your choice of number: "; $number = <>; chomp($number); while ($number >= 0)  {    $counted++;    $sum += $number;     print "Enter another number: ";    $number = <>;    chomp($number); } print "$counted numbers were entered\n"; if ($counted > 0)  {    print "The average is ",$sum/$counted,"\n"; } exit(0); |
| Explain the functioning of conditional structures in perl.  In general terms a conditional structure is a specific type of program structure where an action is performed on the basis of a condition. In perl there is only one conditional structure containing "if". **For ex.**  print "Enter a number: "; $number = <>; chomp($number); if ($number-2\*int($number/2) == 0)  {     print "$number is even\n"; }  elsif (abs($number-2\*int($number/2)) == 1)  {     print "$number is odd\n";  }  else  {    print "You invented a new number!\n"; }  The above program one execution would ask the user to enter a number. Depending on the number entered the output would be if the number is odd or even. The conditional structure in perl contains the if followed by the condition stated between the round brackets and the actions or commands after that. |
| What are the two ways to get private values inside a subroutine or block?  There are two ways a user can get private values inside a subroutine or block: - Local operator: this operator can operate on global variables only. The local operator saves the value of the private variable and makes an arrangement to restore them in the end of the block.  - My operator: this operator can be used by the user to create or define a new variable. The variable created by the my operator will always be declared private to the block inside which it is defeined. **For ex.**  $a = 20;  {    local ($a);     my (@b);     $a = 10;     @b = ("ram", "shyam");    print $a;     print "@b";  } print $a;  print @b; |
| How can information be put into hashes?  When a hash value is referenced it is not created. it is only created once a value is assigned to it. The contents of a hash have no literal representation. In case the hash is to be filled at once the unwinding of the hash must be done. The unwinding of hash means the key value pairs in hash can be created using a list , they can be converted from that as well. In this conversion process the even numbered items are placed on the right and are known as values. The items placed on the left are odd numbered and are stored as keys. The hash has no defined internal ordering and hence the user should not rely on any particular ordering.  **For ex ( creation of hash)**  %birthdays = ( An => "25-02-1975", Ram => "22-12-1983", Shyam => "13-03-1989", Hari => "11-09-1991" ); |
| What can be done for efficient parameter passing in perl? Explain.  In perl aliases are considered to be faster than references as they do not require any dereferencing.  **For ex.**  @array = (10,20); DoubleEachEntry( \*array ); # @array and @copy are identical. print "@array \n";   sub DoubleEachEntry  {    # $\_[0] contains \*array    local     \*copy    = shift;     foreach $element (@copy)     {        $element \*= 2;    } }  By making the use of typeglobs to pass an array a subroutine the operation is much more efficient than compared to the used of references. |
| How can arrays be tied?  The user can work with an array at two levels. Using $#array the user can set and get the value of the entire array and the last elements index. At the other level the user can set or get the individual elements of an array. The user can also create and destroy its elements using various methods such as splice, push etc. An example of tying an array is by suing a file as an array.  **Ex:**  tie @lines, 'TieFile', 'foo.txt'; print $lines[20];  This code on execution will run till its hits the 20th line of the text file. On being asked to fetch the nth line of a file it reads the file till it reaches the specified value and returns it on finding it. The tiearray has two fields: - On field is to store the array of offsets. - The other field is to store the filehandle for opening the file. By making the use of the two fields the the tiefile keeps the track of file offsets for future references. |
| How and what are closures implemented in perl?  The closure is a block of code that is used to capture the environments where it is defined. It specifically captures any lexical variables the block contains and uses defined in an outer space.  **For ex.**  {     my $seq = 3 ;    sub sequence { $seq += 3 } }  print $seq; # this is out of scope print sequence; # this prints 6 print sequence; # this prints 9  Since the lexical variable is out of scope the $seq after the defined block does not work. But the sequence subroutine can still access and increment as the closure { $seq += 3 } has captured the $seq lexical variable. |
| What are STDIN,STDOUT and STDERR?.  All three are special file handles that can be used in perl.  -STDIN: The STDIN file handle is used to read from the keyboard. -STDOUT: This file handle is used to write into the screen or another program. Both the STDIN and STDOUT are the default file handles that are used for reading and writing.  -STDERR: This is also a file handle that is used to write into a screen.  The purpose of the output stream is for in case there is an error that occurs whose output the user does not want reflect in the output file but instead to be displayed on the screen. The STDERR is a standard error stream that is used in perl.  **For ex. A snippet of code making use of all the three handles:**  while (<>)  {     ($day, $month, $year,$page) =     m!^$host\s.\*\[(\d{2})/(...)/(....).\*(?:POST|GET)\s+(\S+)!;    if ($day)     {         if ($page !~ m!^/!)         {            print STDERR "YIKES ... $page\n";        }        print STDOUT "$day - $mname{$month} - $year .... $page\n";    } } |
| How can memory be managed in perl?  In oerl whenever a variable is used it occupies some memory space. Since the computer has limited memory the user must be careful of the memory being used by the program.  **For ex.**  use strict; open(IN,"in");  my @lines = <IN> close(IN); open(OUT,">out");  foreach (@lines) { print OUT m/([^\s]+)/,"\n"; } close(OUT);  In the above program on execution it will read a file and prints the first word of each line in another file. In case of very big files the system would run out of memory eventually.  To avoid such kind of occurrences the file needs can be divided into sections. Upon dividing a file into sections each section would be processes one by one and store a section on the disk on being completed. In this way the limited memory problem of computers can be solved. |
| Show the use of sockets for the server and client side of a conversation?  The conversation on the server side: The server side program uses the socket() function to create a socket. The bind () function is used to define a socket address to make it findable. The listen() function checks if there is anyone who want to talk. The accept() function would simply start the conversation. The call of the socket and binding of address to it :  $tcpProtocolNumber = getprotobyname('tcp') || 6; socket(SOCKET, PF\_INET(), SOCK\_STREAM(), $tcpProtocolNumber) or die("socket: $!"); $port = 20001; $internetPackedAddress = pack('Sna4x8', AF\_INET(), $port, "\0\0\0\0"); bind(SOCKET, $internetPackedAddress)  or die("bind: $!");  The conversation on the client side: The client program makes use of the socket() function to create a socket and the connect() function to initiate a conversation. The process of creation of socket is similar to that of the server side. The connect() function is called like this:  $port = 20001; $internetPackedAddress = pack('Sna4x8', AF\_INET(), $port, "\0\0\0\0"); connect(SOCKET, $internetPackedAddress) or die("connect: $!"); |