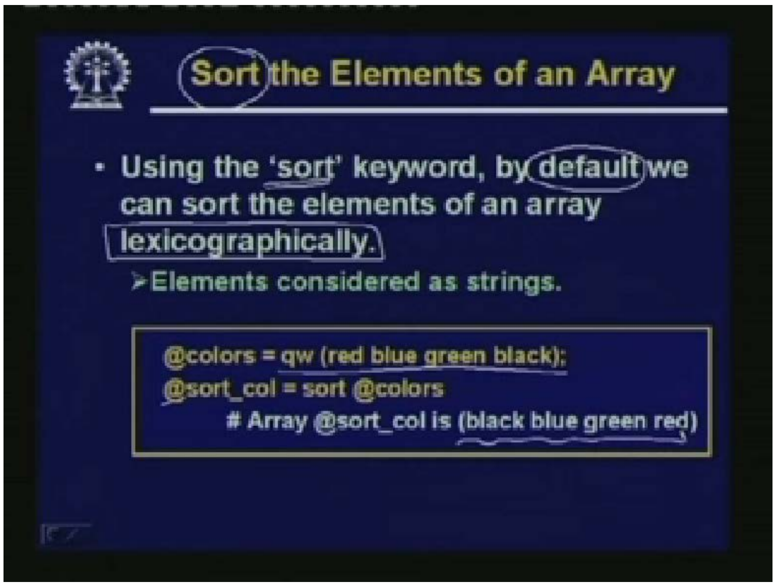
Internet Technology

Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No #22
PERL – Part II

We continue with our discussion on the Perl language and its features. If you recall in our last lecture we were talking about the scalar and the array variables in the Perl. How it can be used? Today we continue with what we are talking about.

(Refer Slide Time: 01:12)



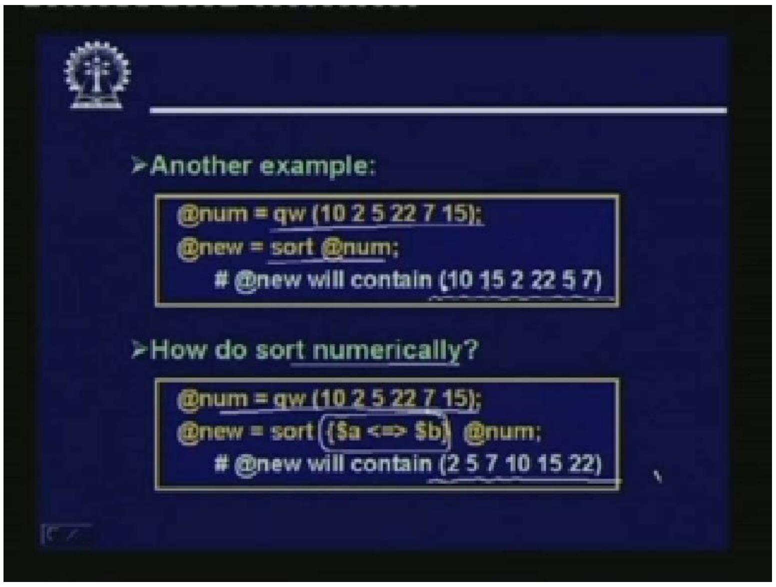
Sort the Elements of an Array

- Using the 'sort' keyword, by default we can sort the elements of an array lexicographically.
  - Elements considered as strings.

```
@colors = qw (red blue green black);
@sort_col = sort @colors
        # Array @sort_col is (black blue green red)
```

We look at some of the other facilities or other you can say features that you can have in association with an array. Let us start with sorting. Perl provides a very simple mechanism to sort the elements of an array using the sort keyword. Default sorting method is we can sort the elements lexicographically. Lexicographically means the elements of the array are treated as character strings and they will be sorted in the order of the dictionary; in the dictionary order. Dictionary sorting means lexicographic ordering. And the sort command by default considers that all elements are a string. For example if you write a number 25 it will considered as a string 2 and 5. So some example colors equal to say using the quote word command you define this array as red, blue, green, black. If you write sort col another array equal to sort colors. So the new array sort col will be this, will be sorted black, blue, green, red. So this is the correct order of sorting lexicographic array.

(Refer Slide Time: 02:43)



➢Another example:

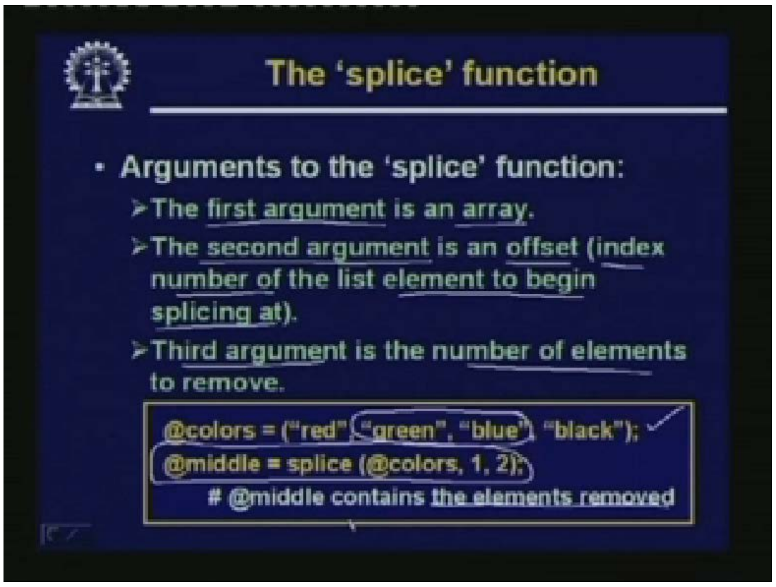
```
@num = qw (10 2 5 22 7 15);
@new = sort @num;
    # @new will contain (10 15 2 22 5 7)
```

➢How do sort numerically?

```
@num = qw (10 2 5 22 7 15);
@new = sort {$a <=> $b} @num;
    # @new will contain (2 5 7 10 15 22)
```

But there is peculiar problem you will see, if you try to sort a number array like this, suppose you had a number array which contain 10 2 5 22 7 15 and if you give sort command to sort this array now, what you will find is that you will be getting the sorted list as something like this which is not what you expect in a numerical sorting. But in lexicography order this is because in dictionary ordering. One will always come before 2. That is why 10 15 the first letter is one they will come before anything starting with 2 2 22 then 5, then 7. So the first letter is 1, then 1, then 2 2 5 7 and so on. Now if you want to sort numerically there is a way out also well will explain why this happens later. But for the timing lets look at the solution take the same array of numbers, same command, but in between you write something more curly bracket dollar a less than equal to greater than dollar b curly bracket closed this in essence tells the sort command. That sorting has to be done in the numerical order. So if you give a sorting command like this you will actually get the sorted list like this what you expect after sorting.

(Refer Slide Time: 04:30)



The 'splice' function

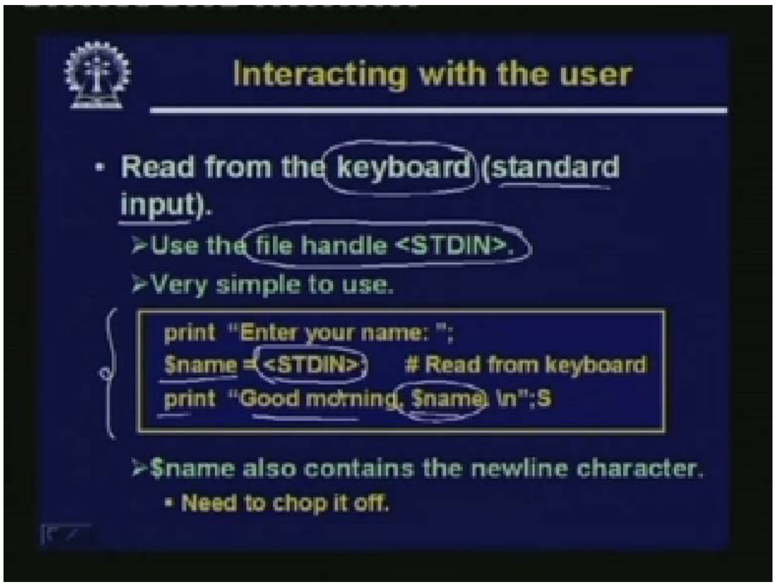
- Arguments to the 'splice' function:
  - The first argument is an array.
  - The second argument is an offset (index number of the list element to begin splicing at).
  - Third argument is the number of elements to remove.

```
@colors = ("red", "green", "blue", "black");
@middle = splice (@colors, 1, 2);
        # @middle contains the elements removed
```

Splice is the function which can selectively take out some elements of the array sometimes. If you have an array from the middle you may want to take out or delete or remove some array. Earlier we had seen using the push pop and shift this kind of functions you can access the elements at the head and the tail. But what about the middle? Suppose I want to remove to elements form the middle of an array, this splice function can be used to do that. So there are three arguments to the splice function. The first argument is the array on which you want to operate. Second argument is an offset which is actually an index number of the list constituting the array at which you are intending to begin splicing or deleting and the third argument tells you that how many number of elements you want to remove.

Let us take an example, colours red, green, blue, black, say this is the initial array then we give a command like this. Splice this array colours 1 2. What this command will do? This command will remove 2 elements starting from 1011 element is green, green and blue. This green and blue these two elements will be removed from the array colors and if you assign the value to some other array middle. Then middle will contain list of the elements removed in this case green, blue right. So this is what splice does. Now let us come to a very important feature of Perl. We will see that it really makes our life much easier for sorting certain kinds of tasks. This is file handling because in the context we are just planning to use Perl in the context of developing the server-side scripts, cgi scripts, there the handling of file is very important.

(Refer Slide Time: 07:10)



Interacting with the user

- Read from the keyboard (standard input).
  - Use the file handle <STDIN>.
  - Very simple to use.

```
print "Enter your name: ";
$name = <STDIN>;    # Read from keyboard
print "Good morning, $name \n";S
```

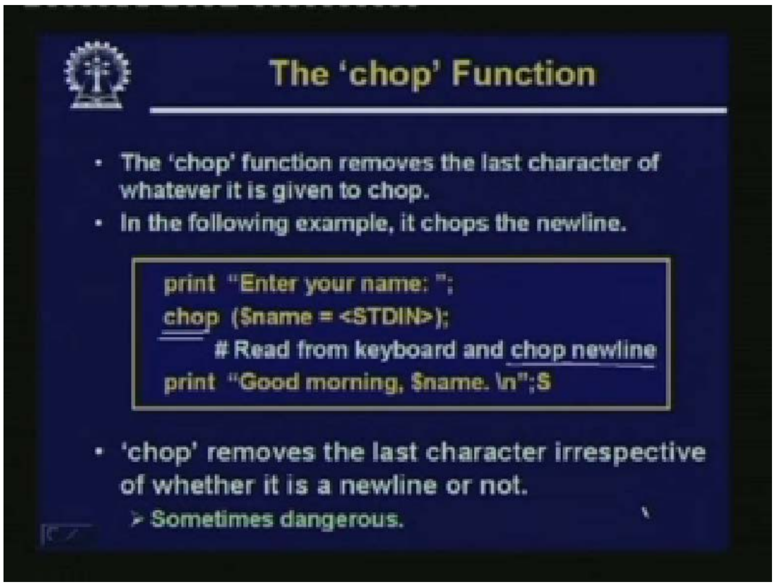
  - $name also contains the newline character.
    - Need to chop it off.

First before going in to accessing regular files on disk, we first talk about how a perl program can interact with the user by reading some data value from the keyboard. Now you may know that many operating systems treat the keyboard as a special file. It is treated as something called standard input reading from the keyboard is considered to be equivalent as reading from a file called standard input. Similarly outputting on the screen is similar to writing some data on a special file called standard output. So these are some well known features and concepts which most operating system supports. So, Perl also does something similar. When it is trying to read from the keyboard. It assumes that as if it is reading from a file standard input using a file handle within angular brackets standard in STDIN. Now in Perl you know in a language like C whenever you open a file a number or an integer is written which is called the file handle all subsequent operations on the file are carried out using that number that is called a file pointer.

That is actually technically called file handler. In Perl a file handle is indicated by enclosing it with angular brackets. So for standard input the default file handle is STDIN. Just look at this, code segment print enter your name, dollar name equal to within bracket STDIN. This STDIN appearing on the right hand side simply means that it reads something from the standard input and whatever you are reading goes in to the variable dollar name. This is as simple as that print Good morning dollar name. This is interpolation, so this name will get replaced by the value you have entered. But there is one thing to be understood here. When you are reading something from the standard input. Say take this example you are reading a name, so the user will type a name followed by the enter character the carriage return. Now whatever goes in to the variable you are assigning it to the carriage return also goes along with it.

So, here if you run this program you will see one peculiar thing. Just see this program like this that if you give this print, what you are expecting? You are expecting that it would print Good morning then the name of the person we just typed in slash n will move the cursor to the next line. This s is not there, but we will see here is that it will go to next line all right. But there will one additional blank line in between as if there are two newline characters gone in right the reason is that the name itself had one newline character at the end carriage return slash n. So there should be a mechanism to remove the newline character from the values you are reading from keyboard before you can actually process or compare with some values. So there is a need to chop off the newline character at the end.

(Refer Slide Time: 11:14)



The 'chop' Function

- The 'chop' function removes the last character of whatever it is given to chop.
- In the following example, it chops the newline.

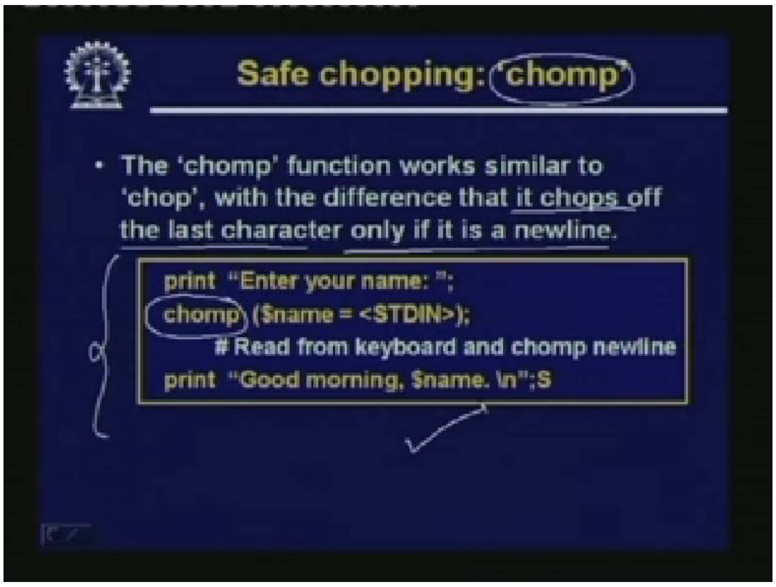
```
print "Enter your name: ";
chop ($name = <STDIN>);
        # Read from keyboard and chop newline
print "Good morning, $name. \n";$
```

- 'chop' removes the last character irrespective of whether it is a newline or not.
  - Sometimes dangerous.

There is a function chop which does exactly this, the chop function removes the last character of whatever you have given it as a parameter it is a string. A string can be a number; it can be string with some carriage, return at the end. The chop function does not check what is there at the end whatever the last character it removes is. Blindly this is the function of the chop; it removes the last character irrespective of what it is, for this specific case. Well in the earlier example, if we look at the earlier example, you are using dollar name equal to STDIN. Here we are doing the same thing, but we are enclosing it within the parameter of function chop. Chop will do the reading; it will chop the last character whatever is read.

So the newline will go and you will be getting whatever you actually want. Now as I said for this example it is but for many other examples you may not want that chop delete the chop character blindly. Because in some cases there can be a newline character in some cases some other special character also depends on the context from where the data you are giving as a parameter to chop is coming be. Because this particular example says that you are taking STDIN and then give a chop. That is, but if the data is coming from somewhere else, it is possible that the carriage return is not there, (12:59 audio not clear) something else is there or nothing is there. So this version of chop which you can is a safer version which does not delete useful characters.

(Refer Slide Time: 13:15)

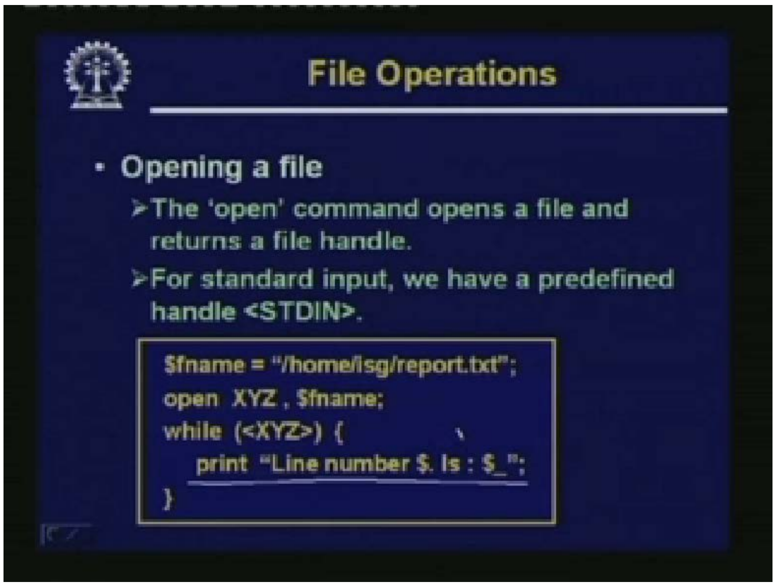

Safe chopping: chomp

- The 'chomp' function works similar to 'chop', with the difference that it chops off the last character only if it is a newline.

```
print "Enter your name: ";
chomp ($name = <STDIN>);
      # Read from keyboard and chomp newline
print "Good morning, $name. \n";S
```

This is called chomp c h o m p. This is the name of version of chop where it chops of the last character only if it is a new line. So the same program you can write by replacing chop by chomp and this is a safer version of the program. In essence that it will only chop the last character, if it is a newline otherwise it will not chop anything.

(Refer Slide Time: 13:51)


File Operations

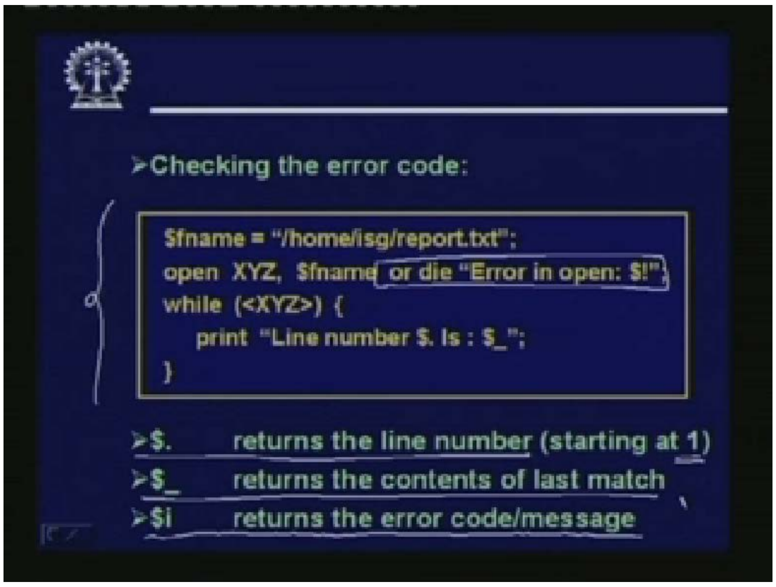
- Opening a file
  - The 'open' command opens a file and returns a file handle.
  - For standard input, we have a predefined handle <STDIN>.

```
$fname = "/home/isg/report.txt";
open XYZ , $fname;
while (<XYZ>) {
   print "Line number $. is : $_";
}
```

Now let us move on to the regular File Operations. Let us see how to open a file. Now a file in perl can be opened by using the open command and when you open a file just like a when you open a file using any other language using C, the open command returns a file handle and as the previous example showed for standard input the file handle was this STDIN. Let us take an example, here which illustrates file opening. Suppose this is file you want to open, this is under home under isg and the name of the file is report dot txt. This we are defining as a text and we are assigning to a variable fname. The syntax is open XYZ name of the file. So after opening this handle XYZ is returned, while within angular brackets XYZ. See there is a difference between XYZ and angular bracket XYZ or between STDIN and angular bracket STDIN.

See STDIN represents a file or handle that is, it is standard input. Within angular bracket STDIN means the contents of that file whatever is there. Similarly here this XYZ represents the file handle, while this within angular bracket XYZ, this means the content. That means it is reading from this file here in this while loop. So while will go on reading until the end of file is encountered line by line and it is printing something Line number. These are special variables $., $_, these we shall talk about later again; $. returns the Line number you are reading and $_ represents a string which is the contents of the present line. Whatever you are reading presently, that is stored in a temporary variable dollar underscore and the Line number is contained in $. So if you look at this program here, now it will print all the lines you are reading with a Line number $.

(Refer Slide Time: 16:56)



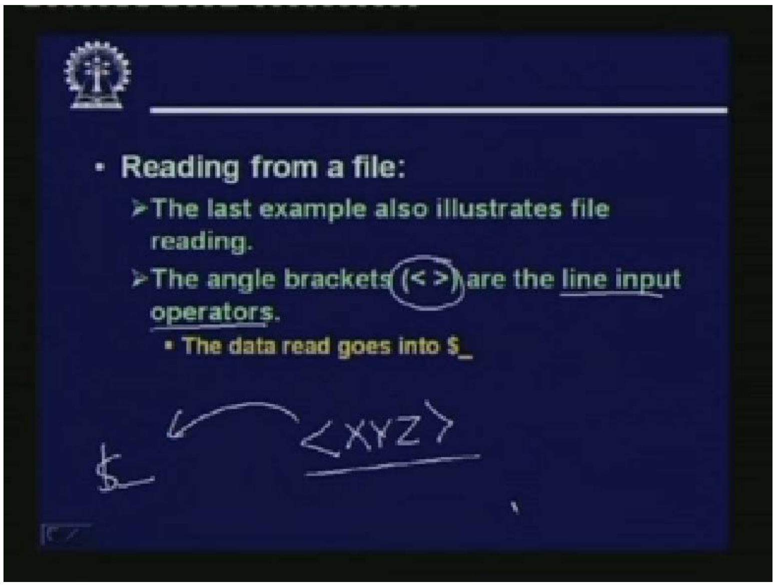
➢Checking the error code:

```
$fname = "/home/isg/report.txt";
open XYZ, $fname or die "Error in open: $!";
while (<XYZ>) {
    print "Line number $. is : $_";
}
```

➢$. returns the line number (starting at 1)
➢$_ returns the contents of last match
➢$i returns the error code/message

So this is the same program with a slightly different. You can say modification here. Here we have added this thing. This means see we are opening this file. After opening you are returning the handle, but you know while opening a file sometimes some errors happen. The file name you are specifying there is no name. With that file somehow the file is locked by someone else you are unable to open. So there can be an error message that can return in your attempt to open your file and the error message has to be sent back to the user. Now this construct in the open command does exactly that or die will be activated. Whenever there is an error message and whatever is there in the quote that will be displayed will be displayed on the standard error which is typically the screen. The Error in open colon $!. $! is again another special variable. As you can see here dollar exclamation returns the error code or message. As I said, $. returns the line number which starts at one and dollar underscore returns the contents of the last match which is the contents of the last line. So this example shows how you can just open a file and also handle errors.

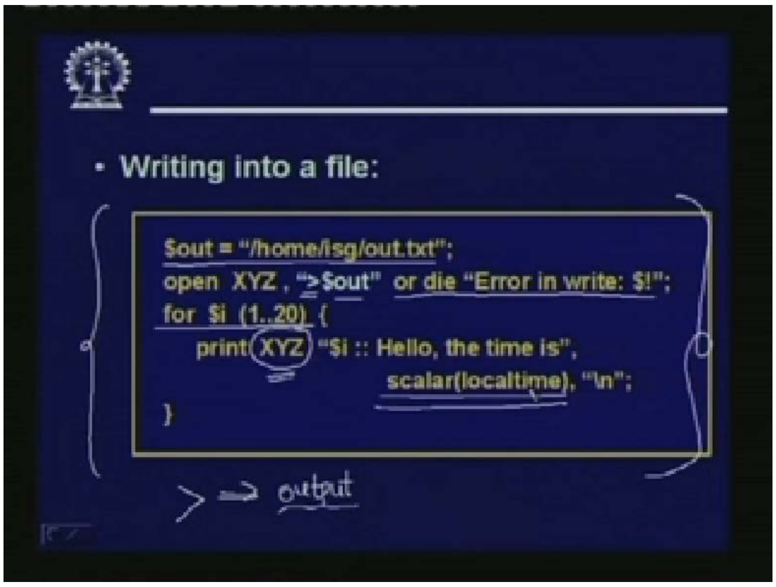
(Refer Slide Time: 18:43)



- **Reading from a file:**
  - The last example also illustrates file reading.
  - The angle brackets (< >) are the line input operators.
    - The data read goes into $_

Next comes Reading. Well after opening the file, how you can read the contents of the file. Now the last example which we have shown to illustrate open also illustrates reading because as I said that the angular brackets whenever we are using to enclose a file handle like this. XYZ this angular bracket is actually the line input operator means that you use the file handle XYZ to read a line of data and whatever you are reading, that data will go to this special variable $_.

(Refer Slide Time: 19:33)

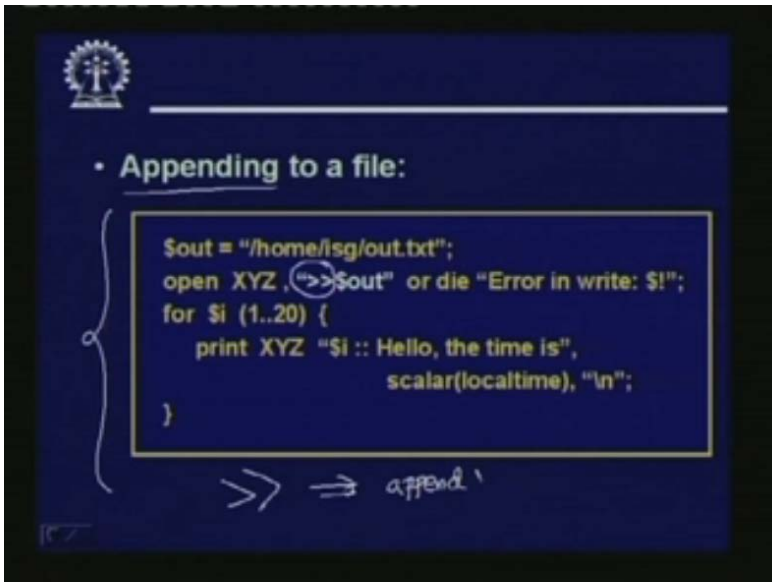


- **Writing into a file:**

```
$out = "/home/isg/out.txt";
open XYZ , ">$out" or die "Error in write: $!";
for $i (1..20) {
    print XYZ "$i :: Hello, the time is",
                  scalar(localtime), "\n";
}
```

So let us look at this version of the program. So again here instead of reading, we are doing writing the program is very simple. There is only one difference; first we are defining the output file. The file name you are opening, but we are using a greater than sign before the file name. This greater than sign, means that you are trying to open the file for output. So for writing a file you should make sure that there is a greater than sign which is put at the beginning of $ out or the name of the file. Whatever the name of the file can also be put here directly no problem or die is a standard error message. If there is any problem in write there will be error message displayed. Now the construct we shall be talking about for dollar I this 1.20 means this is a, for loop where the variable I will continue for up to 20 1 2 3 4 20 and here we are asking to print XYZ dollar I Hello the time is scalar. Local time print XYZ means here we are printing with the file handle specifier. That means you print not on the screen, but on this file $ I, the line number will be displayed Hello the time is we also learn that the scalar local time is built in function in perl which gives you the time. So actually what will get on the file, there will be 20 lines where the time is continuously displayed as the program executes. So this is just an example how you can write in to your file.

(Refer Slide Time: 21:39)



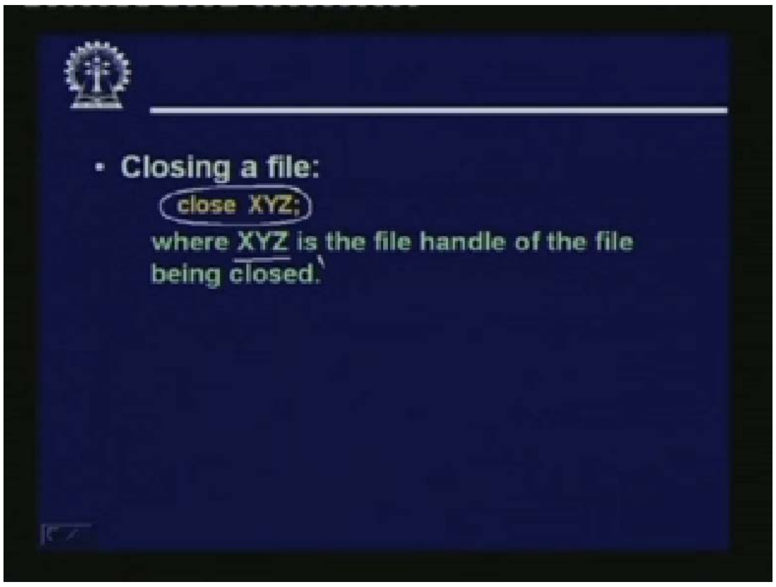
• Appending to a file:

```
$out = "/home/isg/out.txt";
open XYZ ,">>$out" or die "Error in write: $!";
for $i (1..20) {
    print XYZ "$i :: Hello, the time is",
                      scalar(localtime), "\n";
}
```

$>> \Rightarrow$ append

Writing to the end of a filer or appending is almost identical. The only difference is that instead of greater than sign, we use double greater than sign. This is the only difference. Double greater than means append. That is all the rest of the program is identical.

(Refer Slide Time: 22:07)



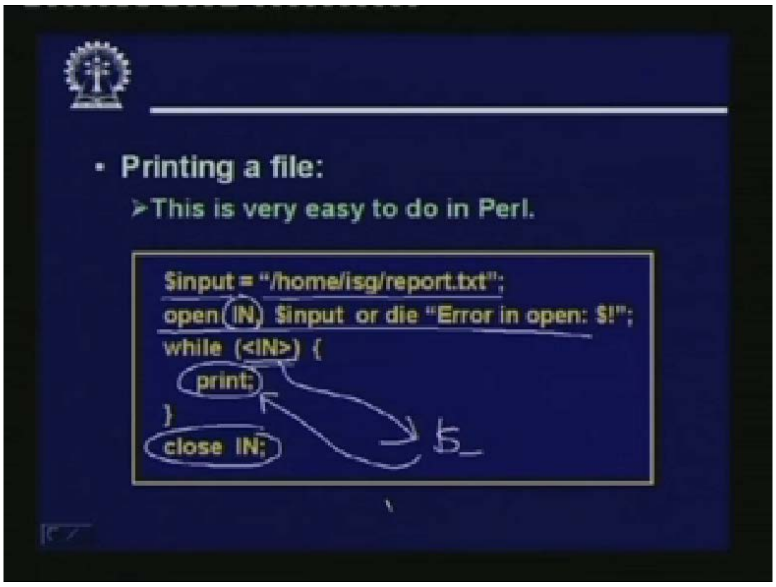
- Closing a file:

  close XYZ;

  where XYZ is the file handle of the file being closed.

Finally when you are done with your reading, writing or appending whatever we have to close the file. Closing a file, you can do using the close command close XYZ, where XYZ is the file handle.

(Refer Slide Time: 22:25)



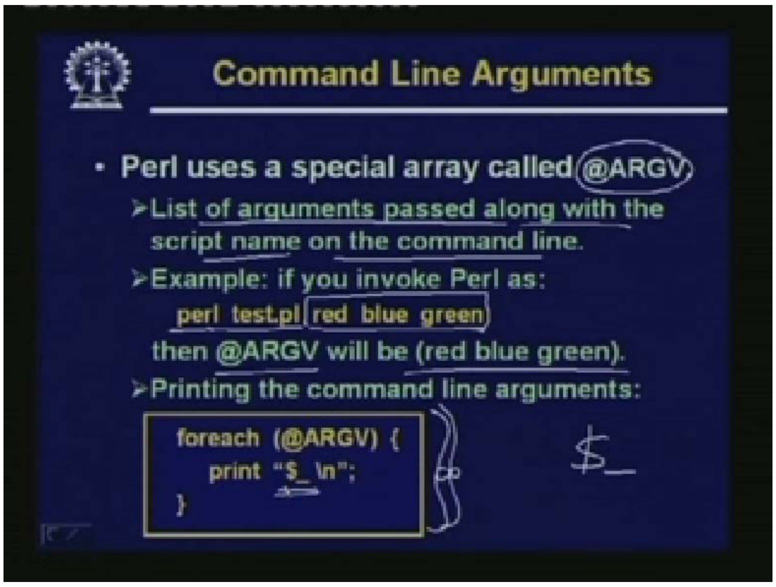
- Printing a file:
  - This is very easy to do in Perl.

```
$input = "/home/isg/report.txt";
open(IN) $input or die "Error in open: $!";
while (<IN>) {
  print;
}
close IN;
```

$\$\_$

Printing a file is very easy in Perl. This shows an example suppose I want to print this particular file home isg report.text. So I store it in a variable, I open it, IN is the file handle will in with the file input operator. This angular bracket I give this while IN means whatever comes in it gets stored in the default variable $_. Now dollar underscore is a default variable, this you should remember suppose I give a print statement without any thing to print by default it will take $_. So you just look at this program. This while IN will read the values with the variables $_, one line at a time and this print statement where you have not specified anything. This print statement will be printing values of $_ continuously line by line. Finally when it is done close it we see that just printing the contents of a file is very simple to do. In Perl you can do it very easily.

(Refer Slide Time: 23:53)



Command Line Arguments

- Perl uses a special array called @ARGV
  - List of arguments passed along with the script name on the command line.
  - Example: if you invoke Perl as:
    perl test.pl red blue green
    then @ARGV will be (red blue green).
  - Printing the command line arguments:

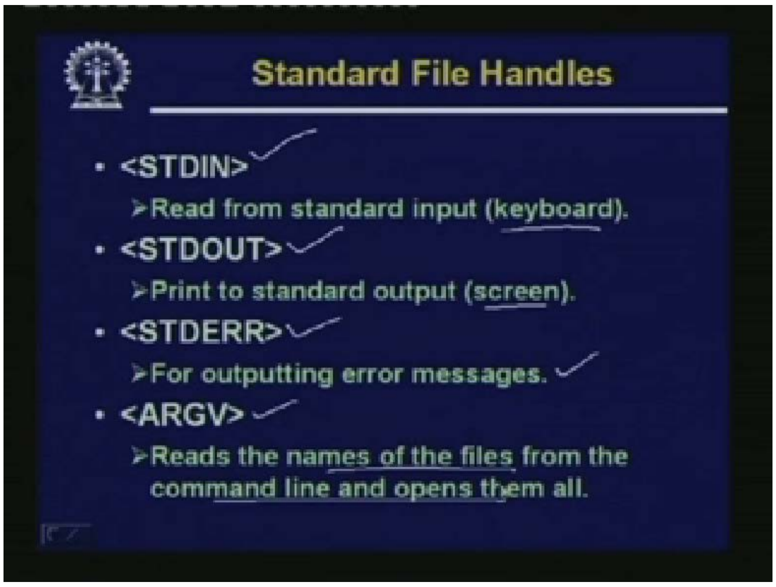
```
foreach (@ARGV) {
    print "$_ \n";
}
```

$\$\_$

There is something called Command Line Argument. Let us also have a look at this. Perl uses a special array called @ARGV. @ARGV actually means the list of arguments which are passed along with the script name on the command line. What does it mean? See normally when you run a Perl program (24:21 audio not clear) command line; you type Perl which is the name of the Perl interpreter followed by the name of Perl program. But when you talk about command line arguments we are saying that when you are running it in addition we are specifying something on the same line red, blue, green, this can be some additional parameters. This is a very common thing you do whenever you say. For example you compile a program, there are so many compiler flags are used as an optional command line arguments instead of parameters. So this example shows you how you can have access to command line parameters for your giving a command to run a Perl program.

Now this ARGV is a special array it says that after the command and file name whatever you have after that red, blue, green, the array ARGV will contain exactly that right. So if you write a short program like this, all these construct we will see later again for each within parenthesis at ARGV. This means that you take all the elements of ARGV one at a time and again when you take it one at time the default will be there in $_. You print either simple print or print $_ explicitly reverse n. Because if you had printed only print then all the things should print in the same line. This new line would not come if you want to print one in each line, then explicitly you have give dollar underscore followed by newline. So this small program snippet will print the contents of ARGV one in each line s.

(Refer Slide Time: 26:17)

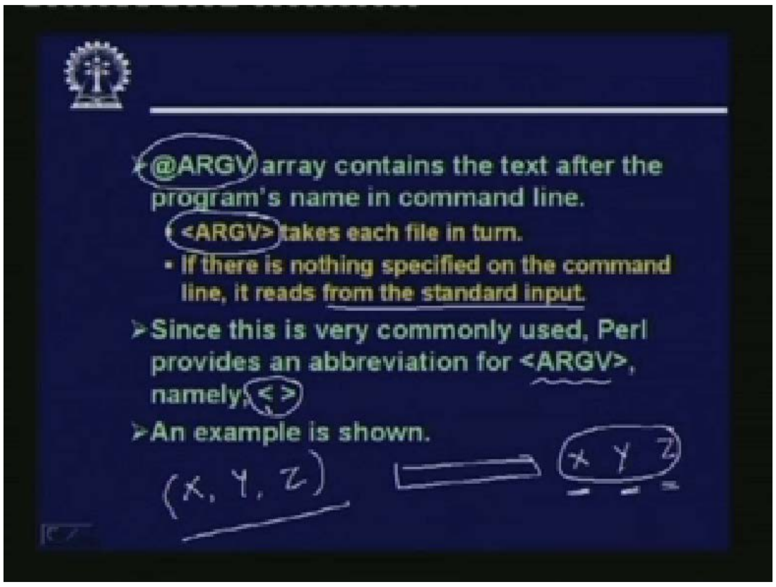


Standard File Handles

- <STDIN>
  - Read from standard input (keyboard).
- <STDOUT>
  - Print to standard output (screen).
- <STDERR>
  - For outputting error messages.
- <ARGV>
  - Reads the names of the files from the command line and opens them all.

Some standard file handles we have already talked about a couple of them. This STDIN refers to the standard input which usually means the keyboard. STDOUT, standard output which usually means the computer screen. This STDERR is the standard error; this is used for outputting error messages by default. This is also the screen and ARGV as have just mentioned this represents the command line arguments. It reads the names if the files from the command line and opens them all.

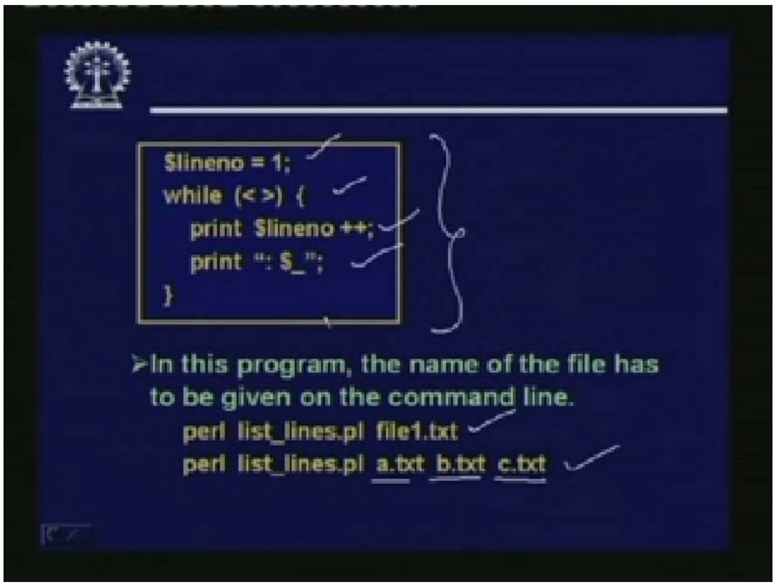
(Refer Slide Time: 27:01)



- @ARGV array contains the text after the program's name in command line.
  - <ARGV> takes each file in turn.
    - If there is nothing specified on the command line, it reads from the standard input.
- Since this is very commonly used, Perl provides an abbreviation for <ARGV>, namely, < >
- An example is shown.

See here there is something to understand. See suppose I give a command to run a program. After that I give some command line arguments X Y Z. That said this X Y Z as a list will go in to the ARGV array right. Now there is a difference the array ARGV and the file handle ARGV. See array ARGV we represent by @A R G V and the file handle ARGV you represent as within angular bracket ARGV. There is a very distinct difference between these two notations. This, array ARGV in this case will contain simply X, Y, Z, the list of elements that you have supplied. Now when you are accessing ARGV with this angular brackets then you are assuming that X, Y and Z are names of some files. So this angular ARGV will start reading from this file itself directly and if no file is specified and still you are trying to use ARGV. Then by default it will start reading from the standard input. Now this, a very commonly used feature that is why some times instead of explicitly using ARGV. You can use an abbreviation just the angular brackets with nothing in between. There is a small example to show.

(Refer Slide Time: 28:56)



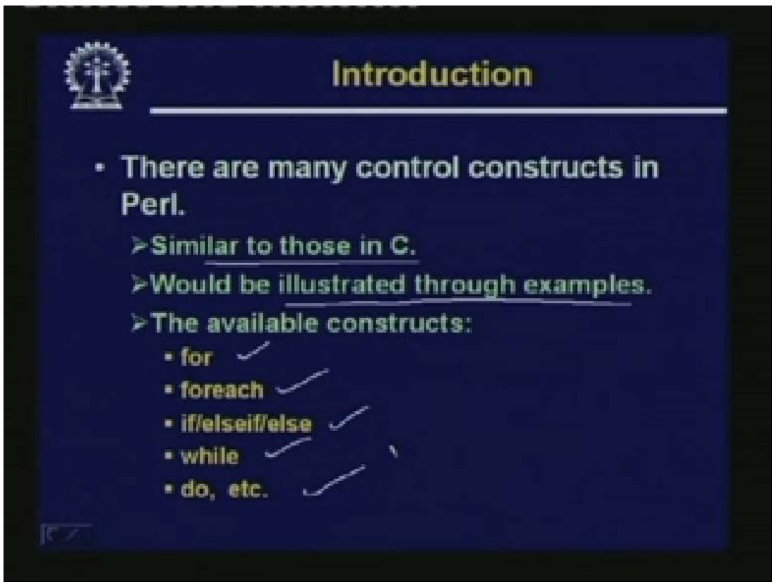
```
$lineno = 1;
while (< >) {
   print $lineno ++;
   print ": $_";
}
```

- In this program, the name of the file has to be given on the command line.

```
perl list_lines.pl file1.txt
perl list_lines.pl a.txt b.txt c.txt
```

Line number equal to one, while means here you are reading from ARGV. Print line number and print. So this program will print the contents of the file you are passing as an argument. Suppose you are running the program like this, then your array ARGV will be file1.text and within angular bracket ARGV means contents of file1.text that would be read one line at a time. Similarly for the next one the contents of a .textb, .textc, .text all will be read one after the other. So this program will display the contents of all the files you have specified in the command line. As you can see in Perl, you can do such a complex thing in such a simple and easy way. This means actually is the power or beauty of Perl you can do. So many complex things including string handling in a very easy and convenient way. So before going further into d, other constraints and the other interesting things about Perl. Let us look at some of the very standard things like the control structures. Just have a quick look at the Control Structures which Perl supports.

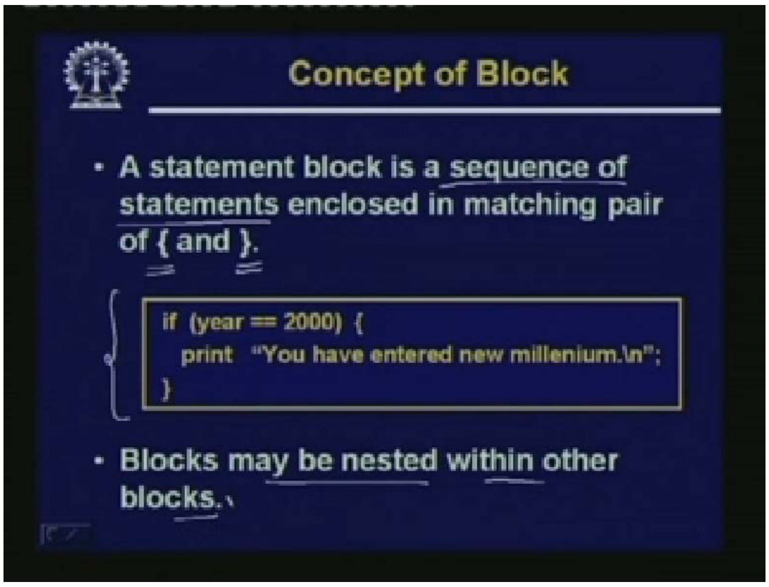
(Refer Slide Time: 30:25)


Introduction

- There are many control constructs in Perl.
  - Similar to those in C.
  - Would be illustrated through examples.
  - The available constructs:
    - for
    - foreach
    - if/elseif/else
    - while
    - do, etc.

There are many control structures which are similar in syntax to those in the language C. We would mainly be showing them through examples. There are constructs like for, foreach, if, Else, some small variations while, do, etcetera. So we shall see that what these constructs are you may be familiar with the names. But how they can used in perl this we shall see.

(Refer Slide Time: 30:58)


Concept of Block

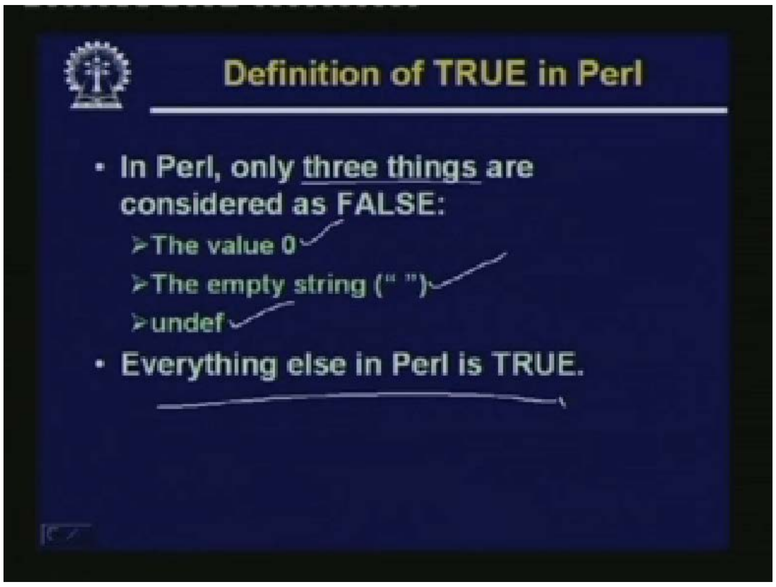
- A statement block is a sequence of statements enclosed in matching pair of { and }.

```
if (year == 2000) {
   print "You have entered new millenium.\n";
}
```

- Blocks may be nested within other blocks.

First we talk about a statement block whose definition is identical to what is there in see. So a statement block is a sequence of statements which are enclosed in matching pair curly bracket open and curly bracket close. Just an example if year double equal to 2000 curly bracket print. You have entered new millennium, new line. This looks very much like your C program segment only. The syntax of the print is slightly different. Blocks again like C may be nested within other blocks.

(Refer Slide Time: 31:45)

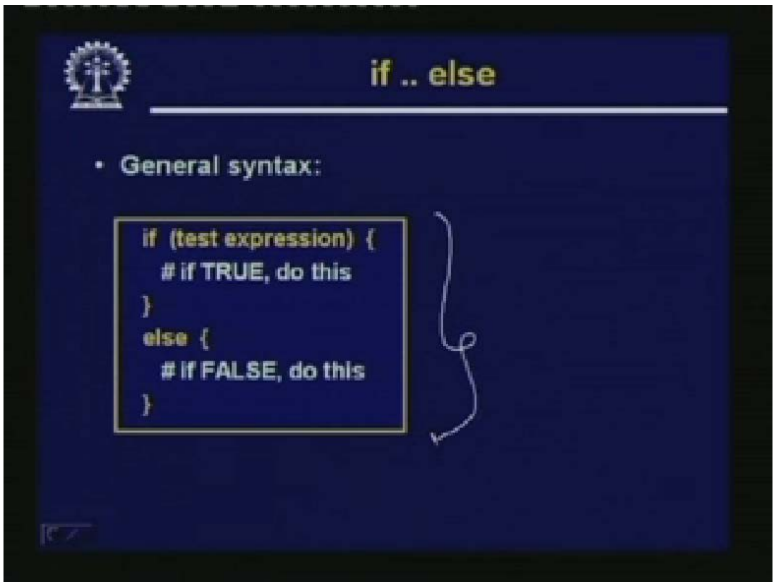


Definition of TRUE in Perl

- In Perl, only three things are considered as FALSE:
  - The value 0
  - The empty string (" ")
  - undef
- Everything else in Perl is TRUE.

Now when we talk about conditional statements, say if then else normally if condition, then else something we are evaluating. If it is true we are taking some decision, if it is false we are taking some other decision. Now let us try to understand what is meant by true and what is meant by false. Now in some language the zero value is treated as false and any non-zero value is treated as true. Let us see what is the convention followed in Perl. In Perl, there are three things which are considered as false. The value zero, if it is a string; the empty string, if it is undefined variable; the undef value. Everything else in Perl is considered TRUE.

(Refer Slide Time: 32:42)



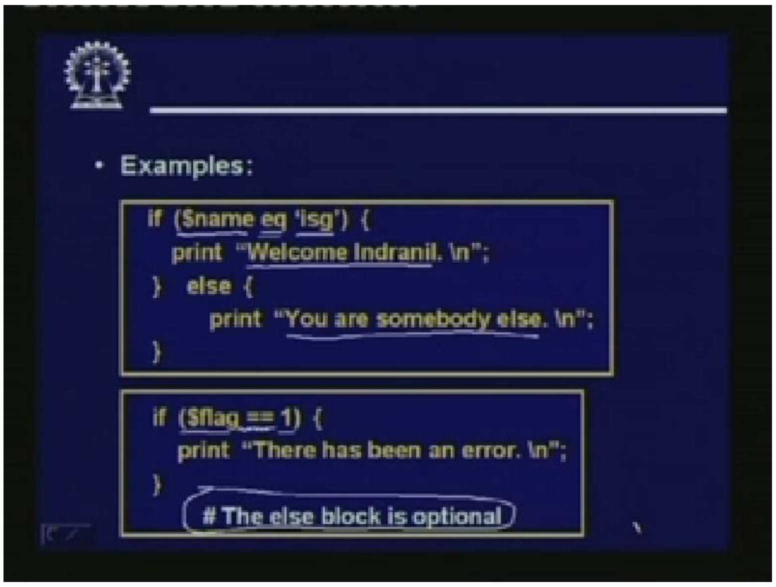
if .. else

- General syntax:

```
if (test expression) {
  # if TRUE, do this
}
else {
  # if FALSE, do this
}
```

Let us start with the syntax of the if else statement. The general syntax is if within brackets, some test expression within quote else within quote. So this is identical to syntax in C.

(Refer Slide Time: 33:00)



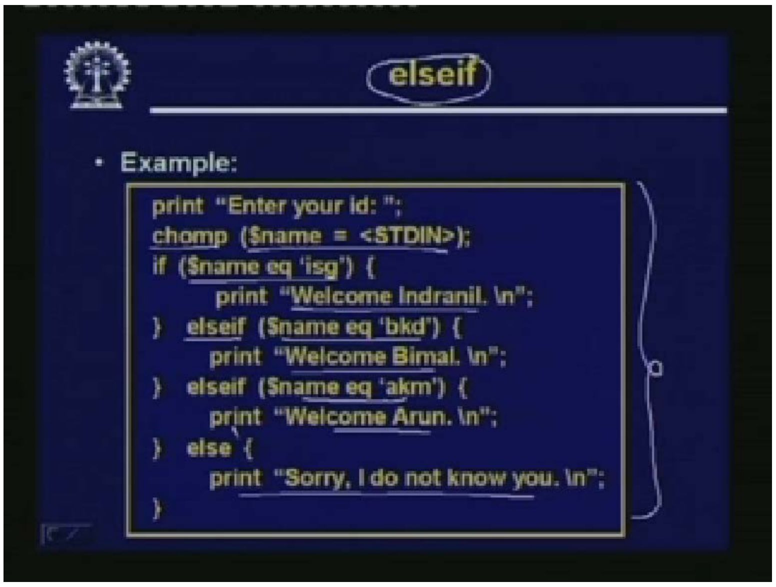
- Examples:

```
if ($name eq 'isg') {
   print "Welcome Indranil. \n";
}  else {
      print "You are somebody else. \n";
}
```

```
if ($flag == 1) {
   print "There has been an error. \n";
}
   # The else block is optional
```

Some small examples, but there is some small difference as this example will tell you the first example. Here we are comparing a variable with a string using the comparison operator eq whether this is equal to isg. If it is so, then we are printing a welcome message, else you are printing some other message. The second example, this shows two things. First it shows that that we can use if statement without else also else block is optional. Second thing is in this case we are comparing a variable with a number. You see here the comparison operator instead of eq we used the C like double equal. So this also tells you that the operator we have used for comparisons. For these kinds of statements differ depending whether the numbers or elements you want to compare are numbers or strings for equality. For example equality check numbers use double equal, for strings we use eq, the elseif, elseif is a variation you can have along with the if statement.

(Refer Slide Time: 34:44)



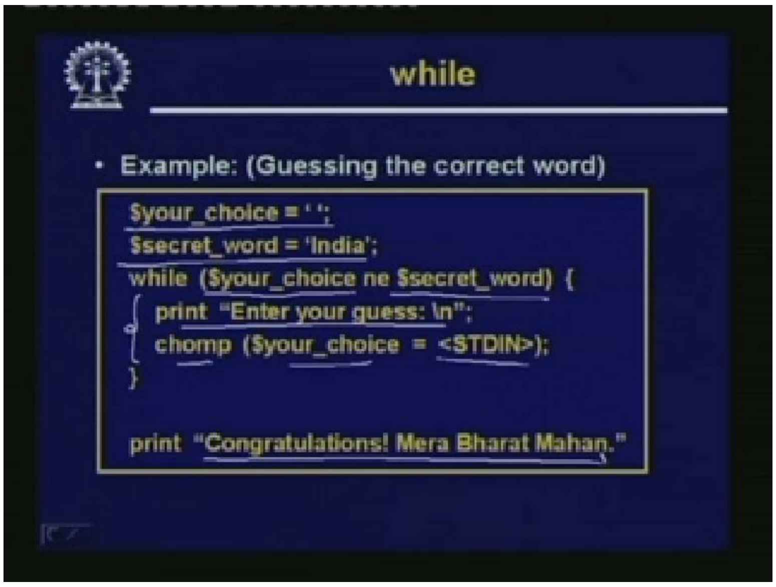
elseif

- Example:

```
print "Enter your id: ";
chomp ($name = <STDIN>);
if ($name eq 'isg') {
      print "Welcome Indranil. \n";
}  elseif ($name eq 'bkd') {
      print "Welcome Bimal. \n";
}  elseif ($name eq 'akm') {
      print "Welcome Arun. \n";
}  else {
      print "Sorry, I do not know you. \n";
}
```

Just a simple a program segment I am showing, it starts with a print, Enter your id and you are reading some user id from the keyboard. User is entering and you are using chomp to cut of the new line. Then we are comparing whether the id is isg, bkd or akm and accordingly you are welcoming the corresponding person. If dollar name equals isg, you are printing Welcome Indranil elseif, elseif is a keyword. If it is equal to bkd you are welcoming Bimal elseif, if it is akm you are welcoming Arun, else you are printing with a Sorry message. So in this way if then else is quite similar to the C syntax. But there is some small difference. This elseif must be together no space in between.

(Refer Slide Time: 35:40)



while

- Example: (Guessing the correct word)

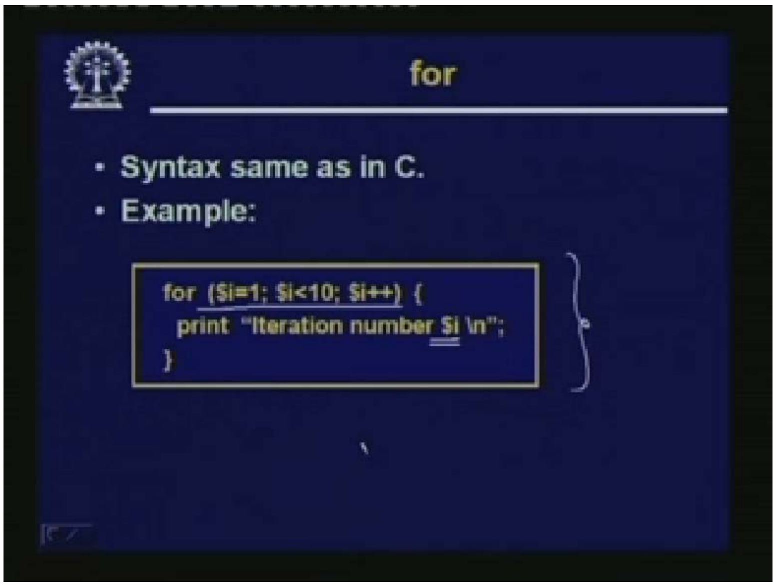
```
$your_choice = ' ';
$secret_word = 'India';
while ($your_choice ne $secret_word) {
   print "Enter your guess: \n";
   chomp ($your_choice = <STDIN>);
}

print "Congratulations! Mera Bharat Mahan."
```

So these are standard features. While well I am not trying to explain the syntax of while because this is very similar to that in C. But rather would like to show an example where the context in which such a while statement is typically used can be illustrated see in C. You can use while statement to compute the factorial of a number. For example when you are multiplying numbers in iterations but Perl is mainly meant for string manipulation. Let us see how we can use while for string manipulation or some kind of string related operation. Well this a simple application where I am just assuming that I have a secret word called India and I am asking the person. The person whose is running this program to guess this secret word.

Now as long as the correct guess is not given the program continues running in the while loop. So let us see first is that the choice the user enters is initially set to null and the secret word is India. There is a while statement while your choice not equal to secret word, for the first time it will obviously not be equal because one is India then print Enter your guess. Chomp your choice STDIN, so you are reading the choice from the standard input and you are continuously comparing in the while loop if it is equal or not equal to the secret word. Now as soon as these while loop condition is false there are equal it will come out and will be printing a message Congratulations Mera Bharat Mahan.

(Refer Slide Time: 37:52)



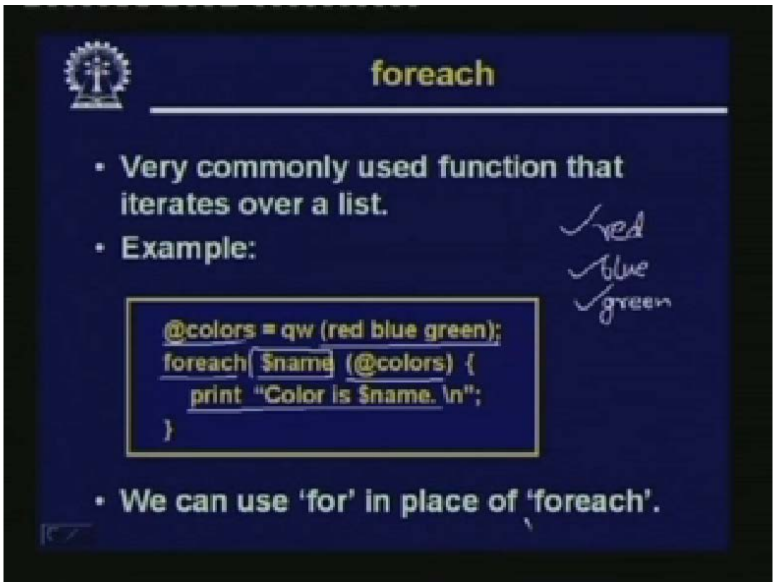
for

- Syntax same as in C.
- Example:

```
for ($i=1; $i<10; $i++) {
  print "Iteration number $i \n";
}
```

For syntax is similar to C. But you can use for in a number of different contexts you can see. This example shows the context where the usage is identical to that in C. We use a variable dollar I we have initial value, the termination value and the increments and you can. Just use the value of dollar I (38:20 audio not clear). This is a simple example foreach.

(Refer Slide Time: 38:25)



foreach

- Very commonly used function that iterates over a list.
- Example:

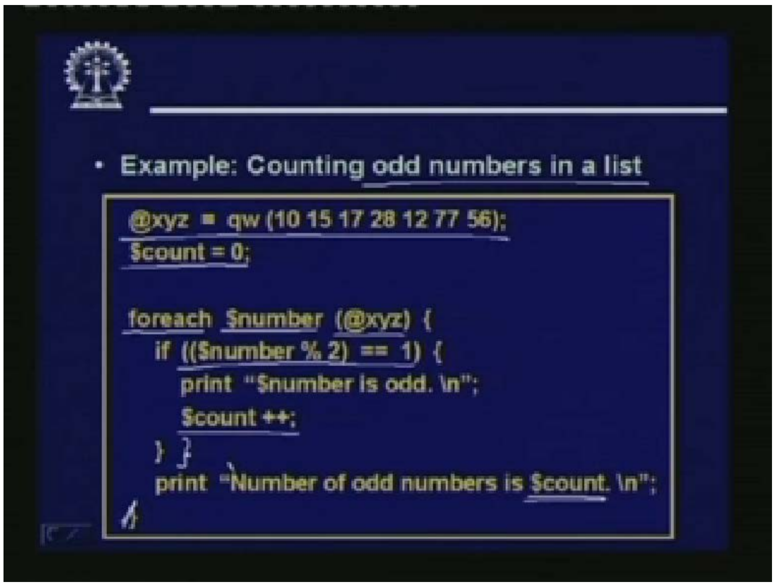
```
@colors = qw (red blue green);
foreach $name (@colors) {
   print "Color is $name. \n";
}
```

- We can use 'for' in place of 'foreach'.

✓red
✓blue
✓green

Foreach is mentioned as a constant, as a construct which iterates over a list. Suppose colors is an array which is defined as a list red blue green. If you give for each variable name within bracket array, this construct means that you will be looping over all the elements of this array since there are three elements red, blue and green. So you will be looping three times once for red, once for blue, once for green and the current value that you are at this particular variable will be getting that value. First this dollar name will get red, second time it will get blue, second time it will get green. So within the loop your printing Color is dollar name. So the colors get printed. But the thing to note is that although for and foreach had been mentioned, as two separate construct or commands. But you can use them interchangeable. For you can use here, foreach you can use in the previous case, but some of the Perl authors they have provided some guidelines that you better use for those kind of application where you are counting the number of iterations like in conventional for loops. We use foreach for this kind of list based operations where you are accessing the elements of the list.

(Refer Slide Time: 40:14)



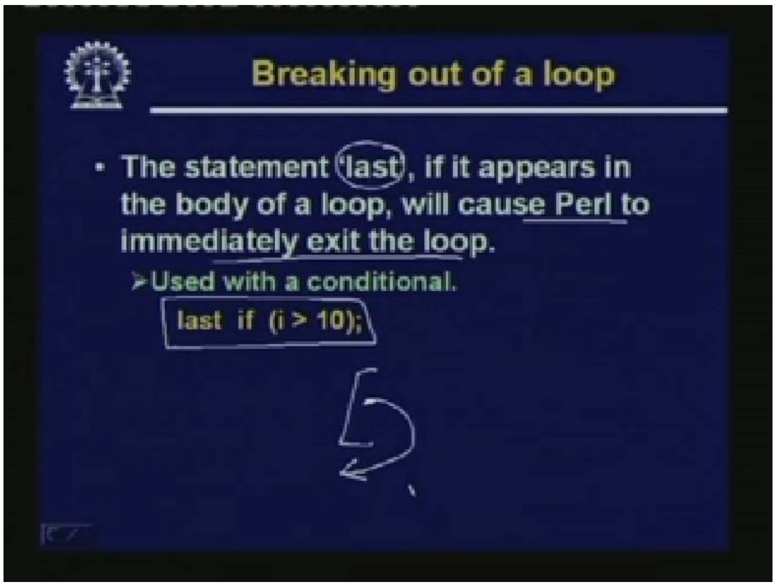
- Example: Counting odd numbers in a list

```
@xyz = qw (10 15 17 28 12 77 56);
$count = 0;

foreach $number (@xyz) {
  if (($number % 2) == 1) {
    print "$number is odd. \n";
    $count ++;
  } }
  print "Number of odd numbers is $count. \n";
```

So let us take an example here. This is an example which counts the number of odd numbers in a list. The first line defines the initial value of the array, defines the array. This at XYZ equal to 10 15 17 28 12 77 56. This is the count value which is initialized to 0 which I am counting foreach scalar array. So you are reading the numbers just storing them in dollar. Number one at a time using the modulus operator I am checking if it is odd or not. If modulus 2 equal to 1, it means number is odd. So I am printing number is odd dollar number is odd I am printing the actual number 15 17 77 3 such numbers are there and I am also incrementing the value of the count. So after the loop is finished, number of odd numbers is dollar count. Now if you basically put this inside this loop, then this line will be printed once for every element as it did as it goes on counting. But if you want this number to be printed only at the end total number three are found, then this parenthesis you have to give it before (41:50 word not audible) then the print statement.

(Refer Slide Time: 41:55)



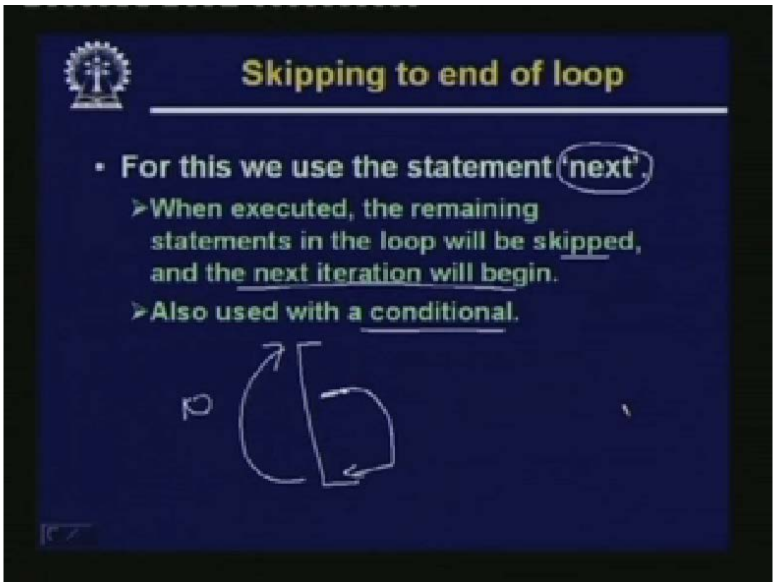
Breaking out of a loop

- The statement 'last', if it appears in the body of a loop, will cause Perl to immediately exit the loop.
  - Used with a conditional.

    `last if (i > 10);`

Now for breaking out of loop. Suppose you have a loop here some condition, based on condition you want to come out of the loop there is a statement called last. If it appears in the body of a loop will cause Perl to immediately exit the body of the loop and the condition can be embedded along with the use of last like this. Last if I greater than 10, for example, so you use a conditional element last if the condition is true then you exit the loop.

(Refer Slide Time: 42:32)

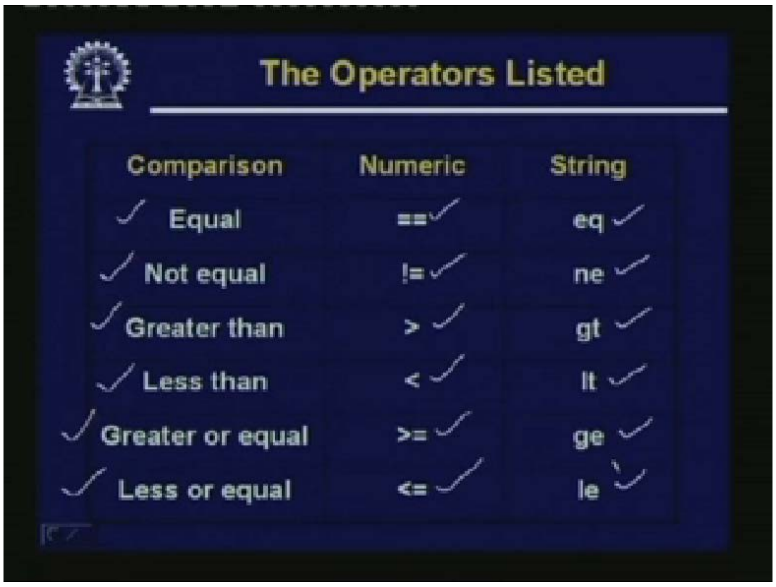


Skipping to end of loop

- For this we use the statement 'next'.
  - When executed, the remaining statements in the loop will be skipped, and the next iteration will begin.
  - Also used with a conditional.

Similarly you may want to skip the body of a loop. For example you have a loop, this loop is an iterative loop, this loop will go on. Say for example 10 times for iteration you are making some check. If some condition is true or true then you want to skip the remaining statements in the loop and continue with the next iteration not exit the loop. So you have a construct called next here when executed the remaining statements in the loop will be skipped and the next iteration will begin. So the usage of next is also identical you will have to use with a conditional next followed by a condition. Now let us quickly look at the different Relational Operators that is supported by Perl. As I said some of the Relational Operators are identical to that in C. In some cases it is different because as we have seen in one example that the operators we have used for comparison is different for numbers is different from string. In C you can only compare numbers you cannot compare strings for strings you have to use string comparison function string cmp or something, but here you can.

(Refer Slide Time: 44:05)

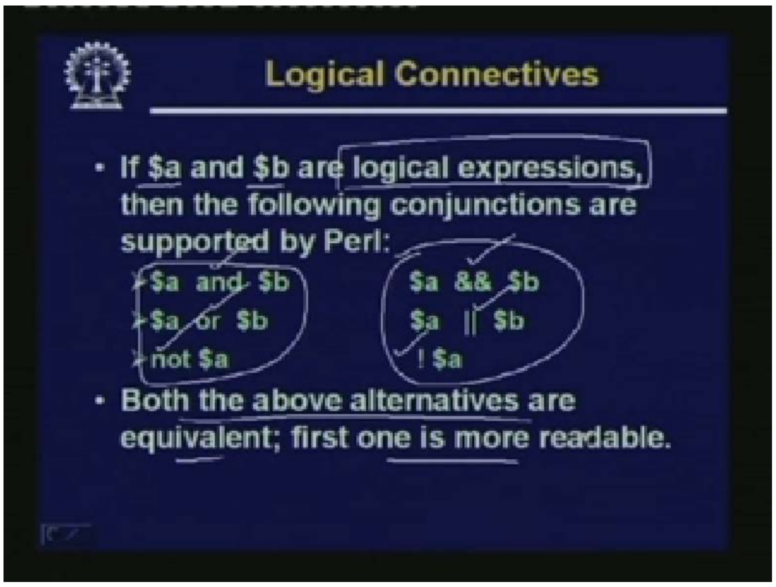


The Operators Listed

Comparison	Numeric	String
✓ Equal	== ✓	eq ✓
✓ Not equal	!= ✓	ne ✓
✓ Greater than	> ✓	gt ✓
✓ Less than	< ✓	lt ✓
✓ Greater or equal	>= ✓	ge ✓
✓ Less or equal	<= ✓	le ✓

So the Operators Listed in a tabular form. Here there are 6 comparison operations. Equal, not equal, Greater than, Less than, Greater or equal, Less or equal, for numbers these are identical to what is available in C. But for string here we use mnemonics eq, ne, Greater than, lt, less than, ge, Greater or equal, le, less equal. So when you are using comparison operator the context should be clear to you depending on the context you are using either one of these. Either the once for numeric or the once for the strings.

(Refer Slide Time: 44:55)



Logical Connectives

- If $a and $b are logical expressions, then the following conjunctions are supported by Perl:

```
$a and $b        $a && $b
$a or $b         $a || $b
not $a           ! $a
```

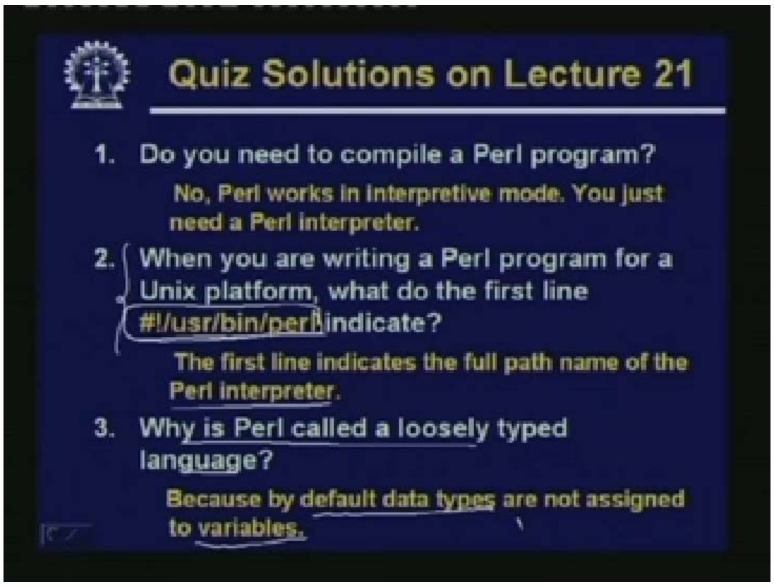
- Both the above alternatives are equivalent; first one is more readable.

Now in addition to the operators you have some Logical Connectives also. The Logical Connectives will allow you to define more complex relational expressions. Suppose I can write if a equal to b and b equal to c then do this if a equal to 10 and b greater than 5 to this. So I can combine this and or of a number of smaller relational condition to build up a complex condition. So for this you need something called Logical Connectives and Logical Connectives there are three such. These are defined as follows. If dollar a and dollar b are logical expressions. Logical expressions means they have values of true or false then the following conjunctions are supported by Perl. Well there are two styles there are identical equivalent. The second style is the C style where we can use double ampersand for AND double bar for OR and exclamation for NOT.

But this is much more English like and easy to understand and or and not. It is recommended that we use this style in Perl. So that the program becomes easier to read and understand. As I said both the above alternatives are equivalent, but the first one is more readable and so it is recommended that you use the first one in your programs are applications. Now to summarize in today's lecture we had looked at a few things. We had looked at some operations on files we had looked at some of condition operation operators that are supported by Perl and the relational operators we have had a very quick look at it. Now using these, I think you are now in a position to start writing small Perl programs. Now I strongly recommend that means I have given a number of examples in the course of this lecture. You try out these lectures yourself. You run them on your machine by installing Perl as I had mentioned earlier and you try to find out what is actually coming.

What are the variations do to the program you can play around with? You can learn a lot that way. Well I am only trying to give you guideline, what are the things that Perl supports and how they are. But it is up to you to run programs to write some more programs to play with it and understand the integrities of the construct and concepts. Now in the next class we would be continuing with our quest for newer features of the language Perl. We will see that there are number of other interesting features in the language in particular there is something called regular expressions which is considered to be one of the most important features of Perl. It is slightly complex but we will try to look in to them and try to understand how to use them. So we have come to the end of today's lecture whatever we have discussed. Let us now move on to the solutions of the questions we had posed last time.

(Refer Slide Time: 48:47)

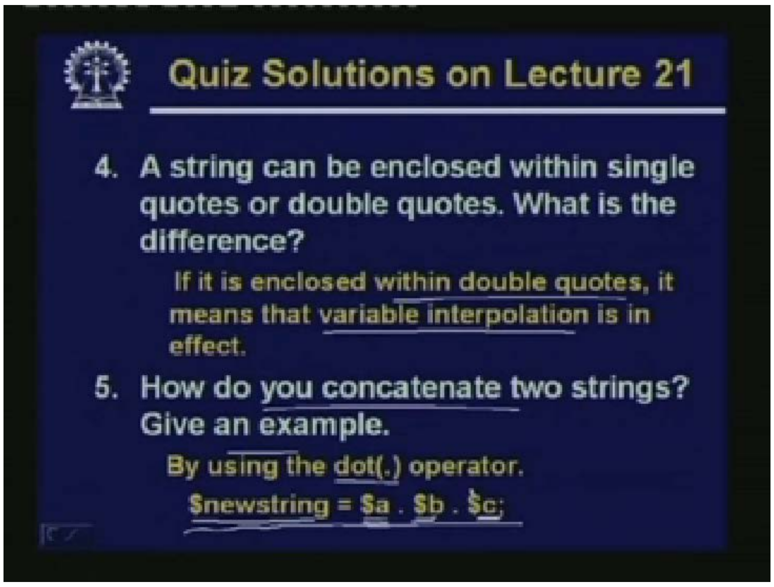


Quiz Solutions on Lecture 21

1. Do you need to compile a Perl program?
   No, Perl works in interpretive mode. You just need a Perl interpreter.
2. When you are writing a Perl program for a Unix platform, what do the first line #!/usr/bin/perl indicate?
   The first line indicates the full path name of the Perl interpreter.
3. Why is Perl called a loosely typed language?
   Because by default data types are not assigned to variables.

Number 1. Do you need to compile a Perl program?
The answer is No. Perl need not have to be complied; we had mentioned that Perl works in interpretive mode. You just need a Perl interpreter which can accept the Perl source code directly and can execute the source code statement between statements you do not need a separate compiler and compilation step for executing.
Second. When you are writing a Perl program for a Unix platform, what do the first line indicate?
Now this we had mentioned at the very beginning when we are talking about the way to run a Perl program. This actually means that this is the path name. That means the user bin perl path name of Perl interpreter. This is a very generalized concept instead of perl interpreter you can give the name of some other program which will start running whenever you run such a way file.
Why is Perl called a loosely typed language?
It is called loosely typed because the data types, the concept data types is not present and you are not defining any variable with any data types by default. So whatever value here you are assigning a variable takes that particular type.

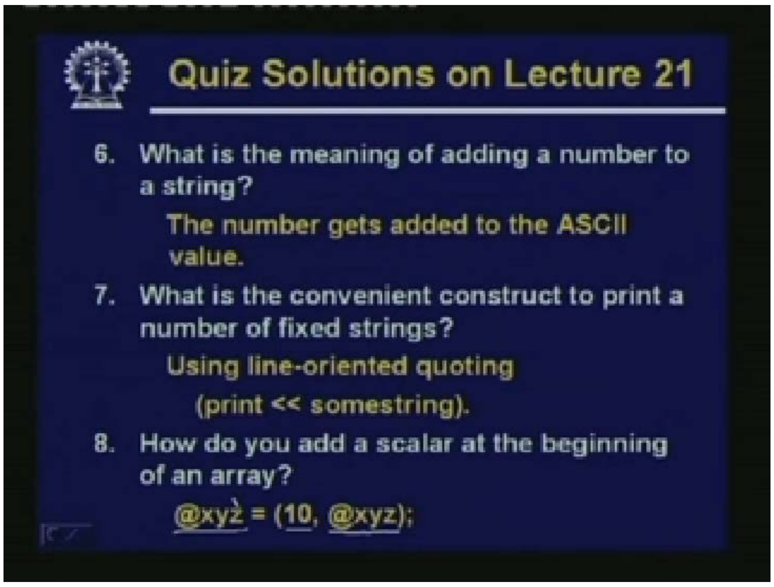
(Refer Slide Time: 50:31)


Quiz Solutions on Lecture 21

4. A string can be enclosed within single quotes or double quotes. What is the difference?
   If it is enclosed within double quotes, it means that variable interpolation is in effect.
5. How do you concatenate two strings? Give an example.
   By using the dot(.) operator.
   $newstring = $a . $b . $c;

A string can be enclosed within single quotes or double quotes. What is the difference?
This we have mentioned repeatedly if a string is within double quotes. Then variable interpolation is in effect and any variable which is included in this string will get expanded by their value.
How do you concatenate two strings? Give an example.
Concatenation can be done by the dot operator we have giving an example in the class. Also several scalar variables you can concatenate by putting dot in between. So you get a new string as a composition or concatenation of the three.

(Refer Slide Time: 51:19)

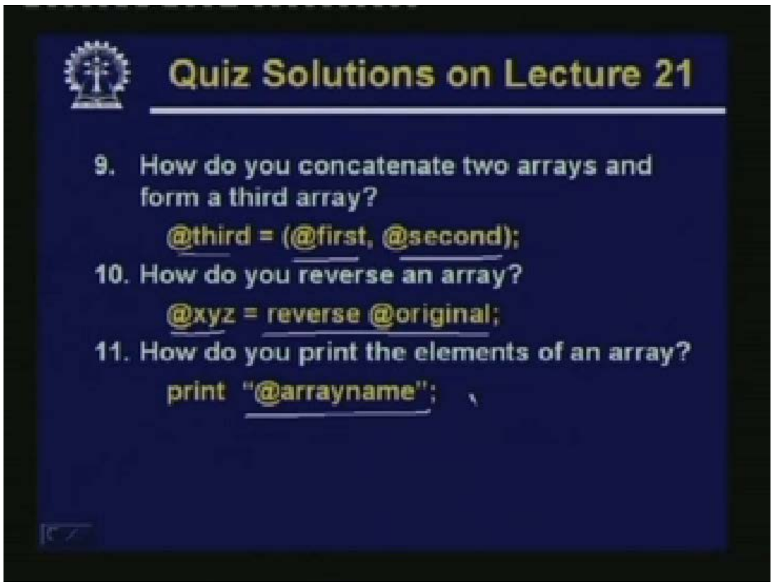

Quiz Solutions on Lecture 21

6. What is the meaning of adding a number to a string?
   The number gets added to the ASCII value.
7. What is the convenient construct to print a number of fixed strings?
   Using line-oriented quoting
   (print << somestring).
8. How do you add a scalar at the beginning of an array?
   @xyz = (10, @xyz);

What is the meaning of adding a number to a string?
This I have said adding means you are actually adding the number to the ASCII value of the string.

What is the convenient construct to print a number of fixed strings?
Well you can give print a number of times, but this line-oriented quoting is much more convenient approach this I had mentioned.
How do you add a scalar at the beginning of an array?
By constructing a list with this scalar to be added at the beginning followed by the array. This if you assign it to the scalar xyz. This 10 will get added in the beginning.

(Refer Slide Time: 52:00)



Quiz Solutions on Lecture 21

9. How do you concatenate two arrays and form a third array?

```
@third = (@first, @second);
```

10. How do you reverse an array?

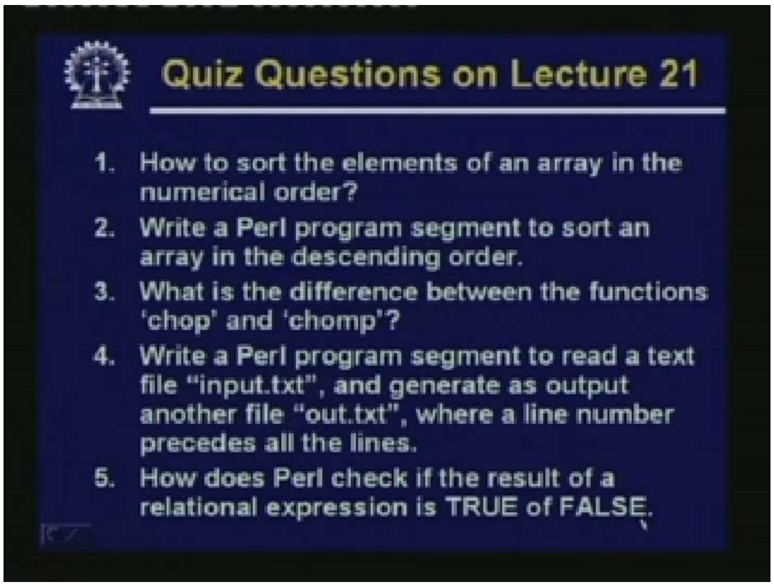
```
@xyz = reverse @original;
```

11. How do you print the elements of an array?

```
print "@arrayname";
```

How do you concatenate two arrays and form a third array?
Simple. Form a list with these two arrays and assign it to a third array.
How do you reverse an array?
Just give reverse of an array and assign it to something else.
How do you print the elements of an array?
Print within double quote at arrayname. So here also of you give within double quote then this interpolation will be carried out and the contents of the array will be displayed. The elements of the array will be displayed. Now some questions from today's lecture.

(Refer Slide Time: 52:41)

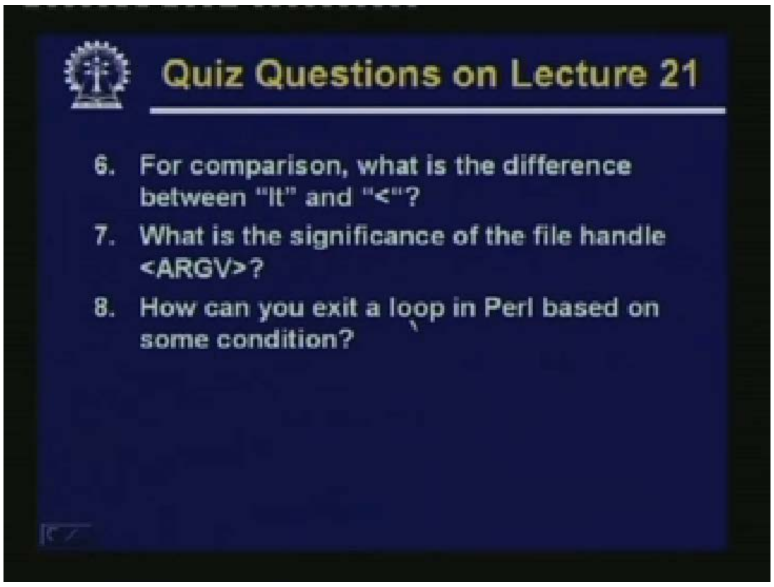


Quiz Questions on Lecture 21

1. How to sort the elements of an array in the numerical order?
2. Write a Perl program segment to sort an array in the descending order.
3. What is the difference between the functions 'chop' and 'chomp'?
4. Write a Perl program segment to read a text file "input.txt", and generate as output another file "out.txt", where a line number precedes all the lines.
5. How does Perl check if the result of a relational expression is TRUE of FALSE.

How to sort the elements of an array in the numerical order?
Write a Perl program segment to sort an array in the descending order?
What is the difference between the functions chop and chomp?
Write a Perl program segment to read a txt file input.text and generate as output another file out.txt where a line number precedes all the lines?
How does Perl check if the results of a relational expressions is TRUE or FALSE?

(Refer Slide Time: 53:15)



Quiz Questions on Lecture 21

6. For comparison, what is the difference between "lt" and "<"?
7. What is the significance of the file handle <ARGV>?
8. How can you exit a loop in Perl based on some condition?

For comparison, what is the difference between lt and this less than symbol?
What is the significance of the file handle ARGV within angular brackets?
How can you exit a loop in Perl based on some condition?
So these are the questions from today's lecture. So with this we come to the end of today's discussion. Thank you.

(Refer Slide Time: 53:48)

Preview of next lecture

Preview of next lecture.

(Refer Slide Time: 53:49)

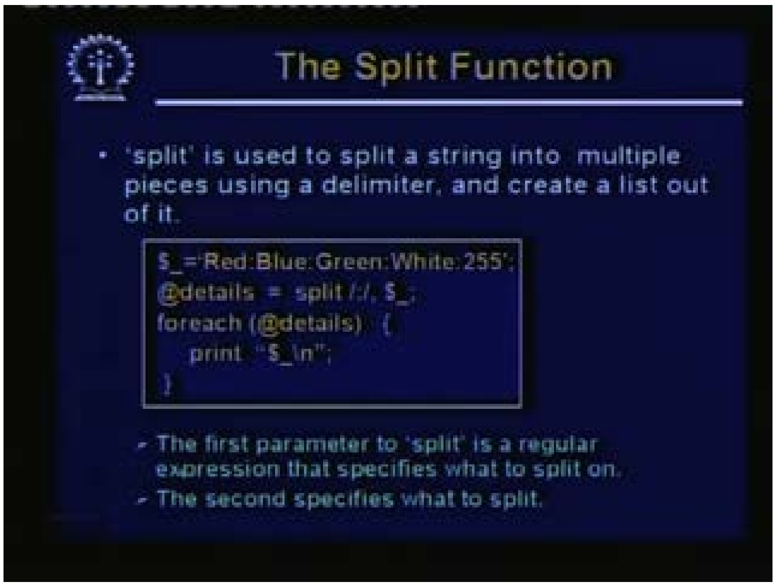
Lecture No. # 23

PERL - Part III

PERL – Part III

In this lecture we continue with the discussion on the Perl language. Now if you recall, we had already discussed the concept of variables expressions, how the expressions are evaluated in Perl. We talked about arrays and we talked about a few other string manipulation features that are available in Perl. Now in the present lecture we would first start looking at a couple of very interesting and important string functions followed by something which is considered to be one of the most powerful features of Perl namely, regular expressions. So we start by looking at some of very interesting string functions.

(Refer Slide Time: 54:45)



The Split Function

- 'split' is used to split a string into multiple pieces using a delimiter, and create a list out of it.

```
$_='Red:Blue:Green:White:255';
@details = split /:/, $_;
foreach (@details) {
    print "$_\n";
}
```

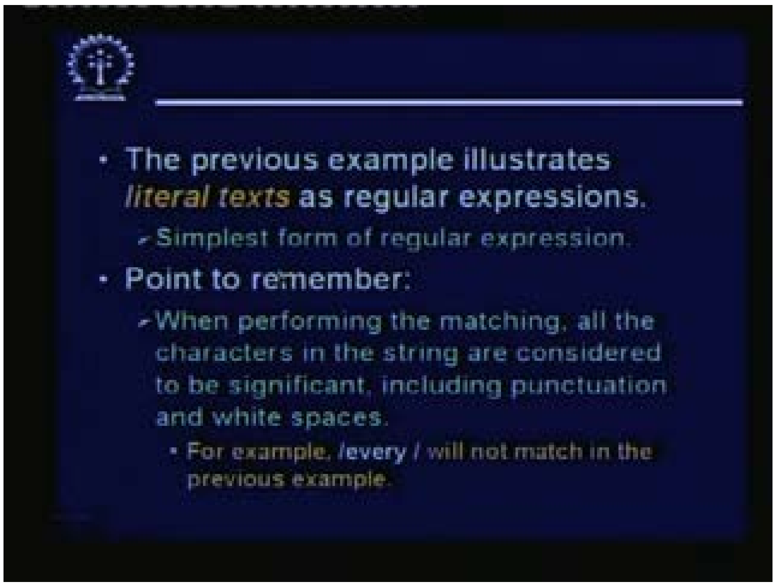
- The first parameter to 'split' is a regular expression that specifies what to split on.
- The second specifies what to split.

Well the first function is called Split. Split as the name implies is used to split a given string in to multiple pieces using a specified delimiter. What this means is that giving a string there are certain delimiters. Some defined delimiters; it can a colon it can be a space. They this particular delimiter is existing in several different points in the string and the whole string will be cut at the locations of the delimiters and the pieces that you get after the cutting, it is formed as a list. So the result of a split is a list of elements. So split breaks up a string in to multiple pieces and creates a list. Let us take a small illustrative example. Suppose I have a list like this. Red:Blue:Green:White:255 separated by some character colon, the whole thing is a list. Now you look at this assignment we have assigned this string to a variable dollar underscore. Well we could have used any conventional variable names. But we have chosen to use the dollar underscore character.

Now dollar underscore has a particular significance dollar underscore is some kind of default variable in Perl. See suppose you assign it to a variable say dollar abc, then whenever you want to refer to the string you should mention the name of the string dollar abc. But this dollar underscores which is considered to be default. If you use it in a string in many operations you will see that it is not required to specify the name of the string. By default it will considered or assumed that the string is (….) can understand this kind of facility gives Perl its power this is an example of regular expression see within this forward slashes I am writing every, this is a a simple example of a regular expression and this equal to tilde. This is an operator which means you search.

You are searching if the string specified in right hand side is present in the string on the left hand side. You can see this is extremely easy to use. As I had mentioned the text between the forward slashes this defines the regular expression and here we are using equal to tilde to check if it is present. This one now as an alternative we can use exclamation sign tilde which means the negation of the condition. This means that the pattern is not present in the string. So in place of equal to tilde if I write this exclamation tilde, this would mean that if this string every is not present in dollar underscore then print something. So whether you want the truth value of the condition checked in the if statement to be whether the string is present or not present, you can use either equal to tilde or exclamation tilde.

(Refer Slide Time: 58:27)



- The previous example illustrates *literal texts* as regular expressions.
  - Simplest form of regular expression.
- Point to remember:
  - When performing the matching, all the characters in the string are considered to be significant, including punctuation and white spaces.
    - For example, **/every /** will not match in the previous example.

So the example that we have seen that is the example of literal text. Literal text means within forward slashes we are specifying simple text to be searched for. This is the simplest possible form of regular expression. Some text enclosed in forward slashes you give and that you can search whether it is present or not. There is something to remember here that when you are performing the match all the characters in the string are considered to be significant.