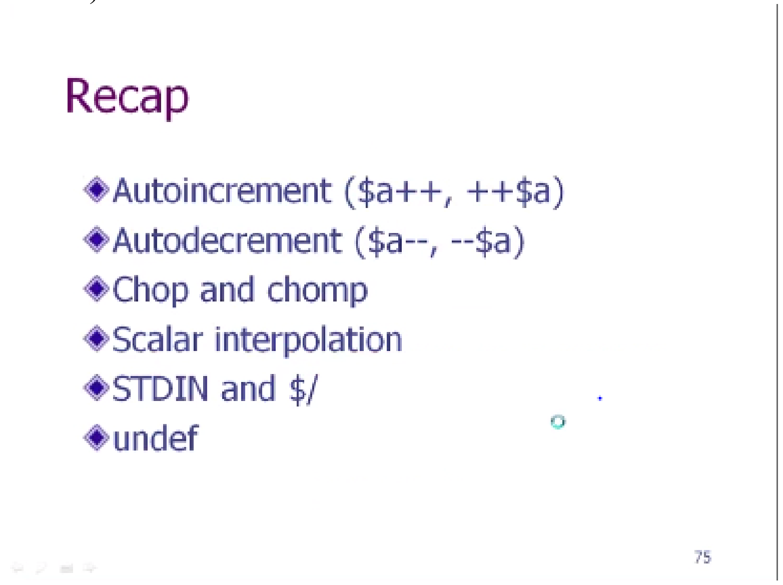
SEER AKADEMI

LINUX PROGRAMMIMG AND SCRIPTING PERL 4

Okay again welcome to the next lecture of the Linux programming and scripting today we are going to continue our discussion on Perl , we have seen some introduction to Perl we introduced some data types the vent into inter- analysis of the scalar data type and a few control structures that we talked about so let us just do a recap of what we discussed last time we also talked about some operators we already covered the operator persistence.
I hope you remember the presents table because that will be used in all the lectures going forward, so just keep in mind and even.
(Refer Slide Time: 01:01)



Recap

- Autoincrement ($a++, ++$a)
- Autodecrement ($a--, --$a)
- Chop and chomp
- Scalar interpolation
- STDIN and $/
- undef

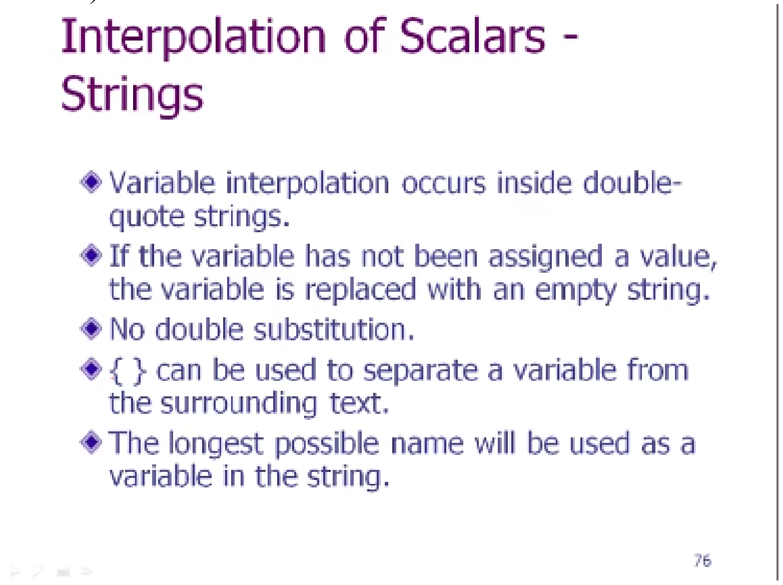
75

Let us talk about that so in the last lecture we covered the auto increment and decrement commands which are these + signs the table + signs and we also like saw some differences between more to put the + first or to attend, we also saw the auto decrement which is actually the mythical commands and again we saw like how we determine Perl and tou and how various things will be interpretation we also talked about the chop and the chomp commands the chomp command is the command that you is used to remove the newline character from the word.

So essentially we saw that and then we took several examples as to how to use the chomp function which is pretty much like. I mean used everywhere today and I am also know to be command read any line from a file you want to chomp and get rid of the newline characters at the end then we talked about some of the scale of interpolation basically like one scalar variable how related within the command within a run statement inside Perl.

We will talk about talk more details about this form the next slide we also talked about the STDIN and what how that functions if we if you look at this $ slash as a delimiter basically and then we can use it to read multiple lines from file or multiple lines from a standard in input and then finally we also saw this on that form essentially, so I am going to actually take some more details about these three like I mean just what we talked about last time in case the last time what we pay what would be incentive.

(Refer Slide Time: 03:06)



Interpolation of Scalars - Strings

- Variable interpolation occurs inside double-quote strings.
- If the variable has not been assigned a value, the variable is replaced with an empty string.
- No double substitution.
- { } can be used to separate a variable from the surrounding text.
- The longest possible name will be used as a variable in the string.

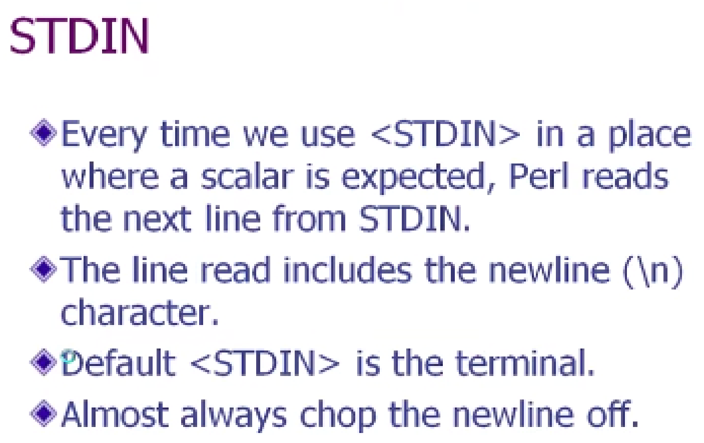
76

So let us look at the interpolation of the scalar variables mainly we talked about the rules governing this the variable interpolation occurs inside a double coated string essentially so this is one of the differences that we saw in the previous lecture there if the singer the difference between a single quote on the public good and if the variable has not been assigned the value then the variable is replaced with an empty string.

So that is another key differentiation potential a way to distinguish, so the program will not generate an error instead it will basically is the unassigned the value with the with an empty string one other thing is like there is no double substitution meaning an odd substitute once and then have that as an another variable, so that in substitute again, so it is only one time use essentially.

So that is another thing then in case like, I mean you are you have letters basically that are continuous and then part of it denote a variable then you can use these the brackets essentially to separate the variable from the surrounding text this will be more apparent when I talk about talk

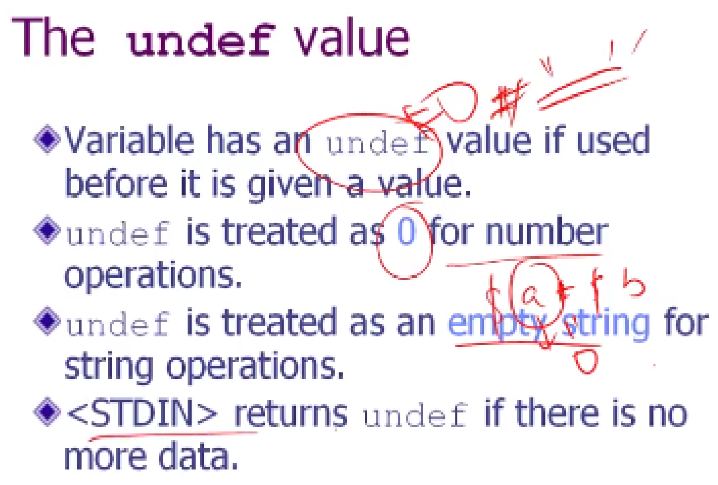
about this next rule which is right here which is the longest possible name will be used as a variable in a string.
So if you say like name Anand, I hear and then you put a $ this whole thing is treated as just one variable at this point say like I mean you want actually variable is $ anand and then there is higher is just some set of text basically to denote this you may want to put the parentheses here.
So that it knows that okay $ anand variable and I a resolved the surrounding text so you can now see like, I mean the how this rule will be applied basically why these two roads crews go hand-in-hand.
(Refer Slide Time: 05:38)



STDIN

- Every time we use <STDIN> in a place where a scalar is expected, Perl reads the next line from STDIN.
- The line read includes the newline (\n) character.
- Default <STDIN> is the terminal.
- Almost always chop the newline off.

So the other thing that we talked about was the STDIN every time we you we use standard in place of a scale of expected the Perl reads the next line from standard in, so essentially we can Ask you to read this any amount of text from the terminal by just using the standard in one thing to note here is the line ,we did includes the newline character ,so if you want to remove the newline character you will do standard in and then you immediately do a chop.
Instead of down to get rid of the newline character and then the default STDIN even though like I mean you can see that which is actually a file handle this is a terminal essentially, so you read in from the terminal and then you can do things with it and then as I mentioned basically almost always chop the new line of after we read in the STDIN, so this is the key things about this again we also saw some examples. I am not going to quote the examples this is still talking about the recap and then finally.
(Refer Slide Time: 07:03)



The undef value

- Variable has an undef value if used before it is given a value.
- undef is treated as 0 for number operations.
- undef is treated as an empty string for string operations.
- <STDIN> returns undef if there is no more data.

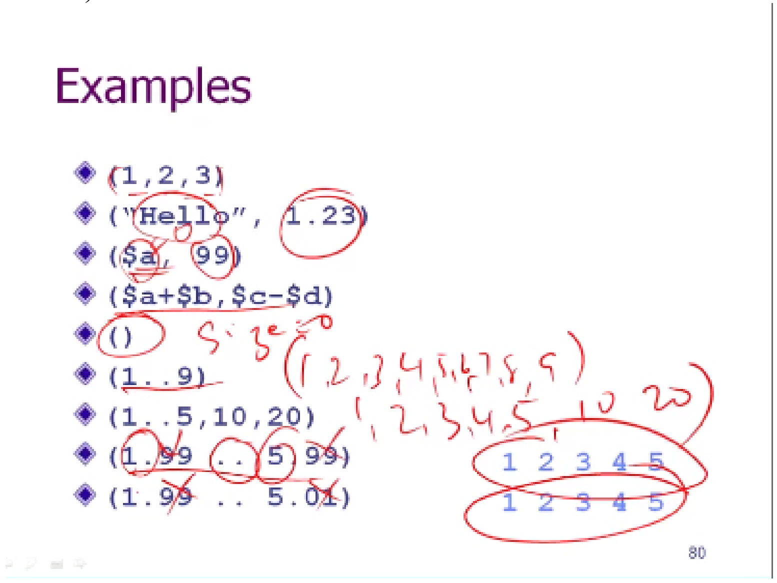
The undef value buttons are essentially the unless is actually you can think of it as a value and that value is used the value if used before it is given wrong so before you give it value is what is the undef essentially it is not equal to 0 or it is not equal to a null string so I think I mean that is one thing that you want to understand only like for the number of these things the undef is treated as 0.

So if you have any kind of that $ a + $ B where the $ A is not defined then this will equal to 0 or this operation extension, so otherwise like I mean it is like not the zero but when you perform an operation did that are you the better variable then it is treated as 0 and it is also treated as an empty string or string of this rule does not hold good then you perform an operation and then STDIN returns the undef if there are there is no more data.

So now we will start today's topic ,I think like and so this is fairly clear we will encounter this under in need programs in the coming lectures .I just want to do again understand and keep in mind how to use so today we will start our topic with the discussions on array and list data mostly we will be focusing on array today and lists are basically with even more talk about it in the next one again.

What is an array is an ordered list of a scalar data in Perl one good thing is like there is no declaring need to define an array and the size can be 0 to any number mostly like this number is limited by the amount of memory that is that is allowed in the machine , and we can easily increase the size of an array by just adding new elements to it so we will see like how we can be array and how we will do additional arrays.

So that array literal is a list of comma separated values in post in parentheses, so we will see like how to deform in very neutral and then there are some special operator one this one this is the conquer range operator it has two meanings depending on the context and we will see that.
(Refer Slide Time: 10:10)

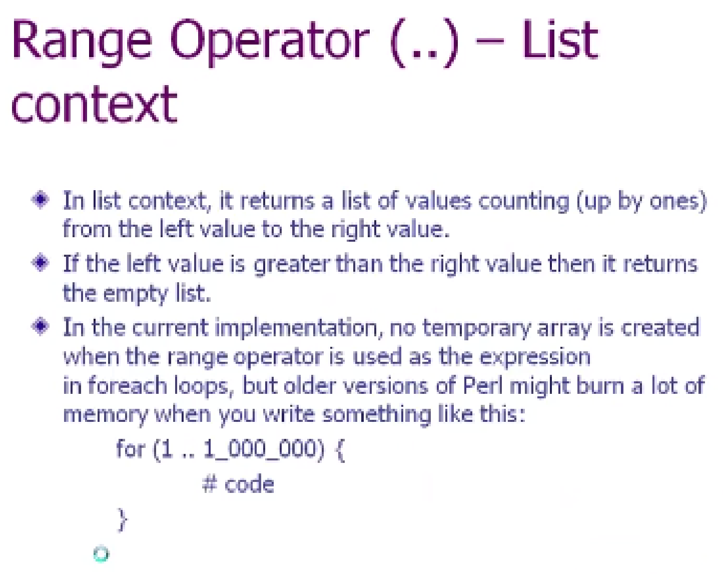


Examples

- (1,2,3)
- ("Hello", 1.23)
- ($a, 99)
- ($a+$b,$c-$d)
- ()
- (1..9)
- (1..5,10,20)
- (1.99 .. 5.99)
- (1.99 .. 5.01)

size =0
(1,2,3,4,5,6,7,8,9)
1,2,3,4,5,10 20
1 2 3 4 5
1 2 3 4 5
80

So let us first look at some of the examples of amore so this is an array literal basically for book two parentheses and then the values are in close even though canvases and they are comma separated this is another array which has a string as one of the elements and then there Is a numbers and again this is another area if you define this basically then the $ a becomes the first element and 99 second otherwise the this becomes an undef or zero.
Then any operations performed this one essentially like I mean this an arithmetic operator so with an array and then this is an empty array of array of size =0 and then we have the range operators one through nine so actually here it will be like 1 2 3 4 7 8 9 that is your array and in this one it is 1 2 3 4 5 10 and 20 possible ,so now what will this array be essentially so again here it is a we will talk about the rules as to how to do how.
Array is defined but essentially like I mean it will be like this 1 2 3 4 5 and even for this one is 12345 because it only takes the integer from the first and increments one at a time until it reaches the final number, so these will be ignored this will be more this will be more or will be.
(Refer Slide Time: 12:12)



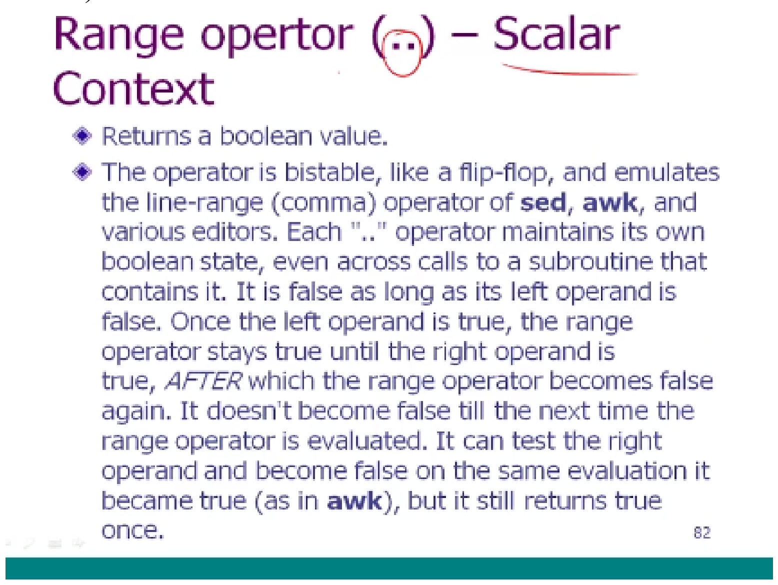
Range Operator (..) – List context

- In list context, it returns a list of values counting (up by ones) from the left value to the right value.
- If the left value is greater than the right value then it returns the empty list.
- In the current implementation, no temporary array is created when the range operator is used as the expression in foreach loops, but older versions of Perl might burn a lot of memory when you write something like this:

```
for (1 .. 1_000_000) {
        # code
}
```

So let us look at the two contexts that I mentioned for the range operator ,so the first one is the list context which is pretty much what is defined in here all these four examples define the list context of the three in the list context it returns the list of values counting up by ones from the left value to the right one so the left value is greater than the right value then it returns an empty list so that is another key thing to note so but in the modern implementation so like I mean if you look at some of the Perl book the older Perl book you will see like this one caveat written in that there if you use this kind of a number 1..1 underscore 10001000 then it can burn A lot of memory because it creates an ugly for you a temporary area for you but this temporary array is no longer created in the musician world So this statement even though it is not very good news it is a fake statement it is not an it would not open issues so in this statement.

(Refer Slide Time: 13:44)

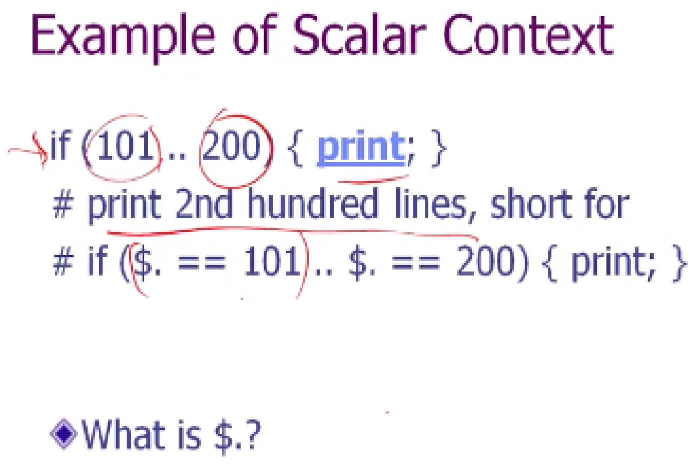


Range opertor (..) – Scalar Context

- Returns a boolean value.
- The operator is bistable, like a flip-flop, and emulates the line-range (comma) operator of **sed**, **awk**, and various editors. Each ".." operator maintains its own boolean state, even across calls to a subroutine that contains it. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, *AFTER* which the range operator becomes false again. It doesn't become false till the next time the range operator is evaluated. It can test the right operand and become false on the same evaluation it became true (as in **awk**), but it still returns true once.

82

The second context the range operator to mention is the scalar context in scalar context the operator dot ,dot actually returns a Boolean value so it is like a flip-flop it emulates the long-range operator said arc and other editors this is the line range operator is also known as comma operator so each of the dot, dot operator maintains its own Boolean States even across calls to subroutines that contains it so the way it works is it is false as long as the left wrangles falls once the left operand.
Becomes true then the range operator stays true until the right operand is true after which the range operator becomes false again it does not become false till the next time the range operator is evaluated so you can test the right app operator operand and become false on the same evaluation and it becomes true but it still retains the true at least once so these are all like. I mean big set of rules let us say like. I am in an example because that pretty much gives us how this command works.
(Refer Slide Time: 15:15)



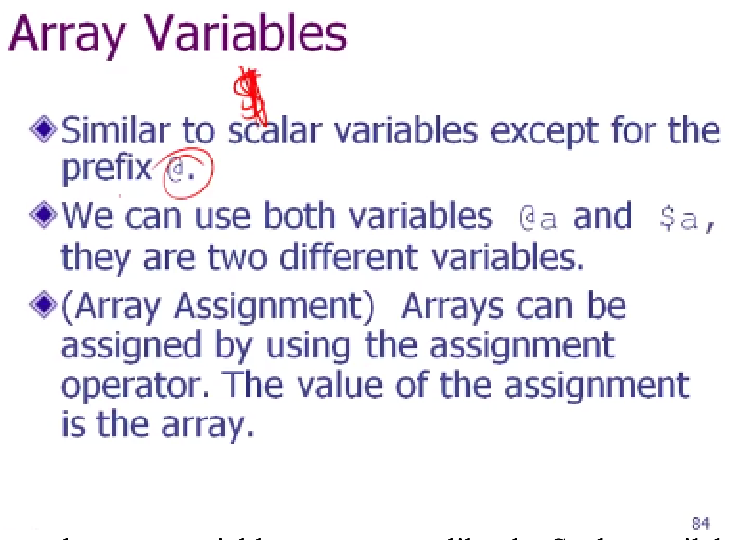
Example of Scalar Context

```
if (101.. 200) { print; }
  # print 2nd hundred lines, short for
  # if ($. == 101).. $. == 200) { print; }
```

- What is $.?

So here is a simple scalar context essentially there we use this number here 101 dot, dot 200 print so the way to interpret this command is essentially it prints the 200 to that the second hundred lines essentially, so basically it prints from line number 101 to 200 but it is the short for these conditions they see a like $ dot equal to equal to 1 0 1 & dot equal to equal to 200 Print so the question is like I mean so this is another $ variable we saw like two of them before and now the third one which is the $ dot operator so the $ dot essentially it is the it is the shorthand for input line number .
So essentially like I mean what we are saying here is this 101 is actually the nine mumbles so the line number is equal to 101 all the way to line number equal to 200 prints. This so here actually

let me discuss want quality operator but there are other ways of using the dot operator as well so in the scale of context and so you can test multiple conditions and then come up with these answers for that.
(Refer Slide Time: 17:29)

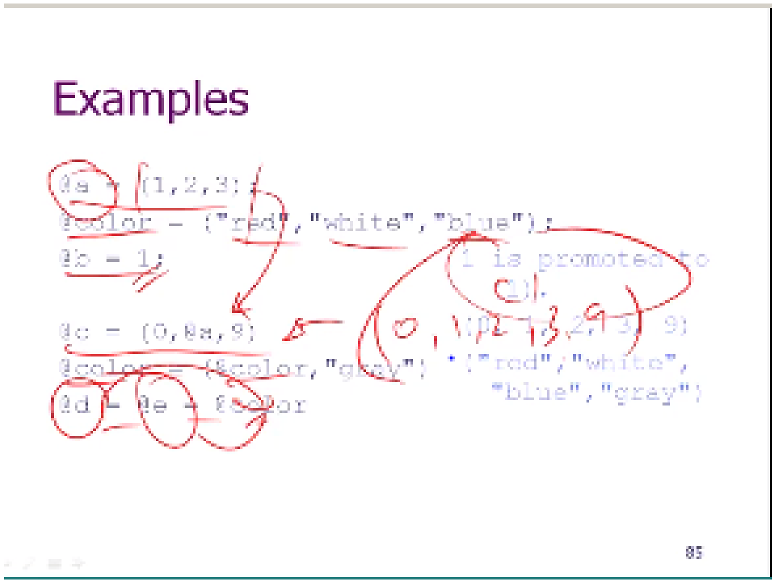


Array Variables

- Similar to scalar variables except for the prefix @.
- We can use both variables @a and $a, they are two different variables.
- (Array Assignment) Arrays can be assigned by using the assignment operator. The value of the assignment is the array.

84

So now let us talk about the array variables ,so we saw like the Scalar available exactly they start with the $ the array variables are they start with the ampersand or at so since the they have different prefix compared to scalar variable we can have the same name but you can contribute variables or example the ampersand a and $ a are actually like two different variables.
So you can use them interchangeably or you can I mean you can each will have its own value so it is not dependent on one valuable okay, so that is one thing the array assignment essentially where arrays can be assigned by using the consent of operator the value of the assignment is the entire array.
(Refer Slide Time: 18:43)

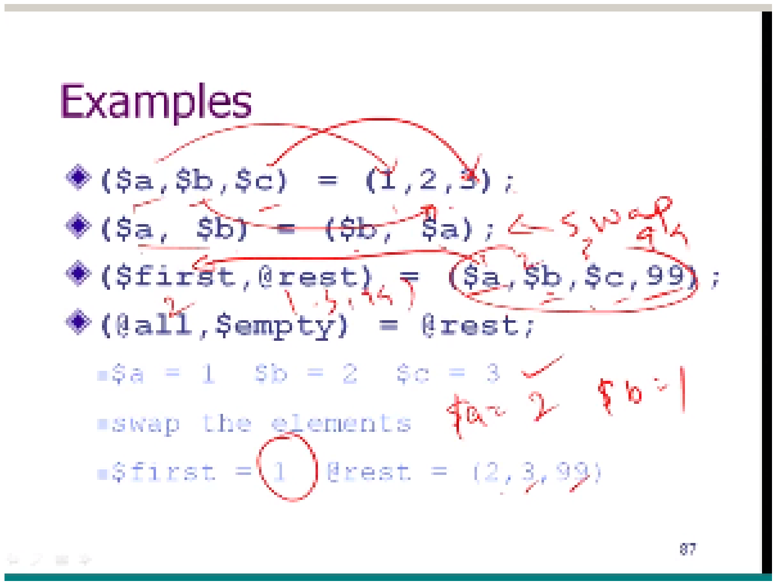


Examples

```
@a = (1,2,3);
@color = ("red","white","blue");
@b = 1;                       1 is promoted to
                                 (1).
@c = (0,@a,9)                 (0,1,2,3,9)
@color = (&color,"grey")  ("red","white",
                             "blue","grey")
@d = @e = &color
```

85

So how do we use this so here are some examples first of all declaration of an array we know that this is the way to declare an array and then we assign a variable which is the ampersand a and here this is the string array containing colors the names of colors so at color is red white and blue and at B equal to 1 this is also a interesting array containing this one element here is another one C is basically 0 at a and 9.
So this can be you can replace this here so that it becomes like 0 1 2 3 and then the same thing the array is actually updated towards the end to the $ and you can see that actually infuse both places and this is the multi assignment case where actually like both $.
I have been at E and that B are assigned stay mass color so here so one more clarification here which is the one I mentioned basically it is a single imminent very so it is basically it will be promoted to like the parent accession so that it can be the UM it will be an array that so here the same thing like 0 1 2 3 9 and then here again the gray up to the end and then this one so both $ deontology are asking the same color as the variable.
(Refer Slide Time:20:35)



Examples

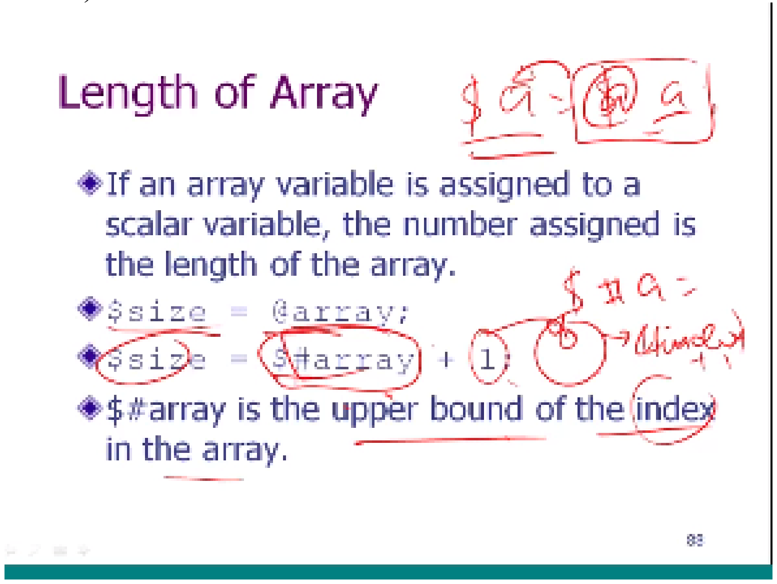
- ($a,$b,$c) = (1,2,3);
- ($a, $b) = ($b, $a); <- swap
- ($first,@rest) = ($a,$b,$c,99);
- (@all,$empty) = @rest;
  - $a = 1 $b = 2 $c = 3
  - swap the elements
  - $first = 1 @rest = (2,3,99)

So the array literal essentially like we saw this briefly essentially like, I mean the digital contains only the only variable references it does not have any expression so constants the early array literal can be just used as a wave it can be used on the left hand side of the assignment so let us see some identical, so here $ a $ B $ C equals one two three so what this mean to form essentially like the a gets one B gets two and seamlessly.

So let us be fine and then the second command is essentially like $ a $ being equal to $ B $ a even though you can see that actually they insane this is what is called a swap so we just swap the elements so more the problem a becomes two and $ B becomes one so now let us look at that the first $ first and the $ breath is $ a $ B CC and multiple so the first is whatever will be the first and that gets a time to the $ force so the first is in this array it is a one.
So it gets one and then the rest is $ B $ C and 99 it is two three and so here again like I mean we are not doing the swap this is before the swap so that is why I like I mean it is still this values are 1 2 3 and 99 if we did upload after this statement then this will be more to will be assigned to first and then one 399.
When we are back to second well the best now you can also have an array followed by a scalar variable inside the assignment so in this case like a $ all involve empty essentially the $ all contains the $ at the ampersand where so or sorry the ampersand all contain all of the elements from ampersand rest which is in this case it will be the two three and ninety nine and then the empty is essentially like a minute it returns this under so it has nothing in it and it just returns and SS for the next level so I hope like again this these examples illustrate all the array structure are a

little can be used so this will begin use in many programs. In used to collect information and then also like disseminate the information to the other sections of the program.

(Refer Slide Time: 24:01)



Length of Array

- If an array variable is assigned to a scalar variable, the number assigned is the length of the array.
- `$size = @array;`
- `$size = $#array + 1;`
- $#array is the upper bound of the index in the array.

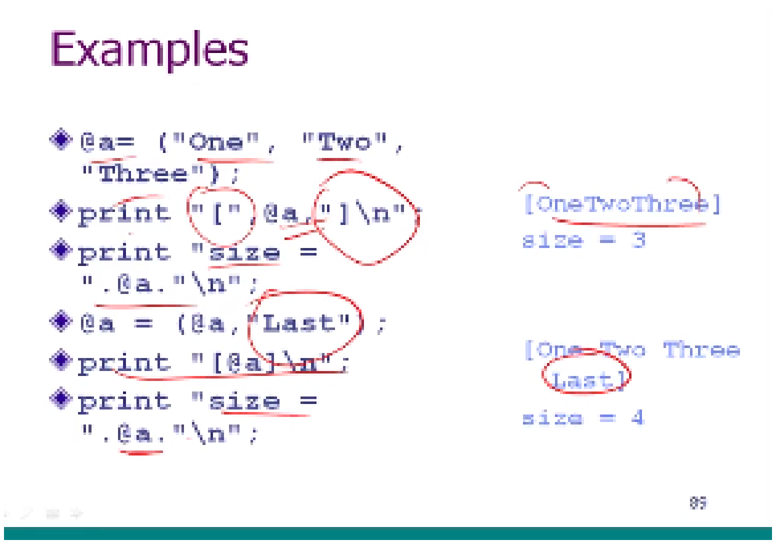
$a = @a
$#a =
→ last index

So now another question is how do you estimate how do you determine the length of an arc the length of an array is determined by a scalar variable and if the array variable is assigned to a scalar variable the number assigned is the length of that array , so what does that mean so you can just stay basically okay $ a equal to dot all are at a so in this statement essentially like a bar over the number of elements.

In at a is assigned to $ it actually to just slightly bit more complicated than that essentially so this one now it takes the number the thing so there is also a special variable this $ hash array this is essentially like stores the upper bomb of the index on in the array what that means is the bound the index of the last element for theory is stored, in this variable so for example in this case it will be like $ as a will hold the index of the last element so typically the array goes from zero to the index of not element now index.

So usually we want to add one to the last index to get the number of arrays because it starts with zero so long so that is the reason why like. I am going to assign the ball size I mean the order array the whole array to the $ side the value of size is actually so $ hash array + 1 so this one denotes this one to represent 0 and then this will be the index of the last element in the array.

(Refer Slide Time: 26:46)


Examples

- `@a= ("One", "Two", "Three");`
- `print "[",@a,"]\n";`
- `print "size = ".@a."\n";`
- `@a = (@a,"Last");`
- `print "[@a]\n";`
- `print "size = ".@a."\n";`

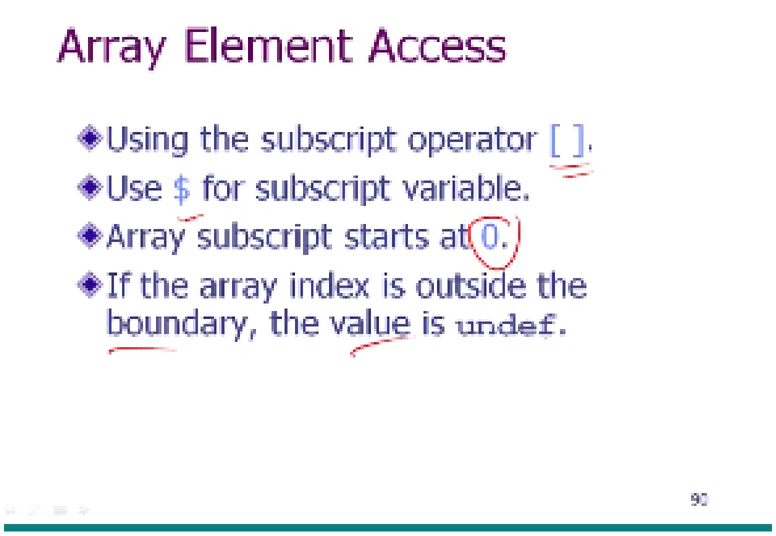
```
[OneTwoThree]
size = 3

[One Two Three
Last]
size = 4
```

89

So some examples here so here basically the ampersand is one two and three so now when we print just a hole array basically prints one two three then prints this square brackets awesome because they are marked as now we want to print the size, and let me take a guess, so the type Is actually three and that is what we got and then the next one we are doing essentially you are just adding additional one more element Which is last to the array and then we are printing out the that simply with a square bracket beginning in the end so now we get this and then finally we have this size equal to ampersand at which it should now denote size as four because we have added one more element about it.

(Refer Slide Time: 28:07)

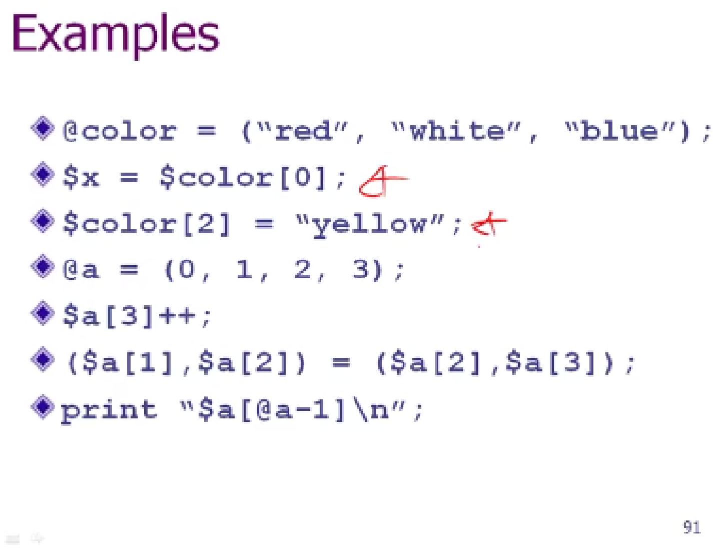

Array Element Access

- Using the subscript operator [ ].
- Use $ for subscript variable.
- Array subscript starts at 0.
- If the array index is outside the boundary, the value is `undef`.

90

So now the element array access itself we kind of figured out that you knows we can use it with the subscript operator. Which is the square bracket and then use the $ or the substrate variable the

array subscripts typically starts at zero we already saw that in books like that when we asked for size is three but essentially it is a denoted as this one two and three so essentially like, I mean it is that $ pound or $ - array + one.
So and if when area index is outside the boundary value is formed opening so this is one of the key concepts of why we want to understand what understands form in the very beginning.
(Refer Slide Time: 29:03)



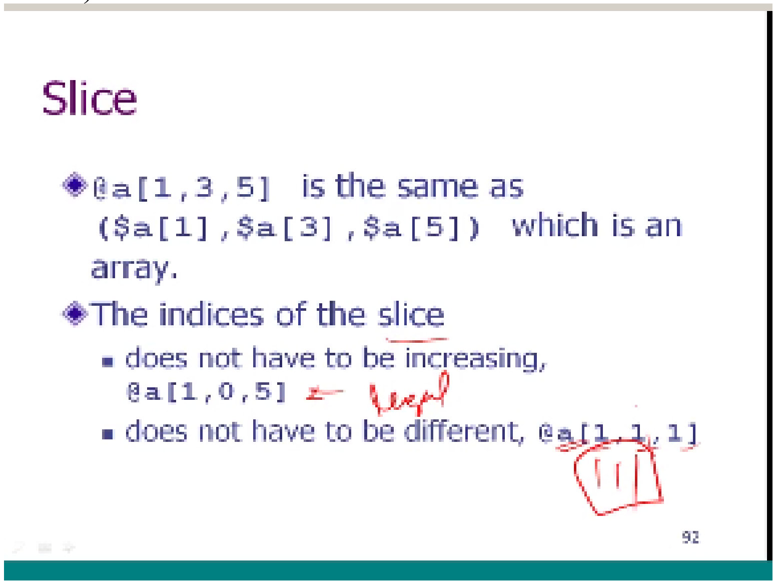
Examples

```
◆@color = ("red", "white", "blue");
◆$x = $color[0];
◆$color[2] = "yellow";
◆@a = (0, 1, 2, 3);
◆$a[3]++;
◆($a[1],$a[2]) = ($a[2],$a[3]);
◆print "$a[@a-1]\n";
```

91

Now there are some examples so the & color is red white and blue and $ X is $ color zero so you can say what this has and now the $ color two is changed to yellow so now and then here there are some more some more ways of accessing the array ,so here actually like I mean so one to denote x $ x=red actually yeah and then here color 2 is yellow that means that color 2 is the last element of the array.
Which is blue and then that is replaced by you know and here again like so this is more like a string or someone s string or string array now let us look at these examples here the array is declared as a numeric level which is 0 1 2 3 and then we increment the third element by 1 so that means that the 3 is becoming 4, and then we write out the so here we are doing like a 1 a 2 is same as e 2 a 3.
That means that we are swapping it so we actually have the array s 10024 and then now we have swapping the first in the second with the second and the third, so this means this will become like 11 or 122 4 and then if we print a $ a % a - 1 then it prints number 4 which is the last that is because denote the current value of the current array essentially and then the current only arrays line number and then more ampersand a - ampersand a services form the number of elements of that array.

Which is 4 and then B -- 1 out of that so this becomes $ a 3 which is essentially it goes and finds out the last element and then prints out so the answer will be 4.
(Refer Slide Time: 32:10)

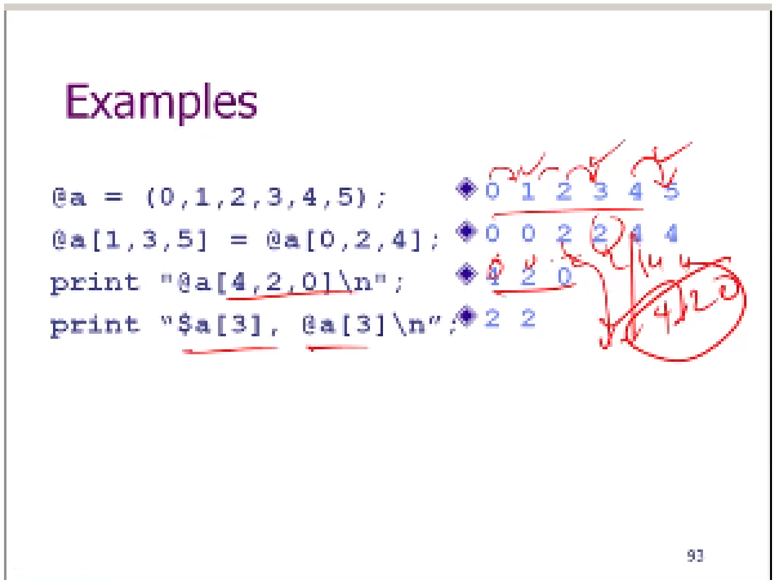


Slice

- `@a[1,3,5]` is the same as `($a[1],$a[3],$a[5])` which is an array.
- The indices of the slice
  - does not have to be increasing, `@a[1,0,5]` ← legal
  - does not have to be different, `@a[1,1,1]`

92

So now we introduce another concept called slice, slice is a shorthand representation to denote multiple elements of an array so in this small example ampersand a 1/3 and five is essentially to the same $ one $ eight three and validate five the indices of the slice does not have to be increasing for example ampersand a 1:05 is perfectly a legal one and also it may not be different it can be like this 111.
Which is essentially says that of the first element of all the from my file I want or my array I need the same value to be set 3 times so if you look at your essentially say 0 so people come in thanks so actually it is better so this is the 0th element this is the posting so it will start building just one moment okay.
(Refer Slide Time: 33:42)

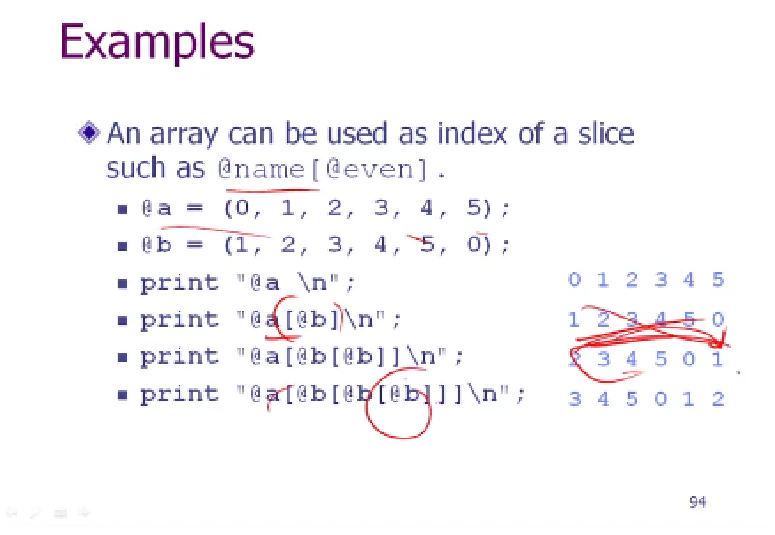


Examples

```
@a = (0,1,2,3,4,5);         ◆ 0 1 2 3 4 5
@a[1,3,5] = @a[0,2,4];      ◆ 0 0 2 2 4 4
print "@a[4,2,0]\n";        ◆ 4 2 0
print "$a[3], @a[3]\n";     ◆ 2 2
```

93

So here are some more examples maybe so $ is 0 1 2 3 4 5 that is here actual array and then we replace 1 3 & 5 which is 1 3 & 5 we assign the two .I mean actually lecture is one shuttle so one three and five will be replaced by zero two and four so essentially like I mean until it becomes 0 u 2 to 44 as the main values of this area and then when she print out for 2 and 0 that is goes there 0 1 2 3 4 and so for 0 1 2 and then it spits out two for two wheel.
So that is what denoted here and then finally we asked it to print $ a three and add it so can you see the difference essentially and then again see how that will be printed out in this case both of them are the same four turns out to and $2 a three is payment here and $. I mean ampersand each awful thing.
(Refer Slide Time: 35:30)

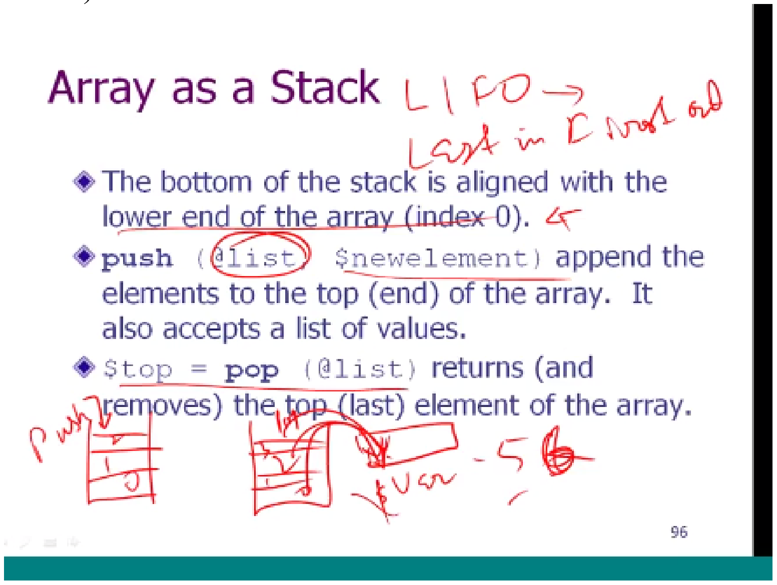


Examples

- An array can be used as index of a slice such as @name[@even].
  - @a = (0, 1, 2, 3, 4, 5);
  - @b = (1, 2, 3, 4, 5, 0);
  - print "@a \n";   0 1 2 3 4 5
  - print "@a[@b]\n";   1 2 3 4 5 0
  - print "@a[@b[@b]]\n";   2 3 4 5 0 1
  - print "@a[@b[@b[@b]]]\n";   3 4 5 0 1 2

94

So one other key thing to note about the slices and array it serves can be used as an index for slice. So here example is like $ name and it has the index of $ even which is also another array so here is another example $ a is defined as 0 1 2 3 4 5 and B is actually like this two shift to the left which is 1 2 3 4 5 and then is zero so when we print the $ ampersand a we get zero one two three four five and then we say print a and then in parentheses well B.
So what will you get pretty much the same essentially like so it could be used the BS mix focusing same results now what if like you use B once again and then try to print that essentially , so now it is almost as if like I mean it is a shifter so shifting to the left so or Essentially to the right, so the to actually goes I mean actually the one goes to the two at the end and then you get 2 3 4 5 0 1 And then similarly recommend you can actually include before sandy and can make it one more shift register.
(Refer Slide Time: 37:04)



Array as a Stack

- The bottom of the stack is aligned with the lower end of the array (index 0).
- push (@list, $newelement) append the elements to the top (end) of the array. It also accepts a list of values.
- $top = pop (@list) returns (and removes) the top (last) element of the array.

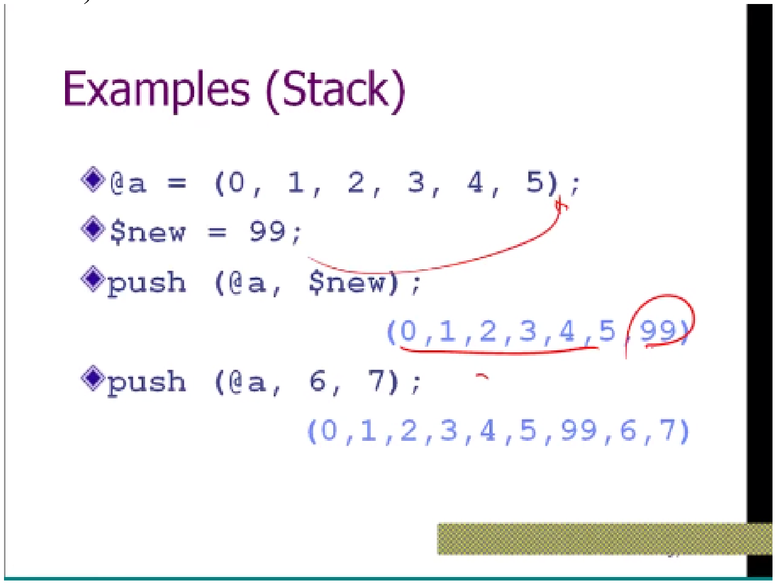
96

LIFO →
Last in First out
push
$var
- 5 6

So now the next topic is how to build array as a stack essentially, so a stack is you can think of like a like a jar of say biscuits essentially so the elements that you insert first goes to all the way to the bottom ,and then basically like when you are popping essentially like. I mean only pops from the top so all the elements in order to access the last element the first element will be pushed in you have the element or you have to access all the elements before pop out all of them and then only you can take the last one.
This particular kind of arrangement is also more of L I F oh I think this is quite familiar to many of you which is last in first up so how do we implement this kind of complex data structure so one thing to notice when we talk about stacks the stack is still aligned at the lower end of the

array as the index 0 this is one keeping Y V can easily access, essentially ligament and then if it is not aligned then the data can get corrupted.
So what we do is essentially like push a list with the new element so at a list is the variable that holds all the value ,so this is one way to do it is at list and then the new element use the push command this events the element at the end of the array so you have one two say five and then even your turn comes basically that gets at about four or six. So somebody has to pretty much like do memory management all the time to understand what is going on.

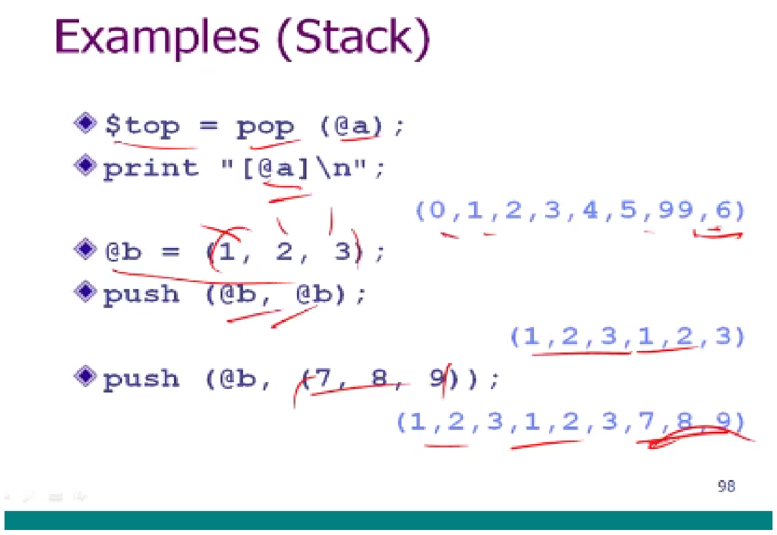
(Refer Slide Time: 39:41)



Examples (Stack)

```
◆@a = (0, 1, 2, 3, 4, 5);
◆$new = 99;
◆push (@a, $new);
                (0,1,2,3,4,5,99)
◆push (@a, 6, 7);
            (0,1,2,3,4,5,99,6,7)
```

The other main thing is the pop essentially the pop function functionality is shown here which is basically pop the list and then attain the top one to the $ top, so you think of it like this whistling so when you this is a bush this is 0 1 2 and then wasteland continues on for the pop you have this array zero one two three and then essentially the way to of the thing is basically one final variable and then this variable force element from the top.
So first you need to clear the third element before you can do the second element before you can do the first one so I think like I mean that is pretty much this topic now shall we go into the so before we go into it essentially let us do some examples again so here is an example, where the array is defined as 4 0 1 2 3 4 5 and then we have a new scalar $ which is equal to 99 and then we say basically push the $ new into the array a so what that means is essentially this is pushed towards the N.
So 0 1 2 3 4 5 8 99 at the end now if you push to with letters six and nine six and seven so now already this change to 99 so when we pushed six and seven in within six inches.
(Refer Slide Time: 42:01)



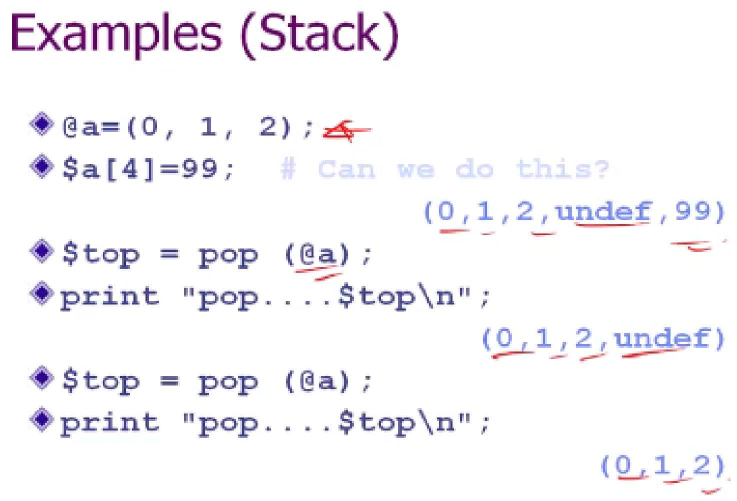
Examples (Stack)

```
◆$top = pop (@a);
◆print "[@a]\n";
                    (0,1,2,3,4,5,99,6)
◆@b = (1, 2, 3);
◆push (@b, @b);
                          (1,2,3,1,2,3)
◆push (@b, (7, 8, 9));
                    (1,2,3,1,2,3,7,8,9)
```

98

Now let us look at the continue with the same example and into the pop section we assigned on the top to pop and percent a so when we try to print ampersand a it gives like 0 1 2 3 4 5 99 and then 6 and then you define another array variable called the $ B then it is assigned to one two three and then we do a push $ B I mean ampersand B 2 amps and B that means that the concept means this at least two times So we get 1 2 3, 1 2 3, now if we push another array 789 that array will be like all these constructs are lost then there is basically it has a longer array of the 1 2 3 and 7 8 9 .

(Refer Slide Time: 43:05)

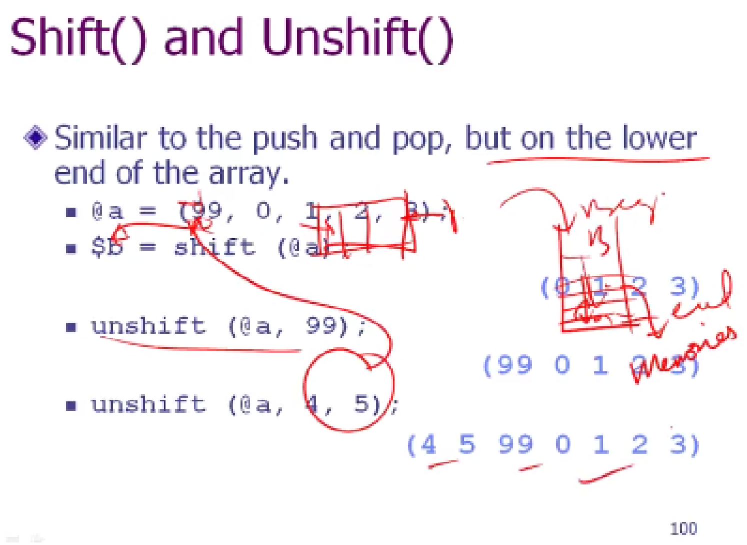


Examples (Stack)

```
◆@a=(0, 1, 2);
◆$a[4]=99;   # Can we do this?
                      (0,1,2,undef,99)
◆$top = pop (@a);
◆print "pop....$top\n";
                          (0,1,2,undef)
◆$top = pop (@a);
◆print "pop....$top\n";
                                 (0,1,2)
```

Let us see some examples , so here we have a declaration where ampersand a is 0 1 2 3 and then $ a we defined that for 99 and then essentially like I mean so if the result of this statement is 0 1

2 unless and then 99, so now the $ top is defined as we want to pop stuff from ampersand away and then , so that was successfully done and then we say basically okay, print the pop through the top and that prints out This information 1 2 0 1 2 then undefined and then top equal to pop and then pop stop, this will print out 0 1 2 okay.
(Refer Slide Time: 44:31)

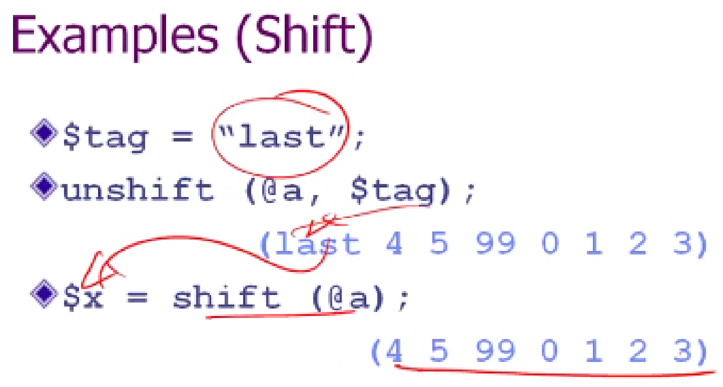


Shift() and Unshift()

- Similar to the push and pop, but on the lower end of the array.
  - `@a = (99, 0, 1, 2, 3);`
  - `$b = shift (@a);` (0 1 2 3)
  - `unshift (@a, 99);` (99 0 1 2 3)
  - `unshift (@a, 4, 5);` (4 5 99 0 1 2 3)

B
end
memories
100

So it will also like go to the lecture for shift and unshift, so the shift and unshift is very similar to the pop up we say basically the this is the beginning of the array basically so the beginning is the end essentially like. I mean now you have a partition here and you basically like memories they are going away especially that one and then more contacts ,so in this one essentially the so the way this will work will be basically that each element is pushed into the array.
So whereas in a shift actually so you can think of this as the so the push operation essentially like I mean pushes the existing elements down and pushes at the front so this is the beginning and this is the end so everything is pushed to the end so and then the pop is its back the same way that as a shift operation essentially it is on the lower end of the array essentially like.
I mean so the index of that will be the new box ascension we will see some examples here so the array is defined as 99 0 1 2 3 and then we shift the array and then the result is essentially 0 1 2 3 because now this 99 essentially goes into the beam so $ B becomes 99 and then the array Stays array has 0 1 2 3 and then when we say like unshift that particular element then it is back basically like it is cushioning to go array brought back And then we fit a lichen shift four and five so both four and five are added at the beginning of the hurry so it is four five nine, nine, nine zero one two three.
(Refer Slide Time: 47:08)

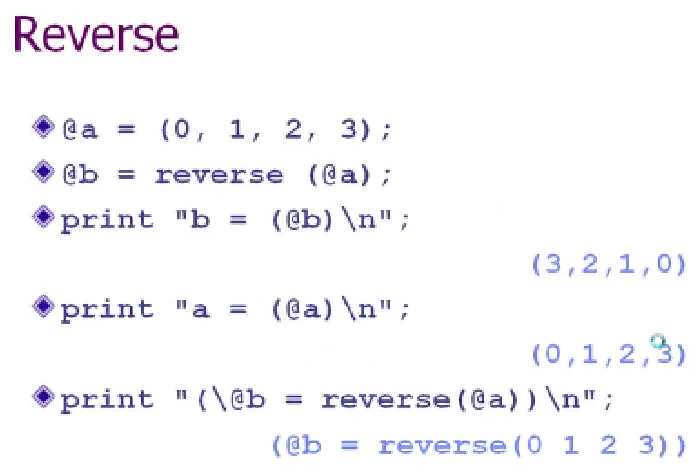


Examples (Shift)

```
◆$tag = "last";
◆unshift (@a, $tag);
            (last 4 5 99 0 1 2 3)
◆$x = shift (@a);
                  (4 5 99 0 1 2 3)
```

101

Some more examples, on the shift so if you say like unshaved $ tag the tag or the tag is defined as last and then that gets added and the as the first element of the array and then if you do an X $ X equal to shift and ampersand a then the last gets assigned to the shift I sent to the X and then the array becomes just for 4 5 99 0 1 2 3.
(Refer Slide Time: 47:47)



Reverse

```
◆@a = (0, 1, 2, 3);
◆@b = reverse (@a);
◆print "b = (@b)\n";
                           (3,2,1,0)
◆print "a = (@a)\n";
                           (0,1,2,3)
◆print "(\@b = reverse(@a))\n";
            (@b = reverse(0 1 2 3))
```

102

So I think, I am going to stop at this point we will talk about the reverse sort and those type of functions. In the next course and again we are still talking about the arrays themselves basically so arrays are future data structure and it is used everywhere and in everyday life, but we have like very little information about this functionality and that is one thing that we covered today oh this is quite useful so what we talked about.

I am just taking you back , so we started our discussion with an array he understood how to declare an array and also like we saw how operations can be performed on an array we understood how to get the length of the array and so that we can act upon it we can use do other things with normal then we also went through some work ways of creating an array um you creating an array and then how to use stacks.
And the shift registers or shifters this is so I think that is all for today ,we will actually continue from this point in the next lecture until then on the tanks thanks for listening thank you very much.