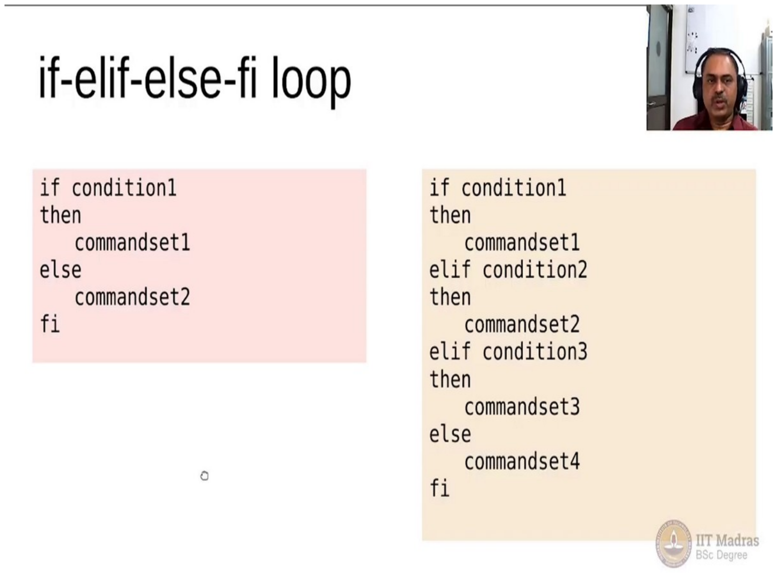
System Commands
Professor. Gandham Phanikumar
Metallurgical and Materials Engineering
Indian Institute of Technology, Madras
Bash scripts - Part 2B

(Refer Slide Time: 00:14)



if-elif-else-fi loop

```
if condition1
then
   commandset1
else
   commandset2
fi
```

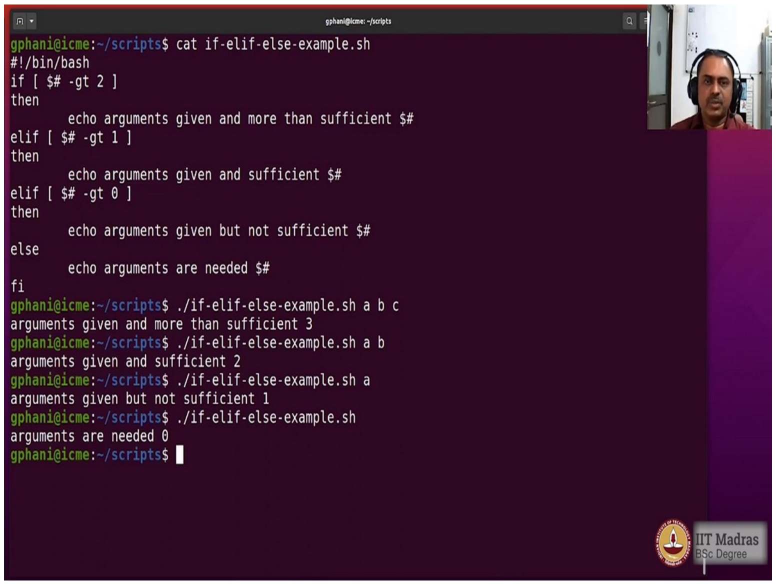
```
if condition1
then
   commandset1
elif condition2
then
   commandset2
elif condition3
then
   commandset3
else
   commandset4
fi
```

IIT Madras
BSc Degree

We are looking at slightly more complex if loop where you have also, else conditions as well as elif condition. So, here on the left hand side you have got if then else loop and on the right hand side you have got a loop that is fairly extensive. So, it is like if, then elif and another elif, and then else and then fi. So, you have 1 2 3 and 4 conditions.

The fourth one is the negation of the condition 3. So, the else is applicable for the previous elif, and therefore, you have got 4 different command sets that can be operated on. So, let us see this in action and try how does that work.

(Refer Slide Time: 00:59)



```
gphani@icme:~/scripts$ cat if-elif-else-example.sh
#!/bin/bash
if [ $# -gt 2 ]
then
        echo arguments given and more than sufficient $#
elif [ $# -gt 1 ]
then
        echo arguments given and sufficient $#
elif [ $# -gt 0 ]
then
        echo arguments given but not sufficient $#
else
        echo arguments are needed $#
fi
gphani@icme:~/scripts$ ./if-elif-else-example.sh a b c
arguments given and more than sufficient 3
gphani@icme:~/scripts$ ./if-elif-else-example.sh a b
arguments given and sufficient 2
gphani@icme:~/scripts$ ./if-elif-else-example.sh a
arguments given but not sufficient 1
gphani@icme:~/scripts$ ./if-elif-else-example.sh
arguments are needed 0
gphani@icme:~/scripts$ 
```

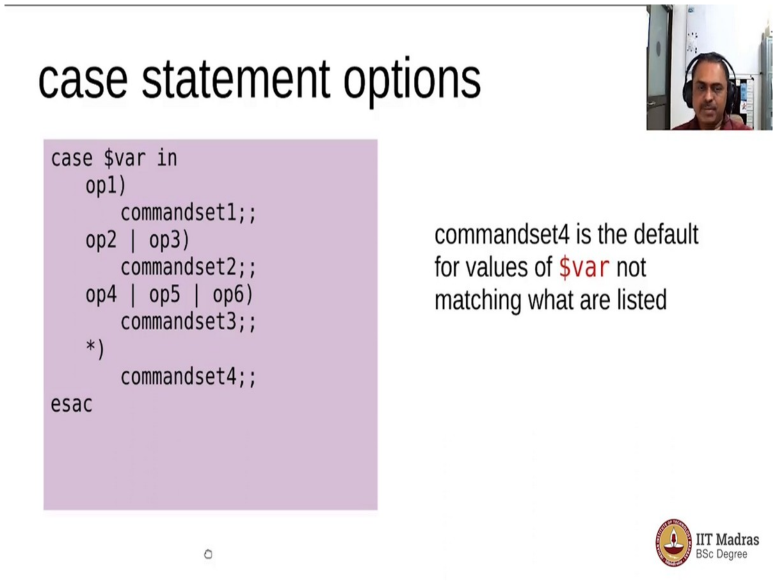
So, the example script for the extensive if elif loop is here. So, what I am doing is that if the number of arguments is greater than 2, then I am saying that the number of arguments given is more than sufficient. So, I am expecting 2 so, if you have given more than 2, I say you have got more than 2 arguments and is the number of arguments is greater than 1, then we are saying the arguments given are sufficient. So, which means that it is processed after the first condition. So, natural needs to be only when there are 2 arguments.

And then the third one is greater than 0, which means it should be 1 argument. So, I am saying argument was given 1 was given, but it is not sufficient. And then if you see else, it is applicable to here, which means it is equal to 0. So, if the number of arguments is 0, then I am saying that arguments are needed. So, this can be executed by using arguments. So, what I will do is I will execute it to display each of these options. So, first, I will execute it with three options.

So, I will say a, b, c. So, there are 3 arguments. So, it says that you have got more than sufficient. Then I give it with 2 arguments, if it says arguments are given sufficient, then I will give 1 argument and it says not sufficient. And then I do not give any argument it says arguments are needed. So, you can see all the four segments have been reproduced. And what we are looking at is just number of arguments to go to each of the loops that we are interested in.

So, you can place any other condition for each of those if elif and the further elif below it as you like, but remember that the logic is going to be checked from the top and therefore you have to always analyze what is being evaluated in the second or third loops carefully seeing what has failed the first condition. So, the if elif loop is quite rich in bash scripts, and you can try that out to test your conditions quite extensively.

(Refer Slide Time: 03:15)



case statement options

```
case $var in
   op1)
      commandset1;;
   op2 | op3)
      commandset2;;
   op4 | op5 | op6)
      commandset3;;
   *)
      commandset4;;
esac
```

commandset4 is the default for values of $var not matching what are listed

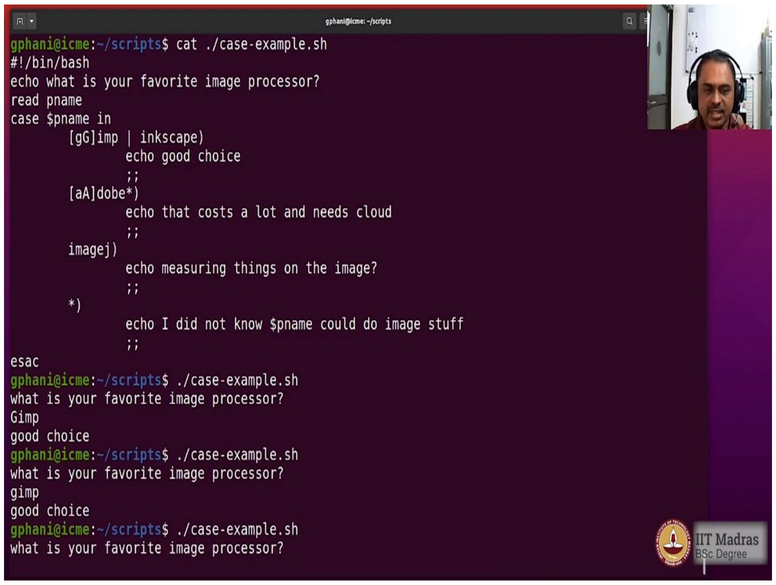
IIT Madras
BSc Degree

Now, you have learned perhaps in C or C plus plus language, that if you have many many else if else if conditions, then perhaps you should be using a case statement. So, in the case of C language, it should be a switch case. And in bash, also it is available. So, it is called the case esac statement, which we have already seen in the previous session.

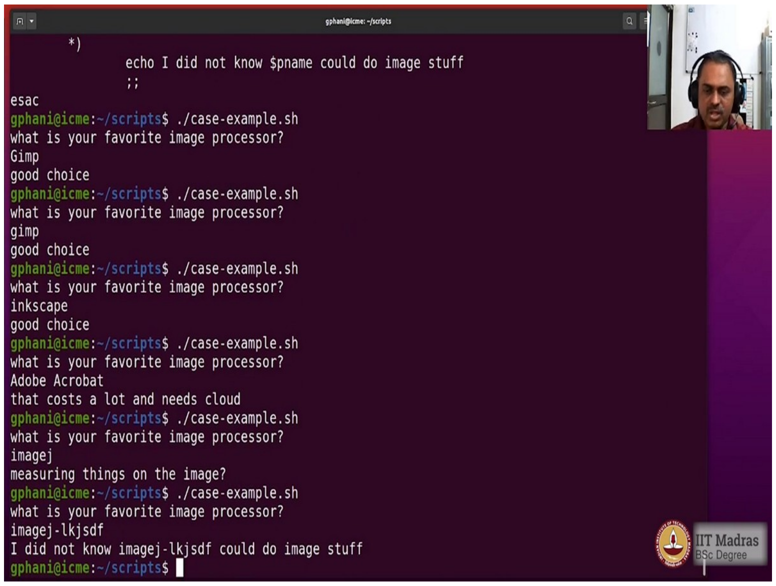
What I want to show you is that within the options that are being matched against the variable, you could also have a pipe to show different combinations that are possible. So, in this example, that is on the screen, you are seeing that if the variable is matching the first option, then the first set of command is executed.

If it matches the second option or third option, then command sets 2 is executed. If it matches either the fourth option fifth option or sixth option, then command set 3 is executed. And if it does not fall into any of them, then by default the command set 4 will be executed. So, let us try this out using an example.

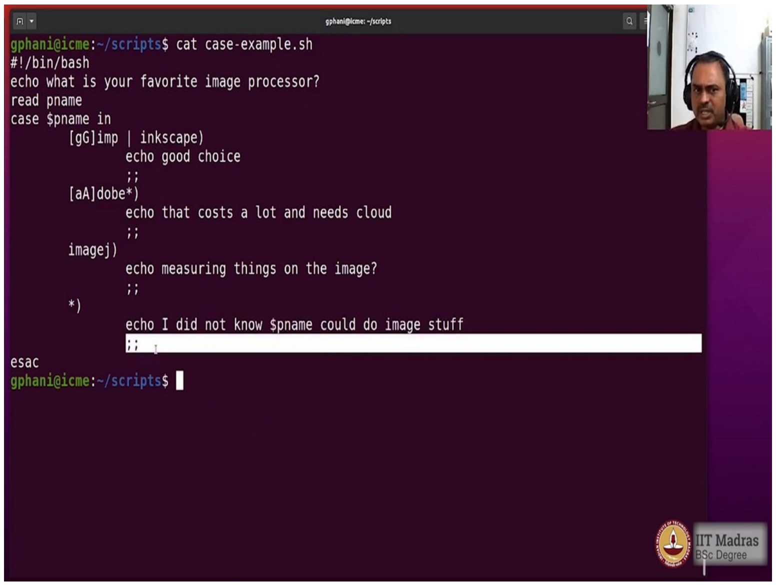
(Refer Slide Time: 04:22)



```
gphani@icme:~/scripts$ cat ./case-example.sh
#!/bin/bash
echo what is your favorite image processor?
read pname
case $pname in
        [gG]imp | inkscape)
                echo good choice
                ;;
        [aA]dobe*)
                echo that costs a lot and needs cloud
                ;;
        imagej)
                echo measuring things on the image?
                ;;
        *)
                echo I did not know $pname could do image stuff
                ;;
esac
gphani@icme:~/scripts$ ./case-example.sh
what is your favorite image processor?
Gimp
good choice
gphani@icme:~/scripts$ ./case-example.sh
what is your favorite image processor?
gimp
good choice
gphani@icme:~/scripts$ ./case-example.sh
what is your favorite image processor?
```



```
        *)
                echo I did not know $pname could do image stuff
                ;;
esac
gphani@icme:~/scripts$ ./case-example.sh
what is your favorite image processor?
Gimp
good choice
gphani@icme:~/scripts$ ./case-example.sh
what is your favorite image processor?
gimp
good choice
gphani@icme:~/scripts$ ./case-example.sh
what is your favorite image processor?
inkscape
good choice
gphani@icme:~/scripts$ ./case-example.sh
what is your favorite image processor?
Adobe Acrobat
that costs a lot and needs cloud
gphani@icme:~/scripts$ ./case-example.sh
what is your favorite image processor?
imagej
measuring things on the image?
gphani@icme:~/scripts$ ./case-example.sh
what is your favorite image processor?
imagej-lkjsdf
I did not know imagej-lkjsdf could do image stuff
gphani@icme:~/scripts$
```



```
gphani@icme:~/scripts$ cat case-example.sh
#!/bin/bash
echo what is your favorite image processor?
read pname
case $pname in
        [gG]imp | inkscape)
                echo good choice
                ;;
        [aA]dobe*)
                echo that costs a lot and needs cloud
                ;;
        imagej)
                echo measuring things on the image?
                ;;
        *)
                echo I did not know $pname could do image stuff
                ;;
esac
gphani@icme:~/scripts$
```

So, the example script to test the case is here. So, what am I doing? I am asking a question, what is your favorite image processor, and then I am picking up the name of the image processor from the prompt. So, I am asking that from the user and whatever was available, then I am going to compare with the options here.

So, either a capital G or small g can be there in the first position. So, if you are going to give the option Gimp or inkscape, then I am giving the output as a good choice. If I see any string that starts with Adobe, either with a small a or capital A, I am going to say that it costs a lot and needs a cloud.

And here if I have an option like images, and I am saying are you measuring things on the image, because it is very good for metrics, and for anything else I am saying that I do not know that such a program could do image stuff. So, let us just run this and see how it is. Remember that the case statement has double semi colons to indicate the end of each of these options. And esac is the way that the case is going to end.

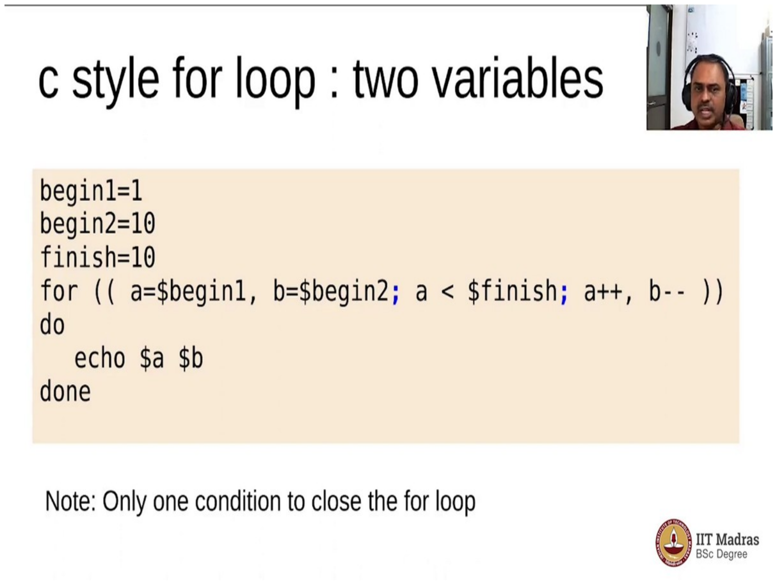
So, let us try this out. And I am going to give the first option. So, I will give Gimp and it says good choice, I try it again, small gimp, it works. And I try it again and give inkscape. And it is working for the first loop. And then I give Adobe Acrobat, and it gives the second option, there is a star here, if you notice, there is a star here, which means that I can give you an option that is slightly longer than what I have listed in this character and if I give image j, then it is giving the fourth option.

Now if I say image j and then I give something else, and you see that I do not match with this because there is no wildcard there. So, it means that I have to exactly match the string and if I did not match, then it will go and fall into the default option in which I am stating that I did not know the name of the option could do image stuff.

So, that is what I am saying here I did not know that image j hyphen lkjsdf could do image stuff. So, you see that the use of star, the use of the bre regular expressions is quite useful in ensuring that the way you are expecting the options to come could be captured as accurately as possible. So, let us look at this case loop again once more.

So, you have got case variable name in after that, the options and then the commands and then double semicolon and again an option command and double semicolon default star command for the default options, which is fallback options and then double semicolon which is optional, the last double semicolon is optional and then esac to close the case loop. So, this is something that is quite useful.

(Refer Slide Time: 07:31)



c style for loop : two variables

```
begin1=1
begin2=10
finish=10
for (( a=$begin1, b=$begin2; a < $finish; a++, b-- ))
do
   echo $a $b
done
```

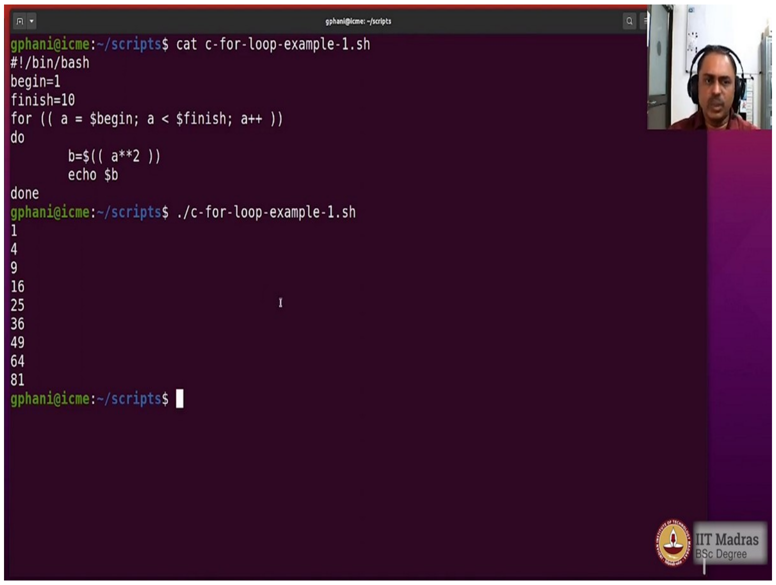
Note: Only one condition to close the for loop

As you would guess the at least the early years of the Linux usage, the programmers are all C programmers, because most of the system commands are all C programs. And therefore, people wanted also, to have some features that are possible as extensions to the POSIX standard to be implemented in bash scripts to mimic some of the loops that they miss came from C language. So, for loop is one such a beautiful construction of language, where you could also, have that in bash, remember that is an extension to the POSIX and may not be available in some of the older shells.

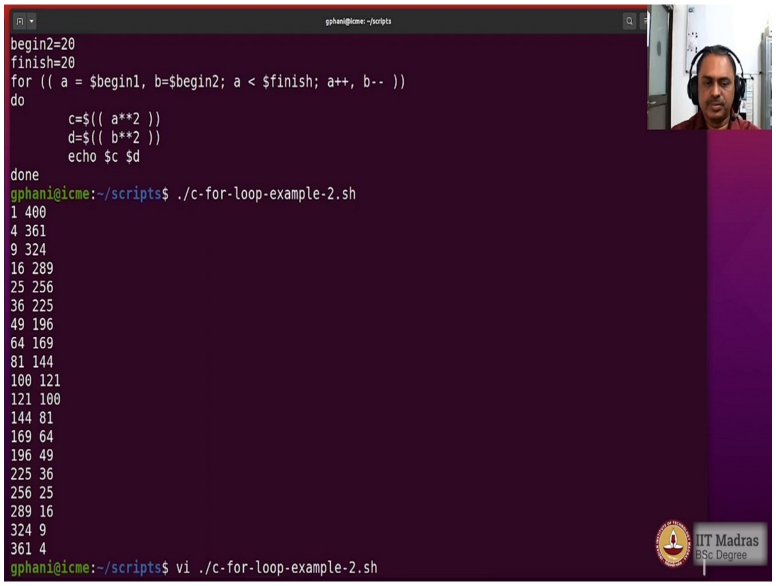
So, the way it is done is as follows. You have the three conditions exactly the same way that it is working in a C language, except that there are no braces to start and close the loop instead you have a do and done to close the loop. And let us just try this script, there are two such loops that are possible. So, you also, have the possibility of multiple variables being initiated multiple increment operations being performed, but you must have a single finish condition.

And you could see that this also, is very useful feature because it allows you to have multiple variables vary in a particular sequence. For example, in this loop, the a is increasing and b is decreasing. And then you are printing them both side by side, the finish condition has to be unambiguous, and that is why there is only one finish condition that is allowed. So, you could have as many variables that are simultaneously changing as you would like. Let us see this in practice how does it look like.

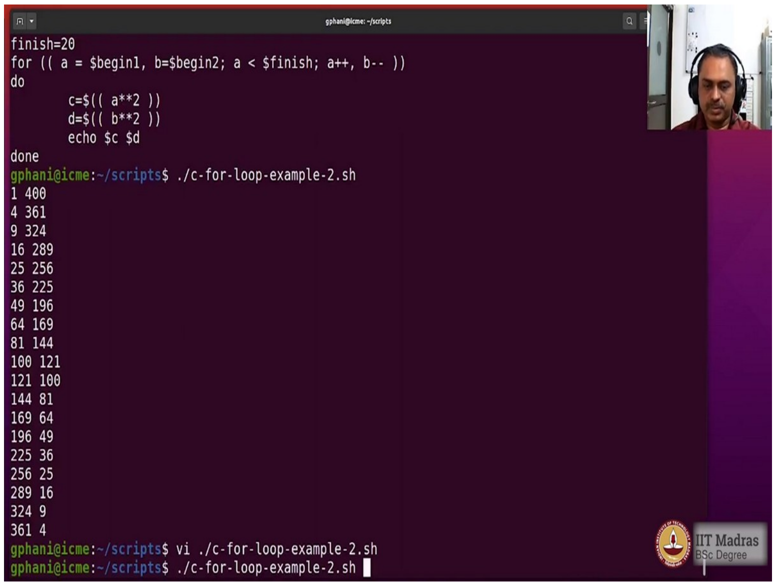
(Refer Slide Time: 09:12)



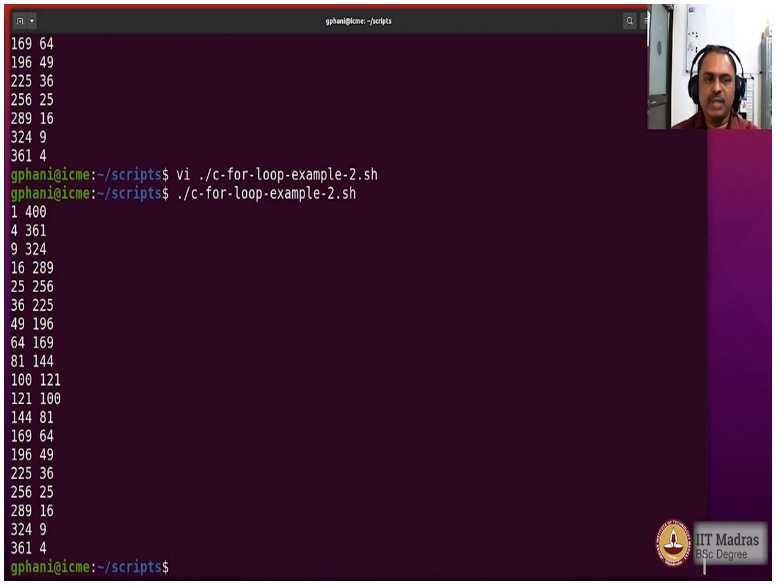
```
gphani@icme:~/scripts$ cat c-for-loop-example-1.sh
#!/bin/bash
begin=1
finish=10
for (( a = $begin; a < $finish; a++ ))
do
        b=$(( a**2 ))
        echo $b
done
gphani@icme:~/scripts$ ./c-for-loop-example-1.sh
1
4
9
16
25
36
49
64
81
gphani@icme:~/scripts$
```



```
begin2=20
finish=20
for (( a = $begin1, b=$begin2; a < $finish; a++, b-- ))
do
        c=$(( a**2 ))
        d=$(( b**2 ))
        echo $c $d
done
gphani@icme:~/scripts$ ./c-for-loop-example-2.sh
1 400
4 361
9 324
16 289
25 256
36 225
49 196
64 169
81 144
100 121
121 100
144 81
169 64
196 49
225 36
256 25
289 16
324 9
361 4
gphani@icme:~/scripts$ vi ./c-for-loop-example-2.sh
```



```
finish=20
for (( a = $begin1, b=$begin2; a < $finish; a++, b-- ))
do
        c=$(( a**2 ))
        d=$(( b**2 ))
        echo $c $d
done
gphani@icme:~/scripts$ ./c-for-loop-example-2.sh
1 400
4 361
9 324
16 289
25 256
36 225
49 196
64 169
81 144
100 121
121 100
144 81
169 64
196 49
225 36
256 25
289 16
324 9
361 4
gphani@icme:~/scripts$ vi ./c-for-loop-example-2.sh
gphani@icme:~/scripts$ ./c-for-loop-example-2.sh
```



```
169 64
196 49
225 36
256 25
289 16
324 9
361 4
gphani@icme:~/scripts$ vi ./c-for-loop-example-2.sh
gphani@icme:~/scripts$ ./c-for-loop-example-2.sh
1 400
4 361
9 324
16 289
25 256
36 225
49 196
64 169
81 144
100 121
121 100
144 81
169 64
196 49
225 36
256 25
289 16
324 9
361 4
gphani@icme:~/scripts$
```

So, the c for loop example the first example is what I have shown you on the slide itself and let us just run that out. And you see that it is just simply displaying the values of a and here I am using only the power operator. So, a to the power 2 a square is being printed. So, from 1 until 10 until but not including because here we are seeing less than, and each time is automatically incremented here in the same way as the language is done.

And remember there is a double parenthesis for the for loop. And here also, there is a double parenthesis because there is an arithmetic operation being performed inside. So, this is a very simple and beautiful loop that does exactly what you would have expected in a C language style loop with a similar kind of a construction.

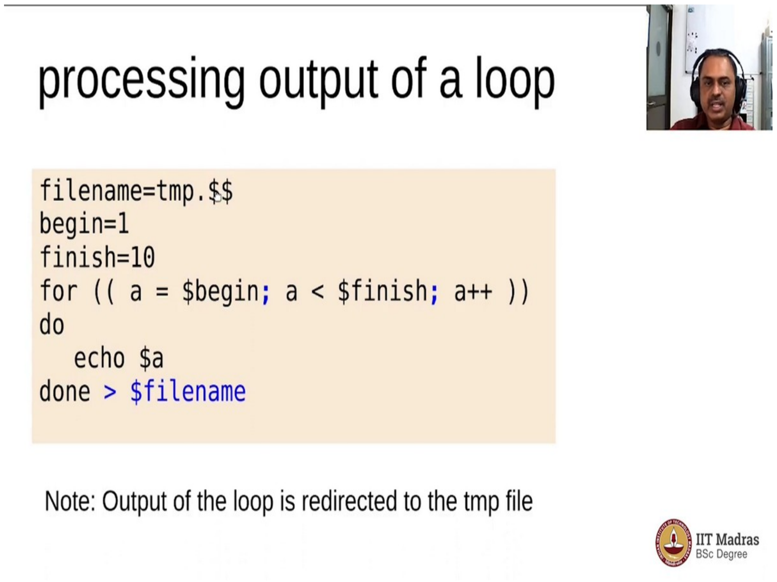
So, we can do this with two variables. So, here what I am doing is I am initializing two variables and incrementing one of them and decrementing the other one the finish condition is given by only one of the two variables, and what I am doing is that I am squaring the first one and the second one and displaying them side by side.

So, let us look at the output. So, as you could expect we are going from 1 to 20. So, from 1 to 20 but not including 20. So, 1 to 19 so, 1 to 19 you are having the square that is displayed, but the b is going in the reverse direction from 20 to 2 and it is also, gain it is also, being squared and displayed. So, this construction where you have got multiple initializations and multiple increments is also sometimes very useful for certain operations.

So, let us modify this 2 see how does it work, in case you are looking at the variability. So, let us check if I can use the different variable here. So, now, because b is starting from 20 and coming upward. So, I would increase it. And the finish would then should be something a small number. So, I will put as 1.

And let us go ahead and try this out. And you will see that the same output has come. So, you can see that you can have the finish condition for 1 of the two variables, but it is up to you which one you want to choose.

(Refer Slide Time: 11:46)



processing output of a loop

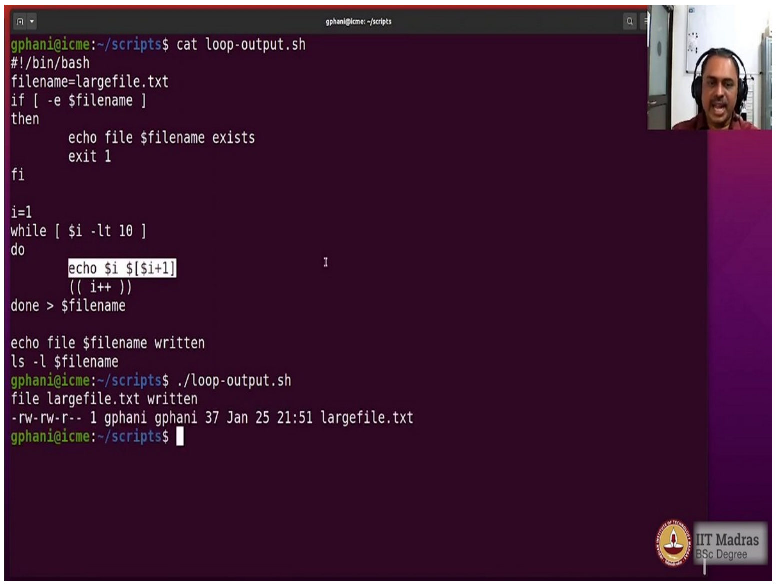
```
filename=tmp.$$
begin=1
finish=10
for (( a = $begin; a < $finish; a++ ))
do
   echo $a
done > $filename
```

Note: Output of the loop is redirected to the tmp file

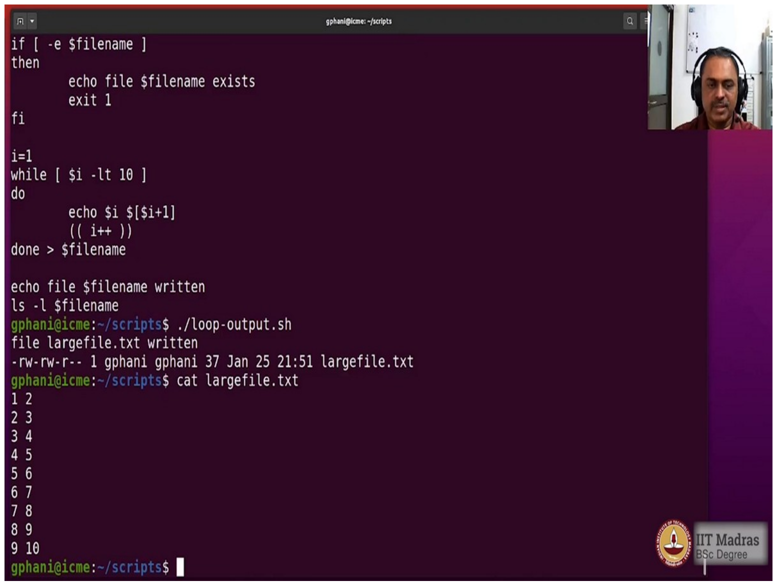
Now there are sometimes requirements that the output of a loop is stored in a file. So, that it can be processed later, you can always split the script in such a way that you can run the script, redirect the output of the script to a file, and then have another script to look at that file to do further processing. So, you can always split the job and do it in segments. But it is elegant if you can also, do some of it within the shell. And a very nice feature is available here.

So, you have the loop that is written exactly like any other loop. But after the end of the loop, if you have a symbol to mention the redirection of the output and the file name is given, then the output of the loop alone is going to be put in that particular file. And here, the file name is given as a temporary file with a dot followed by the process ID. So, that there is likely no confusion with other files. And the output can then be stored in the file name. So, let us just check. How does this look like.

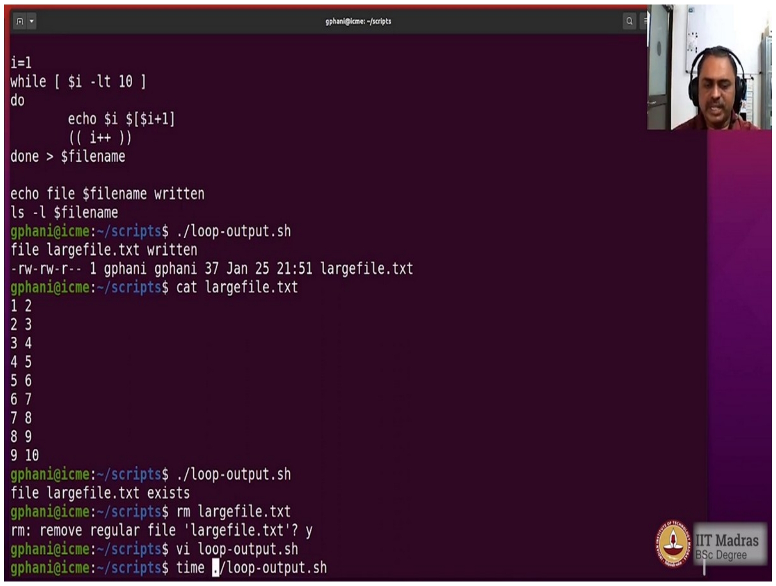
(Refer Slide Time: 12:48)



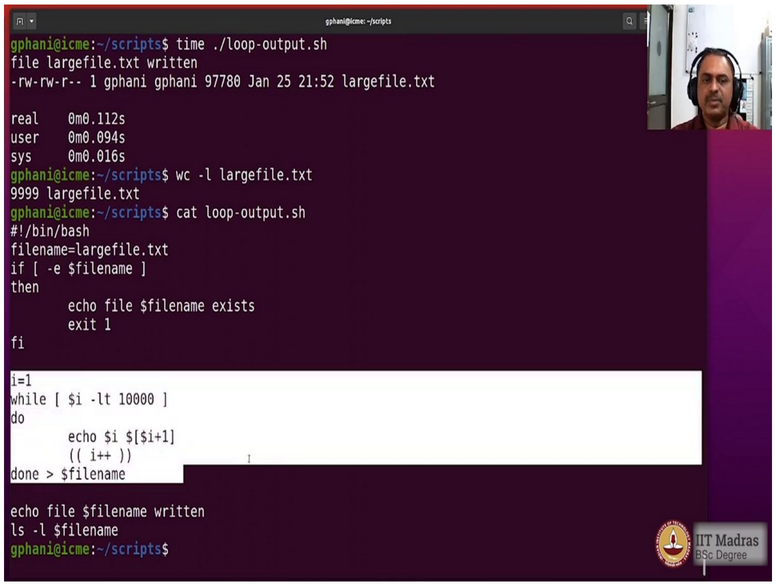
```
gphani@icme:~/scripts$ cat loop-output.sh
#!/bin/bash
filename=largefile.txt
if [ -e $filename ]
then
        echo file $filename exists
        exit 1
fi

i=1
while [ $i -lt 10 ]
do
        echo $i $[$i+1]
        (( i++ ))
done > $filename

echo file $filename written
ls -l $filename
gphani@icme:~/scripts$ ./loop-output.sh
file largefile.txt written
-rw-rw-r-- 1 gphani gphani 37 Jan 25 21:51 largefile.txt
gphani@icme:~/scripts$
```



```
if [ -e $filename ]
then
        echo file $filename exists
        exit 1
fi

i=1
while [ $i -lt 10 ]
do
        echo $i $[$i+1]
        (( i++ ))
done > $filename

echo file $filename written
ls -l $filename
gphani@icme:~/scripts$ ./loop-output.sh
file largefile.txt written
-rw-rw-r-- 1 gphani gphani 37 Jan 25 21:51 largefile.txt
gphani@icme:~/scripts$ cat largefile.txt
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
gphani@icme:~/scripts$
```



```
i=1
while [ $i -lt 10 ]
do
        echo $i $[$i+1]
        (( i++ ))
done > $filename

echo file $filename written
ls -l $filename
gphani@icme:~/scripts$ ./loop-output.sh
file largefile.txt written
-rw-rw-r-- 1 gphani gphani 37 Jan 25 21:51 largefile.txt
gphani@icme:~/scripts$ cat largefile.txt
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
gphani@icme:~/scripts$ ./loop-output.sh
file largefile.txt exists
gphani@icme:~/scripts$ rm largefile.txt
rm: remove regular file 'largefile.txt'? y
gphani@icme:~/scripts$ vi loop-output.sh
gphani@icme:~/scripts$ time ./loop-output.sh
```



```
gphani@icme:~/scripts$ time ./loop-output.sh
file largefile.txt written
-rw-rw-r-- 1 gphani gphani 97780 Jan 25 21:52 largefile.txt

real    0m0.112s
user    0m0.094s
sys     0m0.016s
gphani@icme:~/scripts$ wc -l largefile.txt
9999 largefile.txt
gphani@icme:~/scripts$ cat loop-output.sh
#!/bin/bash
filename=largefile.txt
if [ -e $filename ]
then
        echo file $filename exists
        exit 1
fi

i=1
while [ $i -lt 10000 ]
do
        echo $i $[$i+1]
        (( i++ ))
done > $filename

echo file $filename written
ls -l $filename
gphani@icme:~/scripts$
```

So, here is the loop output script. And I am doing something a little bit more here. So, I am having the file name as large file dot text, and I am checking whether such a file exists. So, if it exists, what I am doing is that I am refusing to go further mentioning that it is existing file. So, I do not go further I exit, which means that I do not want to over write the file.

And if such a file does not exist, then I can go on to redirect my output onto that file. And that is what is being done in this loop. And what am I doing I am outputting two values, i and i plus 1, and how many of them, 10 of them. And do not forget to increment i because otherwise you will be entering an infinite loop because i is 1. And while i is less than 10 the loop is running. And that unless you increment you do not reach the end of the loop. Do not forget that.

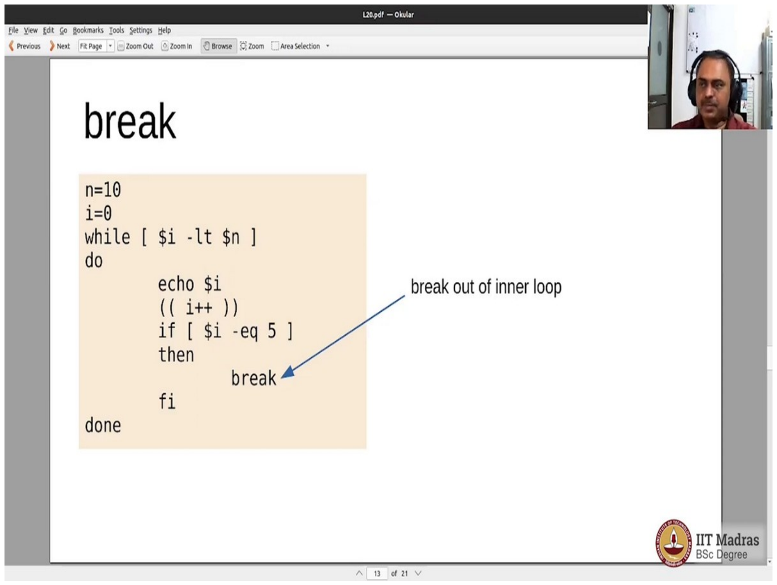
And I am writing two numbers side by side. And when it is done, then it is going to be written to the file name. And after that, I am going to state that file has been written and just check the size of the file. So, let us run this out. And you do not see the output of the echo command of this position coming onto the screen because it is already redirected to the file. So, it has been written and you can see that the file has been written and let us just check the output of the large file, and it has the lines that we expected.

Now, if I run it a second time, what would you expect? You see that because the filename is already existing. So, the file descript would not proceed further and stop. So, this is a check that you can have for yourself. So, that you do not overwrite files if you are not intending to do that. So, what I would do now is I will remove that file, and perhaps I will go ahead and edit it.

So, that instead of 10 numbers, I would actually have let us say 10,000 numbers and then I will run the loop, and I want to also, find out how much time it takes to run this loop. So, I will put a time command in front of it time followed by the script, and then you can see that it runs and for very small time, I mean, it runs in less than a second and you are able to see, almost 100 kilobytes of the data has been generated and has been put in the large file w c minus l large file will tell you that there are 10,000 lines, it has been done in a jiffy.

So, you can produce numerical output to prepare for some other processing or some testing quite fast using these loop conditions or writing the data to files and I do not think one can write a file with. So, much of data in such an elegant fashion as what we have done till now. So, look at the script, it is so, elegant, you are writing the data to a file in just couple of lines.

(Refer Slide Time: 15:53)



break

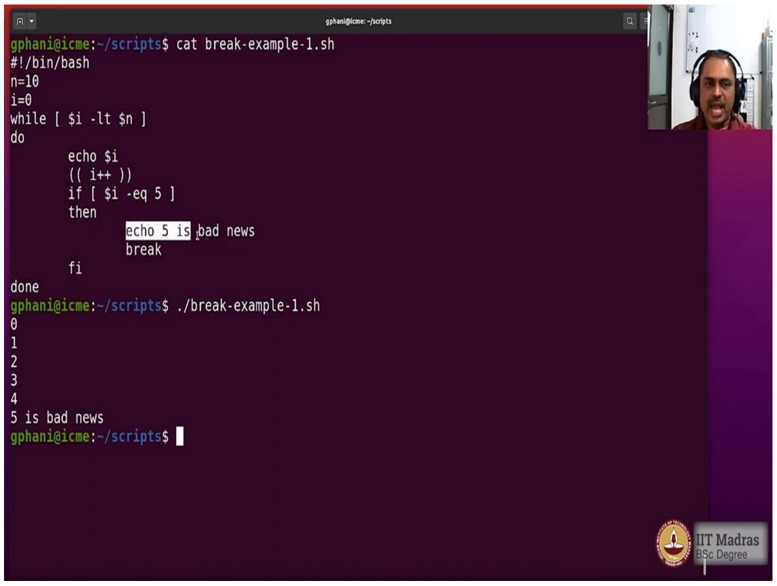
```
n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        (( i++ ))
        if [ $i -eq 5 ]
        then
                break
        fi
done
```

Sometimes there are situations where you would like to break out of a loop. So, here is a situation where we have two loops. And we would like to break out of the inner loop, if we encounter a number here in this case, it is number 5. So, we are writing numbers from 0 to 10. And then we are incrementing the i and then checking whether it is equal to 5 and if it has reached 5, then we break.

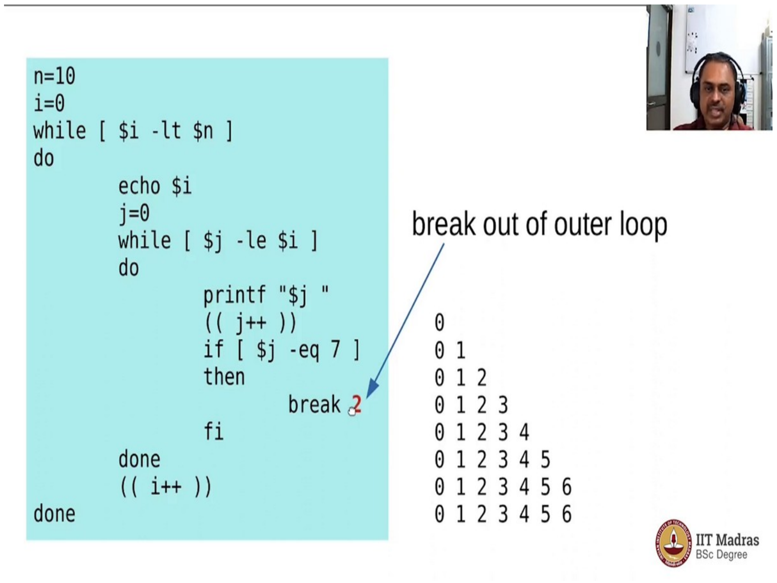
So, it is just to test how do we break out of a loop the test condition here is silly, but you could have any other test condition. Sometimes there are situations where you are doing some file related activities like creating backup files or something and you may want to have a test condition to check how much free space is available on the disk. And if it is less than, say 5 percent. And you do not want to go further and do any more file generation and you may want to come out of the loop by a break.

So, some such possibilities are there in real life. So, you want to come out of a loop is in break and that is entirely possible. So, the condition with which the break is going to be applied is here. And let us try that out with an example.

(Refer Slide Time: 17:07)



```
gphani@icme:~/scripts$ cat break-example-1.sh
#!/bin/bash
n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        (( i++ ))
        if [ $i -eq 5 ]
        then
                echo 5 is bad news
                break
        fi
done
gphani@icme:~/scripts$ ./break-example-1.sh
0
1
2
3
4
5 is bad news
gphani@icme:~/scripts$
```



```
n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        j=0
        while [ $j -le $i ]
        do
                printf "$j "
                (( j++ ))
                if [ $j -eq 7 ]
                then
                        break 2
                fi
        done
        (( i++ ))
done
```

break out of outer loop

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6
```

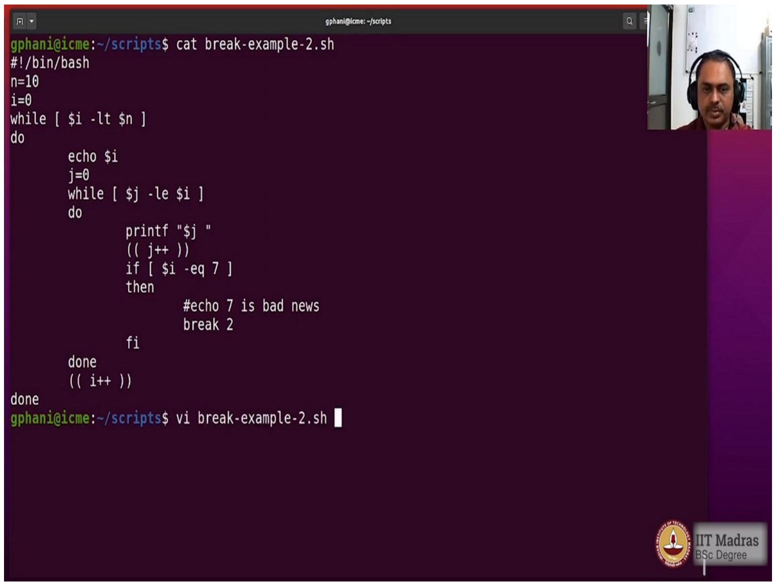
So, the example of the break condition is here. So, we are going from 0 all the way to 10. And when you equal to 5, then we output a string saying 5 is bad news break. So, what do you expect to see as an output. So, you will go from 0 to 5 and then after that, it does not go further. So, the way to use the break is that you have a condition to break.

So, there is a if followed by a condition and then within the condition you may want to have an in message printed before you break which is useful for debugging later on. Sometimes when you are inside a loop and you want to break not from inner loop but from the outer loop also. So, if you have n number of loops that are nested, then you may choose to break out of any of those n loops.

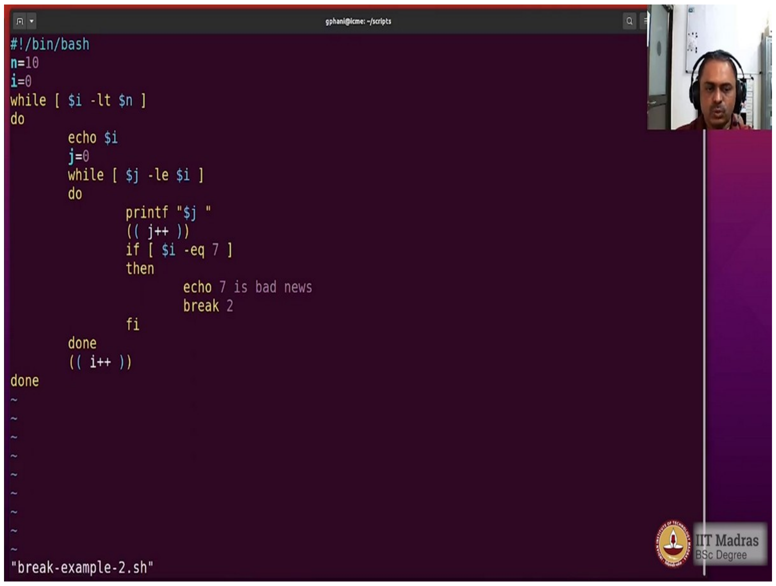
So, a break followed by an integer will tell you up to what level you want to traverse upwards and then break. So, if you say break 2 it means that it is going to break out of not this loop, but one level up that is this loop. So, you could choose to have that and here is a situation where what we are doing is we are printing the numbers and the j value is going horizontally without a break. So, you can see printf without a newline. So, there is going horizontally and whenever we finished the j printing up to the value of i, then i is incremented and then i is printed on the screen.

So, you see that it is like showing your triangle of numbers and we are seeing that whenever the j reaches 7, then we want to come out of that. So, that is up to 6 the triangle is full but the seventh row at the last digit of 7 to be printed then we have a problem we want to come out of that loop and we want to stop executing completely after that. So, that is what is being done with the break 2 as a condition here. So, let us try that out.

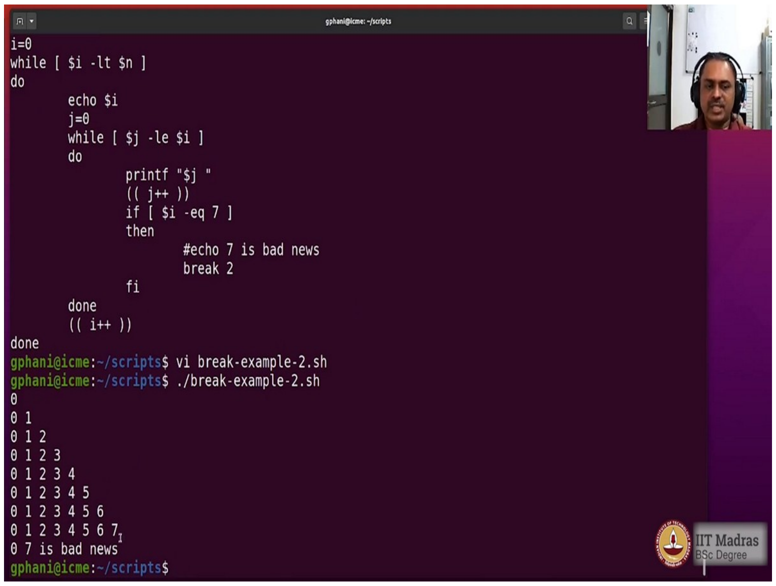
(Refer Slide Time: 19:05)



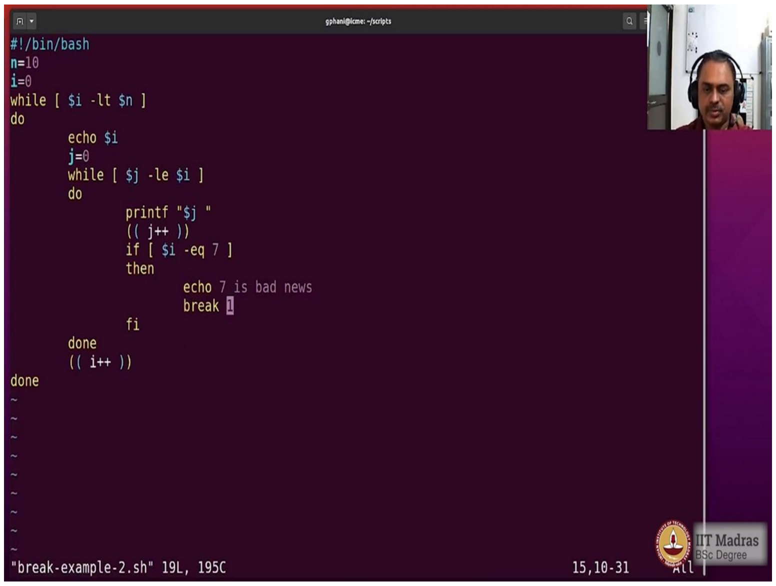
```
gphani@icme:~/scripts$ cat break-example-2.sh
#!/bin/bash
n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        j=0
        while [ $j -le $i ]
        do
                printf "$j "
                (( j++ ))
                if [ $i -eq 7 ]
                then
                        #echo 7 is bad news
                        break 2
                fi
        done
        (( i++ ))
done
gphani@icme:~/scripts$ vi break-example-2.sh
```



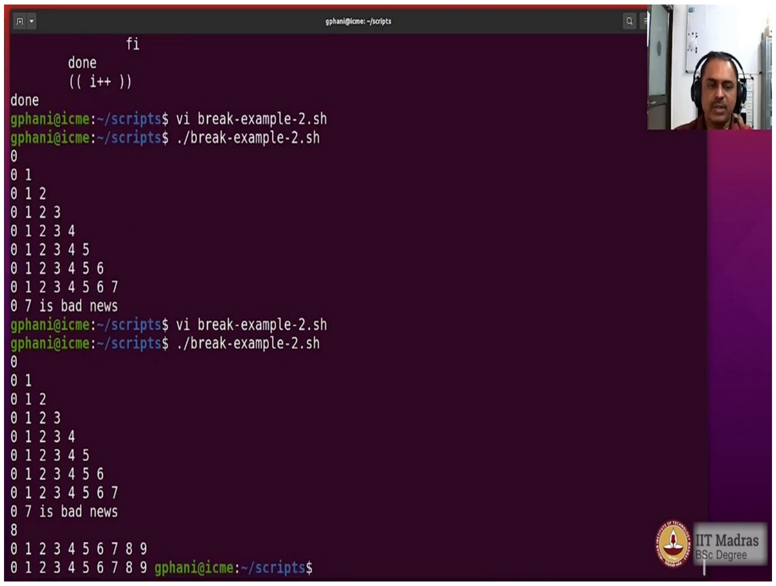
```
#!/bin/bash
n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        j=0
        while [ $j -le $i ]
        do
                printf "$j "
                (( j++ ))
                if [ $i -eq 7 ]
                then
                        echo 7 is bad news
                        break 2
                fi
        done
        (( i++ ))
done
~
~
~
~
~
~
~
~
~
"break-example-2.sh"
```



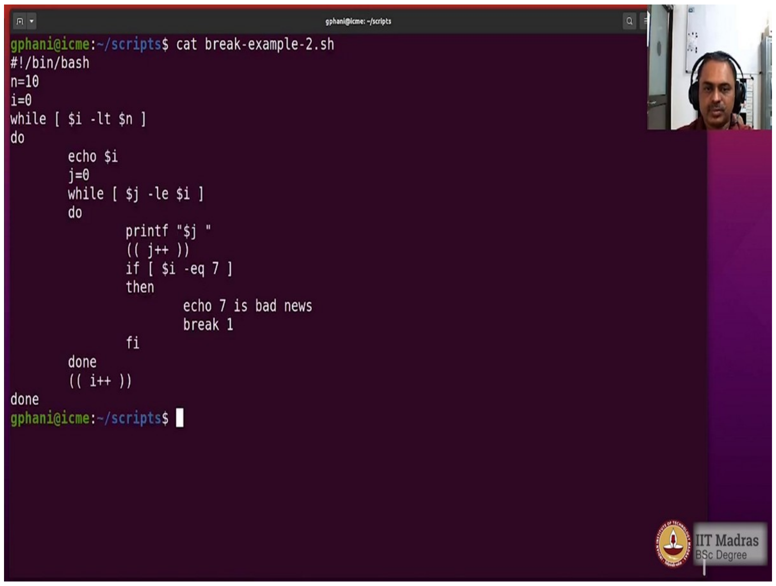
```
i=0
while [ $i -lt $n ]
do
        echo $i
        j=0
        while [ $j -le $i ]
        do
                printf "$j "
                (( j++ ))
                if [ $i -eq 7 ]
                then
                        #echo 7 is bad news
                        break 2
                fi
        done
        (( i++ ))
done
gphani@icme:~/scripts$ vi break-example-2.sh
gphani@icme:~/scripts$ ./break-example-2.sh
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7
0 7 is bad news
gphani@icme:~/scripts$
```



```
#!/bin/bash
n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        j=0
        while [ $j -le $i ]
        do
                printf "$j "
                (( j++ ))
                if [ $i -eq 7 ]
                then
                        echo 7 is bad news
                        break 1
                fi
        done
        (( i++ ))
done
~
~
~
~
~
~
~
~
~
"break-example-2.sh" 19L, 195C                                15,10-31      All
```



```
                fi
        done
        (( i++ ))
done
gphani@icme:~/scripts$ vi break-example-2.sh
gphani@icme:~/scripts$ ./break-example-2.sh
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7
0 7 is bad news
gphani@icme:~/scripts$ vi break-example-2.sh
gphani@icme:~/scripts$ ./break-example-2.sh
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7
0 7 is bad news
8
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 gphani@icme:~/scripts$
```

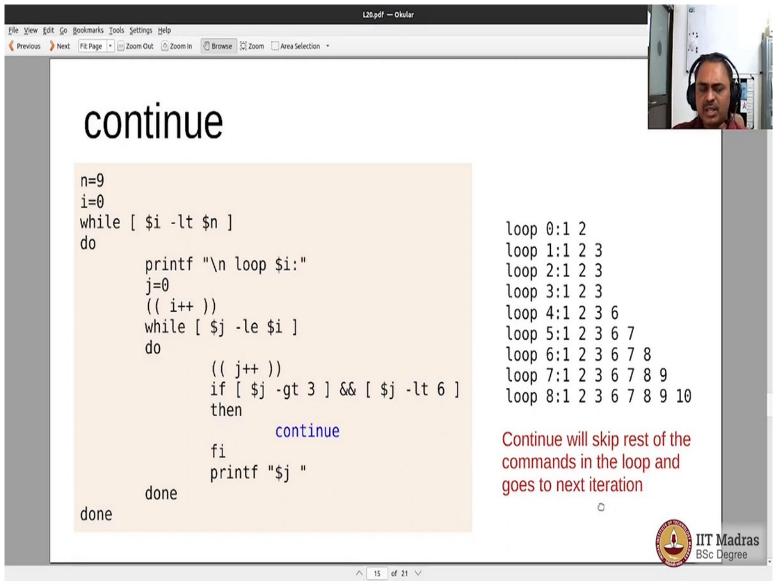


```
gphani@icme:~/scripts$ cat break-example-2.sh
#!/bin/bash
n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        j=0
        while [ $j -le $i ]
        do
                printf "$j "
                (( j++ ))
                if [ $i -eq 7 ]
                then
                        echo 7 is bad news
                        break 1
                fi
        done
        (( i++ ))
done
gphani@icme:~/scripts$ 
```

So, here is the loop that we have seen in the slide and we are breaking out and we also, have a echo command which I would like to uncomment now and then I will break out of that. So, you see that as soon as the 7 is displayed, then you are coming out of that loop. And I can also, come out of the first loop and you see that the display is a bit different because when you come out of the first loop, then the outer loop is still when I come out of the inner loop, the outer loop is still available. So, it is printing upto 9.

So, you can see that the way you want to break out of loops can be quite different in different purposes. So, here is the number that you should tell whether you want to break out of this loop then you have to say break 1, this loop then break 2. So, I have put break 1, which means I am coming out of this loop.

(Refer Slide Time: 20:24)



continue

```
n=9
i=0
while [ $i -lt $n ]
do
        printf "\n loop $i:"
        j=0
        (( i++ ))
        while [ $j -le $i ]
        do
                (( j++ ))
                if [ $j -gt 3 ] && [ $j -lt 6 ]
                then
                        continue
                fi
                printf "$j "
        done
done
```

```
loop 0:1 2
loop 1:1 2 3
loop 2:1 2 3
loop 3:1 2 3
loop 4:1 2 3 6
loop 5:1 2 3 6 7
loop 6:1 2 3 6 7 8
loop 7:1 2 3 6 7 8 9
loop 8:1 2 3 6 7 8 9 10
```

Continue will skip rest of the commands in the loop and goes to next iteration

There are situations in a loop where you do not want to break but you do not want to process commands that are below a particular level, and you want to continue with other iterations that is you have a set of commands that are in the loop and you do not want to process some of them, if you have reached a particular condition and then want to do the next iteration without any problem.

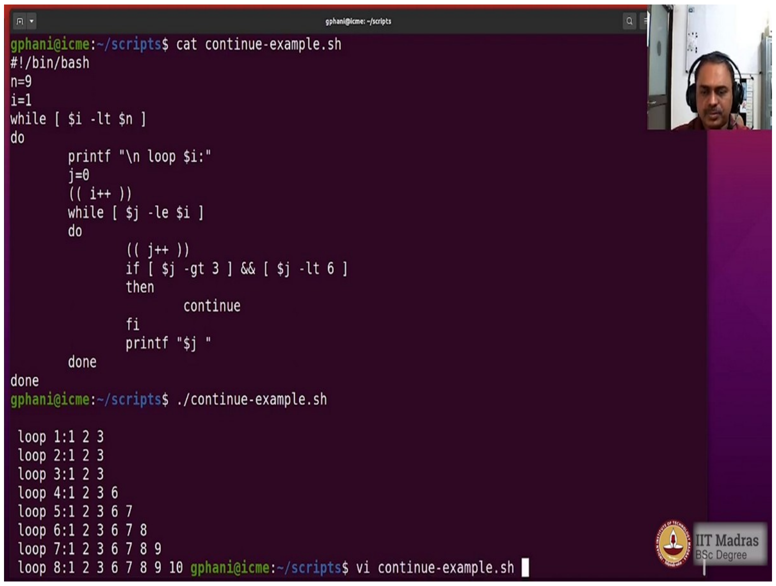
So, which means that you would like to continue the loop by skipping the remaining commands. So, you have the situation here the continue is used, what it says is that whenever the j value is between 4 and 5, then you do not want to print and you can see that here on the output the 4 and 5 are not being printed.

And for other j values, this part is not applicable. So, it goes on to print those values. So, you have seen that 1 2 3 is been printed 6 7 8 9 10 also, are being printed. So, the continuous telling that do not process the commands that come after it, that means the printf dollars j is not being processed under what circumstances whenever j value is greater than 3 and less than 6.

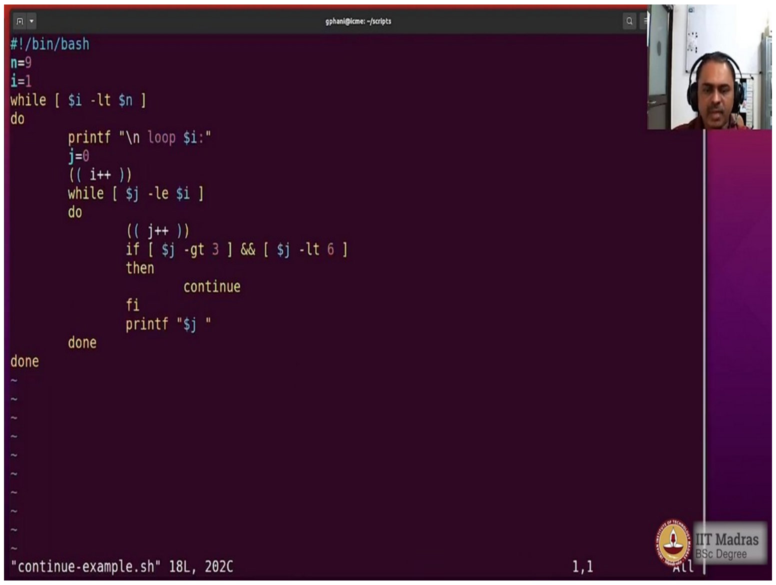
That is whenever the j value is taking value of 4 and 5, do not execute the command that follows the loop that is printf dollars j. So, do not print the j that is what we are telling here, i is printed with a loop in the front and therefore, that is a separate number that we are seeing here the j is what is printed in a horizontal line after the colon and j values of 4 and 5 are being skipped.

Because you can come out of the loop skipping the commands below it, but continue with the iteration process by using the continue feature.

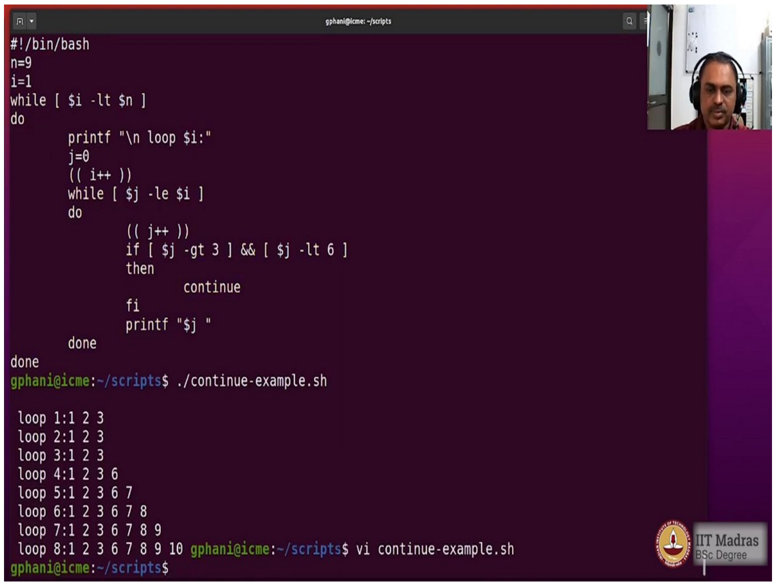
(Refer Slide Time: 22:11)



```
gphani@icme:~/scripts$ cat continue-example.sh
#!/bin/bash
n=9
i=1
while [ $i -lt $n ]
do
        printf "\n loop $i:"
        j=0
        (( i++ ))
        while [ $j -le $i ]
        do
                (( j++ ))
                if [ $j -gt 3 ] && [ $j -lt 6 ]
                then
                        continue
                fi
                printf "$j "
        done
done
gphani@icme:~/scripts$ ./continue-example.sh

 loop 1:1 2 3
 loop 2:1 2 3
 loop 3:1 2 3
 loop 4:1 2 3 6
 loop 5:1 2 3 6 7
 loop 6:1 2 3 6 7 8
 loop 7:1 2 3 6 7 8 9
 loop 8:1 2 3 6 7 8 9 10 gphani@icme:~/scripts$ vi continue-example.sh
```



```
#!/bin/bash
n=9
i=1
while [ $i -lt $n ]
do
        printf "\n loop $i:"
        j=0
        (( i++ ))
        while [ $j -le $i ]
        do
                (( j++ ))
                if [ $j -gt 3 ] && [ $j -lt 6 ]
                then
                        continue
                fi
                printf "$j "
        done
done
~
~
~
~
~
~
~
~
~
~
"continue-example.sh" 18L, 202C                              1,1           All
```

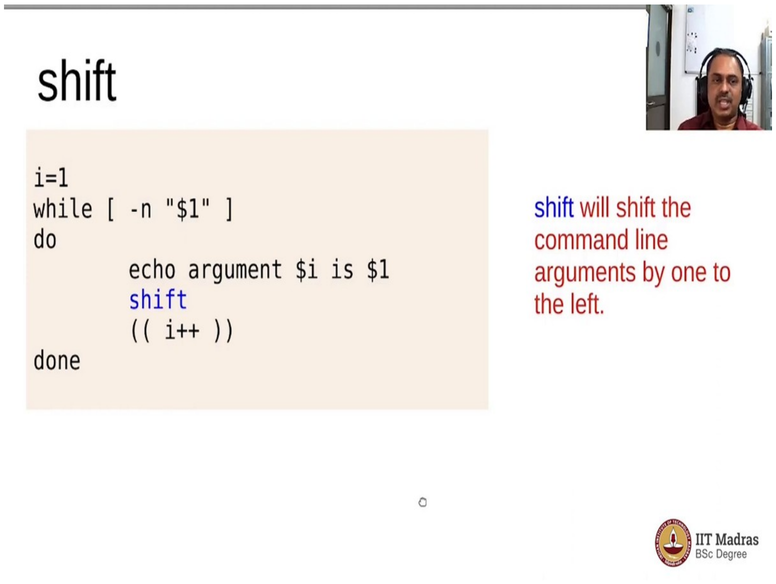


```
#!/bin/bash
n=9
i=1
while [ $i -lt $n ]
do
        printf "\n loop $i:"
        j=0
        (( i++ ))
        while [ $j -le $i ]
        do
                (( j++ ))
                if [ $j -gt 3 ] && [ $j -lt 6 ]
                then
                        continue
                fi
                printf "$j "
        done
done
gphani@icme:~/scripts$ ./continue-example.sh

 loop 1:1 2 3
 loop 2:1 2 3
 loop 3:1 2 3
 loop 4:1 2 3 6
 loop 5:1 2 3 6 7
 loop 6:1 2 3 6 7 8
 loop 7:1 2 3 6 7 8 9
 loop 8:1 2 3 6 7 8 9 10 gphani@icme:~/scripts$ vi continue-example.sh
gphani@icme:~/scripts$
```

So, here is this script that we have seen just now. And we execute that to see that the output is what we have seen on the slide we will now modify that and say that instead of less than 6, I would make it as less than 7 and you see that the output is slightly different it is now between 3 and 7 that are being skipped and otherwise remaining ones are being printed.

So, which means that only one line is being skipped only this command is being skipped. Other than that, the loop is proceeding to continue. So, the continue command that way takes effect only for what is below it.

(Refer Slide Time: 22:59)



shift

```
i=1
while [ -n "$1" ]
do
        echo argument $i is $1
        shift
        (( i++ ))
done
```

shift will shift the command line arguments by one to the left.

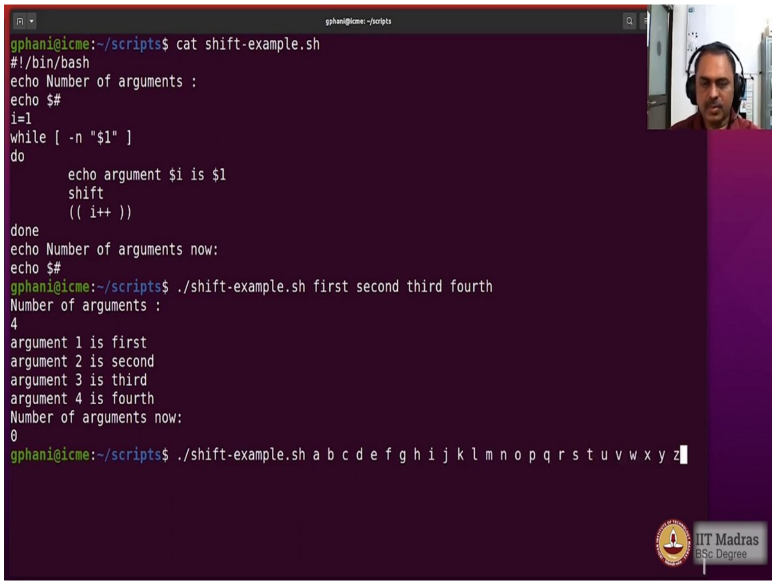
There is another keyword called shift what this does is quite destructive, it would actually shift the arguments by 1 and then reset the positional variables like dollar 1 dollar 2 et cetera. The values are reset. So, that you can refer to the variables after finishing up the processing of some of the variables.

So, you can shift through the variables and then process the arguments. But whatever has been shifted to the left are all lost. Exception is dollar 0 and that refers to the name of the script itself and that will not be touched here is one example of the shift command. So, what we are doing is that we are running through the list of arguments dollar 1 means first argument we are putting in quotes.

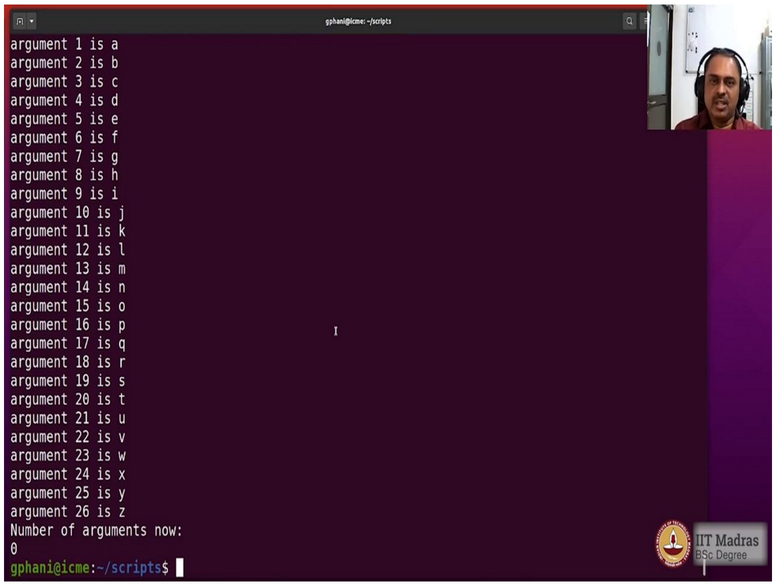
So, that it is treated as a string and then minus n we are asking whether it is a string with nonzero length and if so, then we go on to print the argument number and the value of the argument and we are shifting. And then, we are incrementing i which means that we are printing 1 2 3 4 et cetera as i is incremented but each time we are shifting the arguments one after the other.

So, that we are processing all the arguments one after the other and when we are done with the loop there will be no argument left because minus n will fail when the or arguments are processed and therefore the loop will end. So, let us try this out in shell and see how does it work.

(Refer Slide Time: 24:33)



```
gphani@icme:~/scripts$ cat shift-example.sh
#!/bin/bash
echo Number of arguments :
echo $#
i=1
while [ -n "$1" ]
do
        echo argument $i is $1
        shift
        (( i++ ))
done
echo Number of arguments now:
echo $#
gphani@icme:~/scripts$ ./shift-example.sh first second third fourth
Number of arguments :
4
argument 1 is first
argument 2 is second
argument 3 is third
argument 4 is fourth
Number of arguments now:
0
gphani@icme:~/scripts$ ./shift-example.sh a b c d e f g h i j k l m n o p q r s t u v w x y z
```



```
argument 1 is a
argument 2 is b
argument 3 is c
argument 4 is d
argument 5 is e
argument 6 is f
argument 7 is g
argument 8 is h
argument 9 is i
argument 10 is j
argument 11 is k
argument 12 is l
argument 13 is m
argument 14 is n
argument 15 is o
argument 16 is p
argument 17 is q
argument 18 is r
argument 19 is s
argument 20 is t
argument 21 is u
argument 22 is v
argument 23 is w
argument 24 is x
argument 25 is y
argument 26 is z
Number of arguments now:
0
gphani@icme:~/scripts$
```

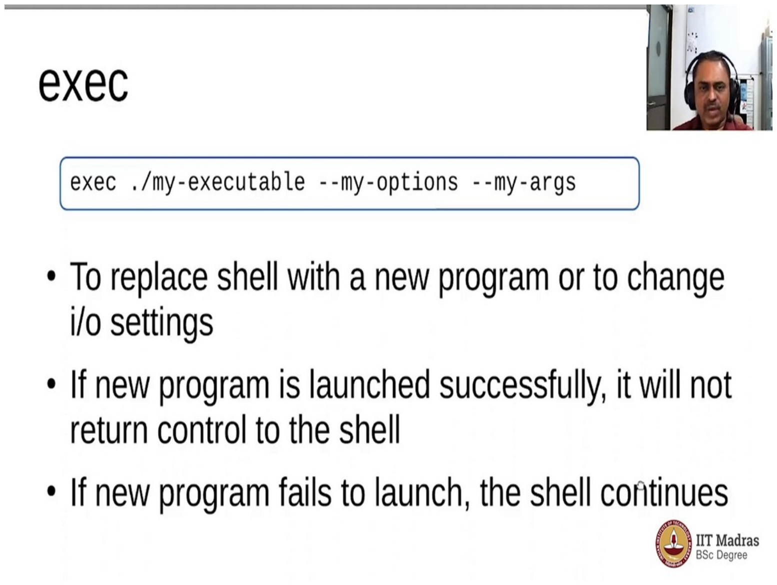
So, here is the example of the shift keyword and what we are doing is we are initially reporting how many arguments are there. After that, we are printing the first argument and then shifting. So, when I shift, the second argument will come to the first position. And I want to note down that is a second argument.

So, the i variable is incremented by 1 using the arithmetic operation And how long is this loop going to run it is going to run until there is no argument available that means that if all the arguments are gone, then the dollar 1 would have null and therefore, the loop will stop. And after it is over, then I want to also, print the number of arguments, and you should see that this is to be 0.

So, let us try this out. So, first, second, third, fourth, and you can see that all the four arguments have come. So, without using the dollar 2 dollar 3 or dollar 4, I am able to access the second third and fourth arguments, because of the shift operator that is available. But it is only good in case you do not need those arguments any further after the processing, because it is destructive in nature, after the arguments are shifted to the left, then they have gone out.

This is also, useful in processing unknown number of arguments one after other. So, let us look at that as an example. So, you can see that all the 26 arguments have been processed one after other and in the end I have got 0 arguments left. So warning, you can use the shift when you do not need the arguments any further. Because, after shifting to the left the arguments are lost.

(Refer Slide Time: 26:26)



exec

```
exec ./my-executable --my-options --my-args
```

- To replace shell with a new program or to change i/o settings
- If new program is launched successfully, it will not return control to the shell
- If new program fails to launch, the shell continues

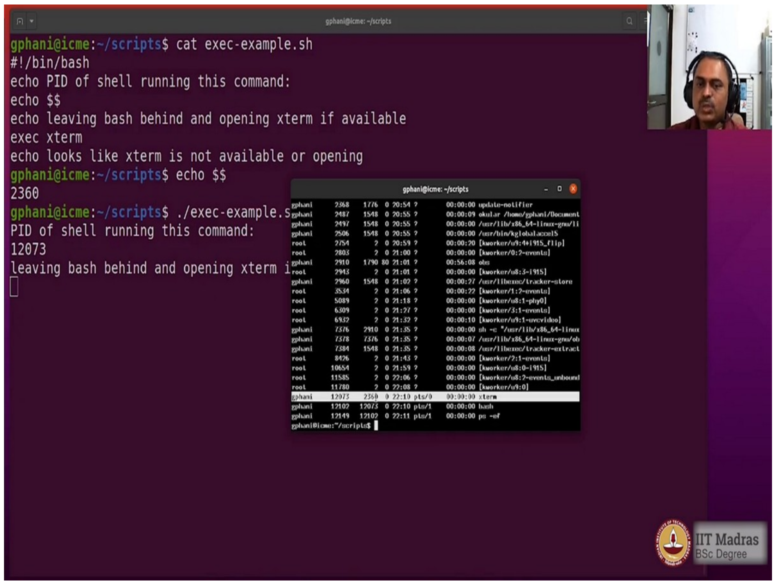
IIT Madras
BSc Degree

There is a command called exec it has a lot of features with respect to redirection of input output, we are not looking at them. Now, it is also, used to replace the shell with a new program. And that is what we will see. And if the new program has been launched successfully, then the control will not be given back to the old shell because it has been replaced. But in case the new program has not been launched, then the control will retain with the shell and you can go on to do other commands.

So, usually the exit command is used to launch a different programs which would take further action and the initial shell that has been used to launch it is no longer required. So, situations like updating packages, and compiling and installing modules, et cetera, that are done by app would use a feature something like this.

Because the shell in which such commands are running are no longer needed once the job is over. So, let us try how does this command work. And for that, I will use a an app called xterm to try how to launch an app from the bash shell.

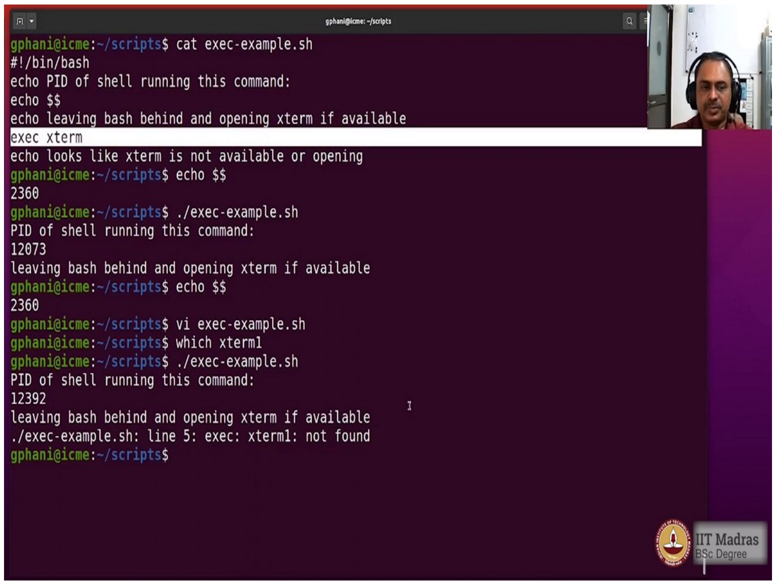
(Refer Slide Time: 27:44)



```
gphani@icme:~/scripts$ cat exec-example.sh
#!/bin/bash
echo PID of shell running this command:
echo $$
echo leaving bash behind and opening xterm if available
exec xterm
echo looks like xterm is not available or opening
gphani@icme:~/scripts$ echo $$
2360
gphani@icme:~/scripts$ ./exec-example.s
PID of shell running this command:
12073
leaving bash behind and opening xterm i
```

gphani@icme: ~/scripts

```
gphani     2368   1776  0 20:54 ?        00:00:00 update-notifier
gphani     2487   1548  0 20:55 ?        00:00:09 okular /home/gphani/Document
gphani     2497   1548  0 20:55 ?        00:00:00 /usr/lib/x86_64-linux-gnu/li
gphani     2506   1548  0 20:55 ?        00:00:00 /usr/bin/kglobalaccel5
root       2754      2  0 20:59 ?        00:00:20 [kworker/u9:4+i915_flip]
root       2803      2  0 21:00 ?        00:00:00 [kworker/0:2-events]
gphani     2910   1790 80 21:01 ?        00:56:08 obs
root       2943      2  0 21:01 ?        00:00:00 [kworker/u8:3-i915]
gphani     2960   1548  0 21:02 ?        00:00:27 /usr/libexec/tracker-store
root       3534      2  0 21:06 ?        00:00:22 [kworker/1:2-events]
root       5089      2  0 21:18 ?        00:00:00 [kworker/u8:1-phy0]
root       6309      2  0 21:27 ?        00:00:00 [kworker/3:1-events]
root       6932      2  0 21:32 ?        00:00:10 [kworker/u9:1-uvcvideo]
gphani     7376   2910  0 21:35 ?        00:00:00 sh -c "/usr/lib/x86_64-linux
gphani     7378   7376  0 21:35 ?        00:00:07 /usr/lib/x86_64-linux-gnu/ob
gphani     7384   1548  0 21:35 ?        00:00:08 /usr/libexec/tracker-extract
root       8426      2  0 21:43 ?        00:00:00 [kworker/2:1-events]
root      10654      2  0 21:59 ?        00:00:00 [kworker/u8:0-i915]
root      11585      2  0 22:06 ?        00:00:00 [kworker/u8:2-events_unbound
root      11780      2  0 22:08 ?        00:00:00 [kworker/u9:0]
gphani    12073   2360  0 22:10 pts/0    00:00:00 xterm
gphani    12102  12073  0 22:10 pts/1    00:00:00 bash
gphani    12149  12102  0 22:11 pts/1    00:00:00 ps -ef
gphani@icme:~/scripts$
```



```
gphani@icme:~/scripts$ cat exec-example.sh
#!/bin/bash
echo PID of shell running this command:
echo $$
echo leaving bash behind and opening xterm if available
exec xterm
echo looks like xterm is not available or opening
gphani@icme:~/scripts$ echo $$
2360
gphani@icme:~/scripts$ ./exec-example.sh
PID of shell running this command:
12073
leaving bash behind and opening xterm if available
gphani@icme:~/scripts$ echo $$
2360
gphani@icme:~/scripts$ vi exec-example.sh
gphani@icme:~/scripts$ which xterm1
gphani@icme:~/scripts$ ./exec-example.sh
PID of shell running this command:
12392
leaving bash behind and opening xterm if available
./exec-example.sh: line 5: exec: xterm1: not found
gphani@icme:~/scripts$
```

Example here, I am showing the process ID of the shell. And then I am launching an xterm and then I am seeing whether the xterm would show me the process ID which is same or different from the parent bash shell. So, first of all, let me see what is the bash shell I am running it is 2360. Now I will execute this script.

So, now I have got the xterm that is available. And when I am launching it, you see that the program exec example was running with a process ID of 12073 and then it has launched the xterm. Now xterm would also, have its own process ID and you see that is 12102 and I see whether this parent shell 12073 is available.

And you see that it is showing as xterm, and it says that it has been launched by 2360 which is a bash. So, you see that it is looking as if from the bash I have directly launched xterm but what we have done is from the bash, we have launched the script exec example. And then it is a script which has launched the xterm.

But however, it shows as if the parent shell for x service 2360 which means that it is an exec example process which got replaced by xterm. So, you see that we are back to the parent cell with the PID 2360. And you could use the exec whenever you do not need the original script or shell that is launching it. And it is a good idea to also, have a comment to be displayed in case the launching has not been successful.

So, that is what we are doing here in case the xterm was not available. So, it would give a message now I will intentionally change the spelling. So, that I do not have this successfully running. So, which xterm1. So, there is nothing called like xterm1 is not there so, it should fail. So, you see that the failure has happened.