**Shell Programming**

**Shell Variables**

Shell variable are similar to other programming languages variable. They are

used to store and manipulate various types of data within a shell program and also set configuration options and customize the shell environment under Linux. The shell variable can hold values of any data type. If we assign a character value the variable become a character variable.

**Rules for Declare Shell Variable:**

1) The name of a variable can contain only letters ( a to z or A to Z), numbers ( 0 to

9) or the underscore character ( \_ ). a variable's name can start only with a letter

or an underscore. Example ali, token\_a, var\_1, var\_2 .

2) You cannot use other characters such as ! ,\*, or - is that these characters have a

special meaning for the shell. If you try to make a variable name with one of

these special characters it confuses the shell. Example 2\_var , -variable , var1-

var2 , var\_a!

3) The shell enables you to store any value you want in a variable.

4) Do not put spaces on both side of the assignment operator when assigning value

to variable.

5) Variables names are case-sensitive

6) You can define a NULL variable. Ex- v= & v=" "

Variables are defined with value or character: variable\_name=variable\_value

a=’A’ or name="Vidnyan college"

Values bounded by single quotes or double.

**No space before and after the assignment operator (=).**

a=10 Numaric Variable

The rules for naming variable are similar as “C programming languages”.

The values is stored in a variable is accessed by prefix with variable name as a doller ($) sign .

Example:

a=10.25 or name=“vidnyan mahavidyalaya sangola”

echo “The values of a is $a” echo $name

10.25

b=$a

echo “The value of b is $b”

10.25

The result of ls-l command is assigned to the variable a.

a=`ls-l`

echo $a

A command can be executed within another command by enclosing the letter in single quotes( ‘ ’).

Setting and Unsetting or deleting a variable tells the shell to remove the variable from the list of variables.

Syntax: set or unset variable\_name

Example: name=" vidnyan mahavidyalaya sangola " or set=”vms”

unset name

echo $name

Exporting a Bash Variable: $ export var2=linux

Variables in the Bourne shell do not have to be declared.

# This program display the date and time, current directory and its content.

echo “Todays Date is”

date

echo “Your Current Working Directory is”

pwd

echo “It contains”

**Execution**

The script can be executed by using the command sh. For example the pcc can be invoked by the following command:

$ sh program.sh

**Variable Types:**

When a shell is running, three main types of variables are present:

**Local Variables:** A local variable is a variable that is present within the

current instance of the shell. It is not available to programs that are started by

the shell. They are set at command prompt.

**Environment Variables:** An environment variable is a variable that is

available to any child process of the shell. Some programs need environment

variables in order to function correctly. Usually a shell script defines only those

environment variables that are needed by the programs that it runs.

 **Common Environment Variables**

|  |  |  |
| --- | --- | --- |
| **System**  **Variable** | **Meaning** | **To View Variable**  **Value Type** |
| BASH\_VERSION | Holds the version of this instance of  bash. | echo $BASH\_VERSION |
| HOSTNAME | The name of the your computer. | echo $HOSTNAME |
| CDPATH | The search path for the cd command. | echo $CDPATH |
| HOME | The home directory of the current user. | echo $HOME |
| PATH | The search path for commands. It is a colon-separated list of directories in which the shell looks for commands. | echo $PATH |
| PS1 | Your prompt settings | echo $PS1 |
| TERM | Your login terminal type. | echo $TERM |
| SHELL | Set path to login shell. | echo $SHELL |
| DISPLAY | Set X display name | echo $DISPLAY |
| EDITOR | Set name of default text editor. | Export  EDITOR=/usr/bin/vim |
| PWD | Your current working directory |  |

 **Shell Variables:** A shell variable is a special variable that is set by the shell

and is required by the shell in order to function correctly. Some of these

variables are environment variables whereas others are local variables.

**Metacharacters :-**

The command options, option arguments and command arguments are separated by the space character. However, we can also use special characters called metacharacters in a Linux command that the shell interprets rather than passing to the command.

**Type Metacharacters**

Filename Substitution ? \* [ . . . ] [ ! . . . ]

I/O Redirection > < >> << m> m> & n

Process Execution ; ( ) & && ||

Quoting Metacharaters \ “ ” ‘ ’ ` `

Positional Parameters $1 . . . . . . . . . . . $9

Special Characters $0 $\*s $@ $# $! $$ $-

|  |  |
| --- | --- |
| **Symbol** | **Meaning and Usage examples** |
| < | Redirection: Get input for the command. Example:  sort < filename.txt # print out the contents of filename.txt  With the lines sorted. |
| > | Redirection: Send the output of the command into the file.If the file does not exist, create it. If it does exist, overwrite it. Example:  cal > filename.txt # puts a current calendar into filename.txt |
| >> | Redirection: Send the output of the command to the end of the file. If the file does not exist, it will be created. If it does exist, it will be appended.  Example:  date >> filename.txt # This adds the current date and time to  the end of filename.txt |
| | | Pipe: This sends the output of the one command to the input of another  command. Example:  cat filename.txt | grep it # print the lines in filename.txt  that contain the string "it" |
| ? | Match any single character. Example:  ls e?.txt # list all files that start with e, end with .txt and have any 1 character in between (like e2.txt) |
| \* | Matches any number of any characters. Example:  ls e\*.txt # This will list all files that start with e, end with .txt and have any characters in between |
| [ ] | One of the characters within the square brackets must be matched.  Ex: ls -l e[abc].txt # This will give a long listing of ea.txt, eb.txt and/or ec.txt |
| $ | Denotes that a string is a variable name- Not used when assigning a  variable. Example:  my\_variable="this string" # Assigning the variable  echo $my\_variable # Referring to the contents of the variable  Also used in parameter substitution to check if a variable is defined and  determine a value. Example:  echo ${my\_variable-other} # Print the value of my\_variable if  defined, otherwise print "other" |
| \ | Escape - Ignore the shell's special meaning for the character after this  symbol. Treat it as though it is just an ordinary character. Example:  echo "I have \$300.00" # Print the $ instead of using it to look up a variable name.  Also used to put commands on multiple lines. Example:  cat filename.txt \ # Don't run yet...  date \ # Still don't do anything...  cal # No backslash on the last one, so run all the commands |
| ( ) | Grouping commands - If you want to run two commands and send their output to the same place, put them in a group together. Example:  (cal; date) > filename.txt # Put a calendar and the date in  filename.txt |
| { } | Used in special cases for variables with the $. One use is in parameter  substitution (see the $ definition, above), the other is in arrays. Example: numbers=( 1 2 3 4 5 )  echo ${numbers[1]} |
| " | Double Quoting - Used to group strings that contain spaces and other  special characters. Example:  my\_variable="test." # Assign a string to the variable |
| ' | Single Quoting- Used to prevent the shell from interpreting special  characters within the quoted string. Example:  my\_variable='backslash: \' # Assign the string to the variable |
| ` | Unquoting. Used within a quoted string to force the shell to interpret and run the command between the backticks. Example:  my\_variable="date is: `date`" # Store the string AND the output of the date command |
| && | Run the command to the right of the double-ampersand ONLY IF the  command on the left succeeded in running. Example:  mkdir stuff && echo "Made the directory" # Print a message on  success of the mkdir command |
| || | Run the command on the right of the double pipe ONLY IF the command on the left failed. Example:  mkdir stuff || echo "mkdir failed!" # Print a message on failure of the mkdir command |
| & | Run the process in the background, allowing you to continue your work on the command line. Example:  john /etc/passwd & # Try to crack the passwords - this takes a couple hours, so do it in the background.  Also used in redirection when copying one stream into the same location as another stream. Example:  cat filename.txt > file2.txt 2>&1 # Send Standard Error (2) to the file2.txt where Standard Output (1) is going |
| ; | Allows you to list multiple commands on a single line, separated by this  character. Example:  date;john passwd; date # Print the date, crack the passwords and print the date again afterwards - cheap benchmarking |
| = | Assignment. Set the variable named on the left to the value presented on  the right. Example: my\_variable="Hello World!" # no space between  the variable name and the string. |

**Shell Scripts**

Linux shells are all fully-featured programming languages as well as simple

ways of invoking programs. A file containing a number of shell commands is

called a shell script.. Shell scripts are simply a sequence of simple commands.

The shell incorporates a powerful programming feature. It facilitates all

programming constructs such as looping, conditional statements, variable declaration, function etc. A shell program is a series of LINUX commands that we have learned. We will combine these commands to perform complex task.

The very big advantage of shell scripts is that they provide a simple way of

automating a number of tasks or command otherwise manual tasks. If you need to

type a sequence of commands in more than twice, you probably want to write shell

script.

Linux do not need a special compiler to execute the shell scripts. These are

interpreted and executed by the shell itself. In shell script it is not necessary to give the name with extension .sh. But customarily name them .sh extension.

 **Importance of Shell Script-**

 Combine lengthy and repetitive sequences of commands into a single, simple

command.

 Generalize a sequence of operations on one set of data, into a procedure that can

be applied to any similar set of data.

 Create new commands using combinations of utilities in ways the original

authors never thought of.

 Simple shell scripts might be written as shell aliases, but the script can be made

available to all users and all processes. Shell aliases apply only to the current

shell.

 Wrap programs over which you have no control inside an environment that you

can control.

 Create customized datasets on the fly, and call applications to work on them, or

create customized application commands/procedures.

 Rapid prototyping.

 Shell scripts can take input from a user or file and output them to the screen.

 Whenever you find yourself doing the same task over and over again you

should use shell scripting, i.e., repetitive task automation.

o Creating your own power tools/utilities.

o Automating command input or entry.

o Customizing administrative tasks.

o Creating simple applications.

o Since scripts are well tested, the chances of errors are reduced while

configuring services or system administration tasks such as adding new

users.

 Easy to use.

 Quick start and interactive debugging.

 Time Saving.

 System Admin task automation.

 Shell scripts can execute without any additional effort on nearly any modern

Linux operating system as they are written an interpreted language.

 **Disadvantages**

 Compatibility problems between different platforms.

 Slow execution speed.

 A new process launched for almost every shell command executed.

Control and Loop structure

Control structure

Shell supports conditional statements which are used to perform different actions

based on different conditions. Following two decision making statements

 The if...else statements

 The case...esac statement

 **The if...else statements**

If else statements are useful decision making statements which can be used to

select an option from a given set of options. Shell supports following forms of if..else

statement −

 if...fi statement

 if...else...fi statement

 if...elif...else...fi statement

** The if...fi statement**

The if...fi statement is the fundamental control statement that allows Shell to

make decisions and execute statements conditionally.

Syntax: if [expression]

then

Statement to be executed if expression is true

Fi

Here Shell expression is evaluated then resulting value is true, given

statement(s) are executed. If expression is false then no statement would be not

executed. Most of the times you will use comparison operators while making decisions.

Give you attention on the spaces between braces and expression. This space is

mandatory otherwise you would get syntax error. If expression is a shell command

then it would be assumed true if it return 0 after its execution. If it is a Boolean

expression then it would be true if it returns true.

Example of if…then…fi statement

**a=10**

**b=20**

**if [ $a == $b ]**

**then**

**echo "a is equal to b"**

**fi**

**if [ $a != $b ]**

**then**

**echo "a is not equal to b"**

**fi**

**Output: a is not equal to b**

** The if...else...fi statement**

The if...else...fi statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

**Syntax: if [expression]**

**then**

**Statement to be executed if expression is true**

**else**

**Statement to be executed if expression is not true**

**Fi**

Here Shell expression is evaluated. If the resulting value is true, given statement(s) are executed. If expression is false then no statement would be not executed.

Example of if...else statement

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

else

echo "a is not equal to b"

fi

Output − a is not equal to b

** The if...elif...fi statement**

The if...elif...fi statement is the one level advance form of control statement

that allows Shell to make correct decision out of several conditions.

Syntax: if [expression 1]

then

Statement to be executed if expression 1 is true

elif [ expression 2 ]

then

Statement to be executed if expression 2 is true

elif [ expression 3 ]

then

Statement to be executed if expression 3 is true

else

Statement to be executed if no expression is true

Fi

A series of if statements, where each if is part of the else clause of the previous

statement. Here statement(s) are executed based on the true condition, if none of the condition is true then else block is executed.

Example of if...elif...fi statement

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

elif [ $a -gt $b ]

then

echo "a is greater than b"

elif [ $a -lt $b ]

then

echo "a is less than b"

else

echo "None of the condition met"

fi

Output − a is less than b

 **The case...esac statement**

Shell supports case...esac statement which handles exactly this situation, and it

does so more efficiently than repeated if...elif statements.

The basic syntax of the case...esac statement is to give an expression to

evaluate and several different statements to execute based on the value of the

expression. The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

Syntax:

case $(Expression variable) in

**<**pattern> 1)

Statement(s) to be executed if pattern1 matches

;;

<pattern> 2)

Statement(s) to be executed if pattern2 matches

;;

<pattern> 3)

Statement(s) to be executed if pattern3 matches

;;

<defualt> \*) Statement

;;

Esac

Here the string Expression is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action. There is no maximum number of patterns, but the minimum is one. When statement(s) part executes, the command ;; indicates that program flow should jump to the end of the entire case statement.

Examples of case...esac statement

FRUIT="kiwi"

case "$FRUIT" in

"apple") echo "Apple pie is quite tasty."

;;

"banana") echo "I like banana nut bread."

;;

"kiwi") echo "New Zealand is famous for kiwi."

;;

Esac

Example

echo “Enter Your choice: \c”

echo “1. List of files”

echo “2. Processes of user”

echo “3. Today’s date”

echo “4.Users of system”

echo “5.Quit to LINUX”

read choice

case “$choice” in

ls –l ;;

ps –f ;;

date ;;

who ;;

exit ;;

esac

**# Write a shell script program to check input character is digit, character of special symbol**.

echo “Enter any character :”

read ch

case $ch in

[a-z]) echo “The character is in lower case”

;;

[A-Z]) echo “The character is in UPPER case”

;;

[0-9]) echo “The character is in digit”

;;

\*) echo “The character is in Special Character”

;;

esac

Output: Enter any character:

A

The character is in UPPER case

Enter any character:

a

The character is in lower case

Enter any character:

5

The character is in digit

Enter any character:

^

The character is in Special Character

**Shell Loop Types**

Loops are a powerful programming tool that enables you to execute a set of commands repeatedly

 The while loop

 The for loop

 The until loop

You would use different loops based on different situation

** The while Statement**

Loops let you perform a set of instructions repeatedly. The shell features three types of loops – while, until and for. The first two are complementary to each other. All of them repeat the instructions set enclosed by certain keywords as often as the control command permits.

The while statement should be quite similar to most programmers. It repeatedly

performs a set of instructions till the control command returns a true status.

**syntax: while <condition is true>**

do

<execute commands>

done

The do and done are keywords. The set of instructions enclosed by do and done are to be performed as long as the condition remains true. This condition is actually the

return value of a LINUX command or program. This means that you can use the test statement here also, with its associated expression, numeric and string comparisons and file.

**# write a shell program to print the first 10 numbers.**

echo “ ”

i=1

while [ $i –le 10 ]

do

echo “Number $i”

i= ‘expr $i + 1’

done

echo “”

Output: 1 2 3 4 5 6 7 8 9 10

**# Write a shell script program to reverse the given number.**

echo “Enter any number :-”

echo “ ”

read no

sum=0

while [ $no –ne 0 ]

do

rem=’expr $no % 10’

sum=’expr $sum \\* 10 + $rem’

no=’expr $no / 10’

done

echo “ ”

echo “Reverse number is = $sum”

Output: Enter any number:- 123

Reverse number is = 6

# Write a shell script program to print the factorial of the given

number.

echo “ ”

echo “Enter the number”

read num

fact=1

while test $num –ne 0

do

fact=’expr $fact \\* $num’

num=’expr $num – 1’

done

echo “Factorial = $fact”

Output: Enter the Number

5

Factorial = 120

# Write a shell script program to print Fibonacci series of a given number.

echo “ ”

a=0

b=1

echo “$a”

echo “$b”

echo “Enter the Number :”

read n

while [ $c –le n ]

do

c=’expr $a + $b’

echo “$c”

done

Output: Enter the Number: 5

1

1

2

3

5

8

** The until Statement**

The until statement complements the while statement in the sense that the loop

body here is executed repeatedly as long as the condition remains false. it is simply

another way of viewing the whole thing. Either can be used interchangeably, except for the fact that the expression has to be invalid switching from one to another.

Syntax: until [ condition false ]

do

<execute commands>

done

**# Write a shell script to print first 10 nos.**

echo “ ”

i=1

until [ $i –gt 10 ]

do

echo “Number $i”

i= ‘expr $i + 1’

done

echo “ ”

Output: 1 2 3 4 5 6 7 8 9 10

**. Nesting Loops**

All the loops support nesting concept which means you can put one loop inside another similar or different loops. This nesting can go up to unlimited number of times based on your requirement.

***Syntax:***

while [condition] # this is loop1, the outer loop  
do  
Statement to be executed if command1 is true  
while command2 ; # this is loop2, the inner loop  
do  
Statement to be executed if command2 is true  
done  
Statement to be executed if command1 is true  
done

***Example:***

a=0  
while [ "$a" -lt 10 ] # this is loop1  
do  
b="$a"  
while [ "$b" -ge 0 ] # this is loop2  
do  
echo -n "$b "  
b=`expr $b - 1`  
done  
echo  
a=`expr $a + 1`  
done

Here -n option let echo to avoid printing a new line character.

0  
1 0  
2 1 0  
3 2 1 0  
4 3 2 1 0  
5 4 3 2 1 0  
6 5 4 3 2 1 0  
7 6 5 4 3 2 1 0  
8 7 6 5 4 3 2 1 0  
9 8 7 6 5 4 3 2 1 0

**Looping with for:**

The for….loop is different in structure from the ones used in other programming languages. There is no next statement here, neither can a step be specified. Unlike while and until, it doesn’t test a condition but uses a list instead. The syntax of this construct is as follows:

***Syntax:***

for variable in list  
do  
<execute commands>  
done

The loop body is identical in structure to the while and until loops. The additional keywords here are variable and list. The list consists of a series of character strings, with each string separated from the other by white space. Each item in the list is assigned to the variable in turn and the loop body is executed. It performs the loop as many times as there are words in the list.

$ for x in 1 2 4 5 or $ for x in {1..5}  
do  
echo “The value of x is $x”  
done  
**Output**: The value of x is 1  
The value of x is 2  
The value of x is 4  
The value of x is 5

**# Write a shell script program to print the series.**

echo “Enter the Range : ”  
read n  
k=1  
for((i=0;i<=$n;j++))  
do  
for((j=1;j<i;j++))  
do  
echo –n “$k”  
k=’expr $k + 1’  
done  
echo \  
done  
Output: Enter the range : 5  
1  
2 3  
4 5 6  
7 8 9 10

**# Write a shell script program to print the series.**

echo “Enter the Range”  
read n  
echo “ ”  
for((i=0;i<=$n;j++))  
do  
for((j=1;j<i;j++))  
do  
echo –n “$i”  
done  
echo \  
echo \  
done

Output: Enter the range  
5

1  
2 2  
3 3 3  
4 4 4 4  
5 5 5 5 5

**The break statement**

The break statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Syntax: break

The break command can also be used to exit from a nested loop using this format -break n. Here n specifies the nth enclosing loop to exit from.

Example: which shows that loop would terminate as soon as a becomes 5:

a=0  
while [ $a -lt 10 ]  
do  
echo $a  
if [ $a -eq 3 ]  
then  
break  
fi  
a=`expr $a + 1`  
done

Output-

0  
1

2

3

example of nested for loop. This script breaks out of both loops if var1 equals 2 and var2 equals 0 –

for var1 in 1 2 3  
do  
for var2 in 0 5  
do  
if [ $var1 -eq 2 -a $var2 -eq 0 ]  
then  
break 2  
else  
echo "$var1 $var2"  
fi  
done  
done

Output –

1 0  
1 5

Inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop & from inner loop as well.

**The continue statement**

The continue statement is similar to the break command, except that it causes the  
current iteration of the loop to exit, rather than the entire loop.  
This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

Syntax: continue

Like with the break statement, an integer argument can be given to the continue  
command to skip commands from nested loops.

continue n

Here n specifies the nth enclosing loop to continue from.

***Example:***

NUMS="1 2 3 4 5 6 7"  
for NUM in $NUMS  
do  
Q=`expr $NUM % 2`  
if [ $Q -eq 0 ]  
then  
echo "Number is an even number!!"  
continue  
fi  
echo "Found odd number"  
done  
Output - Found odd number  
Number is an even number!!  
Found odd number  
Number is an even number!!  
Found odd number  
Number is an even number!!  
Found odd number

**I/O and Redirection:-**

Linux provides the capability to change where standard input comes from, or where output goes using a concept called Input/Output (I/O) redirection. I/O redirection is accomplished using a redirection operator which allows the user to specify the input or output data be redirected to (or from) a file. Note that redirection always results in the data stream going to or coming from a file.

The simplest case to demonstrate this is basic output redirection. The output redirection operator is the > (greater than) symbol, and the general syntax looks as follows:

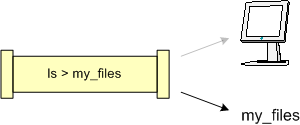
command > output\_file\_spec

Spaces around the redirection operator are not mandatory, but do add readability to the command. Thus in our ls example from above, we can observe the following use of output redirection:

$ ls > my\_files [Enter]

$

Notice there is no output appearing after the command, only the return of the prompt. Why is this, you ask? This is because all output from this command was redirected to the file my\_files. Observe in the following diagram, no data goes to the terminal screen, but to the file instead.



Examining the file as follows results in the contents of the my\_files being displayed:

$ cat my\_files [Enter]

foo  
 bar  
 fred  
 barney  
 dino

$

In this example, if the file my\_files does not exist, the redirection operator causes its creation, and if it does exist, the contents are overwritten. Consider the example below:

$ echo "Hello World!" > my\_files [Enter]

$ cat my\_files [Enter]

Hello World!

Notice here that the previous contents of the my\_files file are gone, and replaced with the string "Hello World!" This might not be the most desirable behavior, so the shell provides us with the capability to append output to files. The append operator is the >>. Thus we can do the following:

$ ls > my\_files [Enter]

$ echo "Hello World!" >> my\_files [Enter]

$ cat my\_files [Enter]

foo  
 bar  
 fred  
 barney  
 dino  
 Hello World!

The first output redirection creates the file if it does not exist, or overwrites its contents if it does, and the second redirection appends the string "Hello World!" to the end of the file. When using the append redirection operator, if the file does not exist, >> will cause its creation and append the output (to the empty file). The ability also exists to redirect the standard input using the input redirection operator, the < (less than) symbol. Note the point of the operator implies the direction. The general syntax of input redirection looks as follows:

command < input\_file\_spec

### Redirection Summary

|  |  |
| --- | --- |
| **Redirection Operator** | **Resulting Operation** |
| command > file | stdout written to file, overwriting if file exists |
| command >> file | stdout written to file, appending if file exists |
| command < file | input read from file |
| command 2> file | stderr written to file, overwriting if file exsits |
| command 2>> file | stderr written to file, appending if file exists |
| command > file 2>&1 | stdout written to file, stderr written to same file descriptor |

**Piping**

The shell uses a pipe to connect standard output of one command to standard input of another command. A pipe (pipeline) has the same effect as redirecting standard output of one command to a file and then using that file as standard input to another command. The pipe is represented by the vertical line character (|).The pipe operation receives output from the command placed before the pipe and sends this data as input to the command placed after the pipe. Filters are often used with pipes.

Piping

***Syntax:* $ command1 | command2**

Suppose you want to send a list of your filenames to the printer to be printed.

You need two commands to do this:

1) The ls command to generate a list of filenames.

2) The lpr command to send the list to the printer.

You need to take the output of the ls command and use it as input for the lpr

command. The pipe operator (|) placed between two commands forms a connection

between them

***Example:* # ls | lpr**

The following command line sorts the lines of the months file and uses head to display the first four months of the sorted list:

**# sort months | head -3**

Apr

Aug

Dec

Why use piping

1.The process is slow. The later command cannot get executed if the earlier ones are not yet executed.

2. An intermediate file is required that has to be removed after the wc command has been executed.

3. When handling large files, temporary files can built up easily and eat up the disk

space.