NDG Introduction to Linux I - Chapter 3: Configuring the Shell

Objectives

Chapter 3: Configuring the Shell

This chapter will cover the following exam objectives:

103.1: Work on the command line v2

Weight: 4

Candidates should be able to interact with shells and commands using the command line. The objective assumes the Bash shell.

Key Knowledge Areas:

Use and edit command history.
Section 3.4 | Section 3.4.1

Invoke commands inside and outside the defined path.
Section 3.2.4

Use and modify the shell environment including defining, referencing and exporting environment variables.
Section 3.2.1 | Section 3.2.2 | Section 3.2.3

Key Terms

.bash_history
File used to store the current history list when the shell is closed.
Section 3.4

env
Print a list of the current Environment variables or change to an alternate environment.

export
Makes an assigned variable available to sub-processes.

history

Print a list of previously executed commands or "re-execute" previously executed commands.

set

Display all variables (local and environment).

unset

Remove one or more variables.

Content

3.1 Introduction

One key component of the Bash shell is shell variables. These variables are critical because they store vital system information and modify the behavior of the Bash shell, as well as many commands. This chapter will discuss in detail what shell variables are and how they can be used to configure the shell.

You will learn how the PATH variable affects how commands are executed and how other variables affect your ability to use the history of your commands. You will also see how to use initialization files to make shell variables persistent, so they will be created each time you log into the system.

Additionally, this chapter will cover how to use the command history and the files that are used to store and configure the command history.

3.2 Shell Variables

A variable is a name or identifier that can be assigned a value. The shell and other commands read the values of these variables, which can result in altered behavior depending on the contents (value) of the variable.

Variables typically hold a single value, like 0 or bob. Some may contain multiple values separated by spaces like Joe Brown or by other characters, such as colons; /usr/bin:/usr/sbin:/bin:/usr/bin:/home/joe/bin.

You assign a value to a variable by typing the name of the variable immediately followed by the equal sign = character and then the value. For example:

```
name="Bob Smith"
```

Variable names should start with a letter (alpha character) or underscore and contain only letters, numbers and the underscore character. It is important to remember that variable names are case sensitive; a and A are different variables. Just as with arguments to commands, single or double quotes should be used when special characters are included in the value assigned to the variable to prevent shell expansion.

Valid Variable Assignments	Invalid Variable Assignments
`a=1`	`1=a`
`_1=a`	`a-1=3`
`LONG_VARIABLE='OK'`	`LONG-VARIABLE='WRONG'`
`Name='Jose Romero'`	`'user name'=anything`

The Bash shell and many commands make extensive use of variables. One of the main uses of variables is to provide a means to configure various preferences for each user. The Bash shell and commands can behave in different ways, based on the value of variables. For the shell, variables can affect what the prompt displays, the directories the shell will search for commands to execute and much more.

3.2.1 Local and Environment Variables

A local variable is only available to the shell in which it was created. An environment variable is available to the shell in which it was created, and it is passed into all other commands/programs started by the shell.

To set the value of a variable, use the following assignment expression. If the variable already exists, the value of the variable is modified. If the variable name does not already exist, the shell creates a new local variable and sets the value:

```
variable=value
```

In the example below, a local variable is created, and the echo command is used to display its value:

```
sysadmin@localhost:~$ name="judy"
sysadmin@localhost:~$ echo $name
judy
```

By convention, lowercase characters are used to create local variable names, and uppercase characters are used when naming an environment variable. For example, a local variable might be called test while an environment variable might be called TEST. While this is a convention that most people follow, it is not a rule. An environment variable can be created directly by using the export command.

```
export variable=value
```

In the example below, an environment variable is created with the export command:

```
sysadmin@localhost:~$ export JOB=engineer
```

Recall that the echo command is used to display output in the terminal. To display the value of the variable, use a dollar sign $ character followed by the variable name as an argument to the echo command:

```
sysadmin@localhost:~$ echo $JOB
engineer
```

We can see the difference between local and environment variables by opening a new shell with the bash command:

```
sysadmin@localhost:~$ bash
To run a command as administrator (user "root"), use "sudo" command.
See "man sudo_root" for details.
sysadmin@localhost:~$ echo $name
sysadmin@localhost:~$ echo $JOB
engineer
```

Notice that the local variable name is empty, while the environment variable JOB returns the value engineer. Also notice that if you return to the first shell by using the exit command, both variables are still available in the original shell:

```
sysadmin@localhost:~$ exit
exit
sysadmin@localhost:~$ echo $name
judy
sysadmin@localhost:~$ echo $JOB
engineer
```

The local variable name is not available in the new shell because by default, when a variable is assigned in the Bash shell, it is initially set as a local variable. When you exit the original shell, only the environment variables will be available. There are several ways that a local variable can be made into an environment variable.

First, an existing local variable can be exported with the export command.

```
export variable
```

In the example below, the local variable name is exported to the environment (with the standard convention of all caps):

```
sysadmin@localhost:~$ NAME=judy
sysadmin@localhost:~$ export NAME
sysadmin@localhost:~$ echo $NAME
judy
```

Second, a new variable can be exported and assigned a value with a single command as demonstrated below with the variable DEPARTMENT:

```
sysadmin@localhost:~$ export DEPARTMENT=science
sysadmin@localhost:~$ echo $DEPARTMENT
science
```

Third, the `declare` or `typeset` command can be used with the export `-x` option to declare a variable to be an environment variable. These commands are synonymous and work the same way:

```
sysadmin@localhost:~$ declare -x EDUCATION=masters
sysadmin@localhost:~$ echo $EDUCATION
masters
sysadmin@localhost:~$ typeset -x EDUCATION=masters
sysadmin@localhost:~$ echo $EDUCATION
masters
```

The `env` command is used to run commands in a modified environment. It can also be used to temporarily create or change environment variables that are only passed to a single command execution by using the following syntax:

```
env [NAME=VALUE] [COMMAND]
```

For example, servers are often set to Coordinated Universal Time (UTC), which is good for maintaining consistent time on servers across the planet, but can be frustrating for practical use to simply tell the time:

```
sysadmin@localhost:~$ date
Sun Mar 10 22:47:44 UTC 2020
```

To temporarily set the time zone variable, use the `env` command. The following will run the `date` command with the temporary variable assignment:

```
sysadmin@localhost:~$ env TZ=EST date
```

```
Sun Mar 10 17:48:16 EST 2020
```

The TZ variable is set only in the environment of the current shell, and only for the duration of the command. The rest of the system will not be affected by this variable. In fact, running the date command again will verify that the TZ variable has reverted to UTC.

```
sysadmin@localhost:~$ date
Sun Mar 10 22:49:46 UTC 2020
```

3.2.2 Displaying Variables

There are several ways to display the values of variables. The set command by itself will display all variables (local and environment). Here, we will pipe the output to the tail command so we can see some of the variables that were set in the previous section:

```
sysadmin@localhost:~$ set | tail
DEPARTMENT=science
SHELL=/bin/bash
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
SHLVL=1
TERM=xterm
UID=1001
USER=sysadmin
VISUAL=vi
_=set
name=judy
```

Note

The example above utilizes a command line pipe, represented by the | character. The pipe character can be used to send the output of one command to another.

The command line pipes will be covered in greater detail later in the course.

Note

The tail command is used to display only the last few lines of a file, (or, when used with a pipe, the output of a previous command). By default, the tail command displays ten lines of a file provided as an argument.

The tail command will be covered in greater detail later in the course.

To display only environment variables, you can use several commands that provide nearly the same output:

```
env

declare -x

typeset -x

export -p
```

```
sysadmin@localhost:~$ env | tail

NAME=judy

MAIL=/var/mail/sysadmin

SHELL=/bin/bash

TERM=xterm

SHLVL=1

EDUCATION=masters

LOGNAME=sysadmin

PATH=/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:

/bin:/usr/games:/usr/local/games

LESSOPEN=| /usr/bin/lesspipe %s

_=/usr/bin/env
```

To display the value of a specific variable, use the echo command with the name of the variable prefixed by the $ (dollar sign). For example, to display the value of the PATH variable, you would execute echo $PATH:

```
sysadmin@localhost:~$ echo $PATH

/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

Consider This

Variables can also be enclosed in the curly brace {} characters in order to delimit them from surrounding text. While the echo ${PATH} command would produce the same result as the echo $PATH command, the curly braces set the variable apart visually, making it easier to see in scripts in some contexts.

3.2.3 Unsetting Variables

If you create a variable and then no longer want that variable to be defined, use the `unset` command to delete it:

```
unset VARIABLE
```

```
sysadmin@localhost:~$ example=12
sysadmin@localhost:~$ echo $example
12
sysadmin@localhost:~$ unset example
sysadmin@localhost:~$ echo $example
```

Warning

Do not unset critical system variables like the PATH variable, as this may lead to a malfunctioning environment.

3.2.4 PATH Variable

The PATH variable is one of the most critical environment variables for the shell, so it is important to understand the effect it has on how commands will be executed.

The PATH variable contains a list of directories that are used to search for commands entered by the user. When the user types a command and then presses the **Enter** key, the PATH directories are searched for an executable file that matches the command name. Processing works through the list of directories from left to right; the first executable file that matches what is typed is the command the shell will try to execute.

Note

Before searching the PATH variable for the command, the shell will first determine if the command is an alias or function, which may result in the PATH variable not being utilized when that specific command is executed.

Additionally, if the command happens to be built-in to the shell, the PATH variable will not be utilized.

Using the echo command to display the current $PATH will return all the directories that files can be executed from. The following example displays a typical PATH variable, with directory names separated from each other by a colon : character:

```
sysadmin@localhost:~$ echo $PATH

/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

The following table illustrates the purpose of some of the directories displayed in the output of the previous command:

Directory	Contents
`/home/sysadmin/bin`	A directory for the current user sysadmin to place programs. Typically used by users who create their own scripts.
`/usr/local/sbin`	Normally empty, but may have administrative commands that have been compiled from local sources.
`/usr/local/bin`	Normally empty, but may have commands that have been compiled from local sources.
`/usr/sbin`	Contains the majority of the administrative command files.
`/usr/bin`	Contains the majority of the commands that are available for regular users to execute.
`/sbin`	Contains the essential administrative commands.
`/bin`	Contains the most fundamental commands that are essential for the operating system to function.

To execute commands that are not contained in the directories that are listed in the PATH variable, several options exist:

- The command may be executed by typing the absolute path to the command.
- The command may be executed with a relative path to the command.
- The PATH variable can be set to include the directory where the command is located.
- The command can be copied to a directory that is listed in the PATH variable.

To demonstrate absolute paths and relative paths an executable script named my.sh is created in the home directory. Next, that script file is given "execute permissions" (permissions are covered in a later chapter):

```
sysadmin@localhost:~$ echo 'echo Hello World!' > my.sh
sysadmin@localhost:~$ chmod u+x my.sh
```

An absolute path specifies the location of a file or directory from the top-level directory through all of the subdirectories to the file or directory. Absolute paths always start with the / character representing the root directory. For example, /usr/bin/ls is an absolute path. It means the ls file, which is in the bin directory, which is in the usr directory, which is in the / (root) directory. A file can be executed using an absolute path like so:

```
sysadmin@localhost:~$ /home/sysadmin/my.sh
Hello World!
```

A relative path specifies the location of a file or directory relative to the current directory. For example, in the /home/sysadmin directory, a relative path of test/newfile would actually refer to the /home/sysadmin/test/newfile file. Relative paths never start with the / character.

Note

Visualizing the directory structure using paths can be confusing at first, but it is a necessary skill for effectively navigating the filesystem.

Absolute paths and relative paths are covered in greater detail in the **NDG Linux Essentials** course.

Using a relative path to execute a file in the current directory requires the use of the . character, which symbolizes the current directory:

```
sysadmin@localhost:~$ ./my.sh
Hello World!
```

Sometimes a user wants their home directory added to the PATH variable in order to run scripts and programs without using ./ in front of the file name. They might be tempted to modify the PATH variable like so:

```
sysadmin@localhost:~$ echo $PATH
/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

```
sysadmin@localhost:~$ pwd
/home/sysadmin
sysadmin@localhost:~$ PATH=/home/sysadmin
```

Unfortunately, modifying a variable this way overwrites the contents. Therefore, everything that was previously contained in the PATH variable will be lost.

```
sysadmin@localhost:~$ echo $PATH
/home/sysadmin
```

Programs listed outside of the /home/sysadmin directory will now only be accessible by using their full path name. For example, assuming that the home directory has not yet been added to the PATH variable, the `uname -a` command would behave as expected:

```
sysadmin@localhost:~$ uname -a
Linux localhost 3.15.6+ #2 SMP Wed Jul 23 01:26:02 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

After assigning the home directory to the PATH variable, the `uname -a` command will need its full path name to execute successfully:

```
sysadmin@localhost:~$ PATH=/home/sysadmin
sysadmin@localhost:~$ uname -a
-bash: uname: command not found
sysadmin@localhost:~$ /bin/uname -a
Linux localhost 3.15.6+ #2 SMP Wed Jul 23 01:26:02 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

Important

If you changed the PATH variable like we did in the previous example and want to reset it, simply use the `exit` command to logout:

```
sysadmin@localhost:~$ exit
```

Once logged back in, the PATH variable will be reset to its original value.

It is possible to add a new directory to the PATH variable without overwriting its previous contents. Import the current value of the $PATH variable into the newly defined PATH variable by using it on both sides of the assignment statement:

```
sysadmin@localhost:~$ PATH=$PATH
```

```
sysadmin@localhost:~$ PATH=$PATH:/home/sysadmin

sysadmin@localhost:~$ echo $PATH

/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games:/usr/local/games:/home/sysadmin
```

Now, scripts located in the /home/sysadmin directory can execute without using a path:

```
sysadmin@localhost:~$ my.sh

Hello World!
```

Warning

In general, it is a bad idea to modify the $PATH variable. If it were to change, administrators would view it as suspicious activity. Malicious forces want to gain elevated privileges and access to sensitive information residing on Linux servers. One way to do this is to write a script which shares the name of a system command, then change the PATH variable to include the administrator's home directory. When the administrator types in the command, it actually runs the malicious script!

3.3 Initialization Files

When a user opens a new shell, either during login or when they run a terminal that starts a shell, the shell is customized by files called initialization (or configuration) files. These initialization files set the value of variables, create aliases and functions, and execute other commands that are useful in starting the shell.

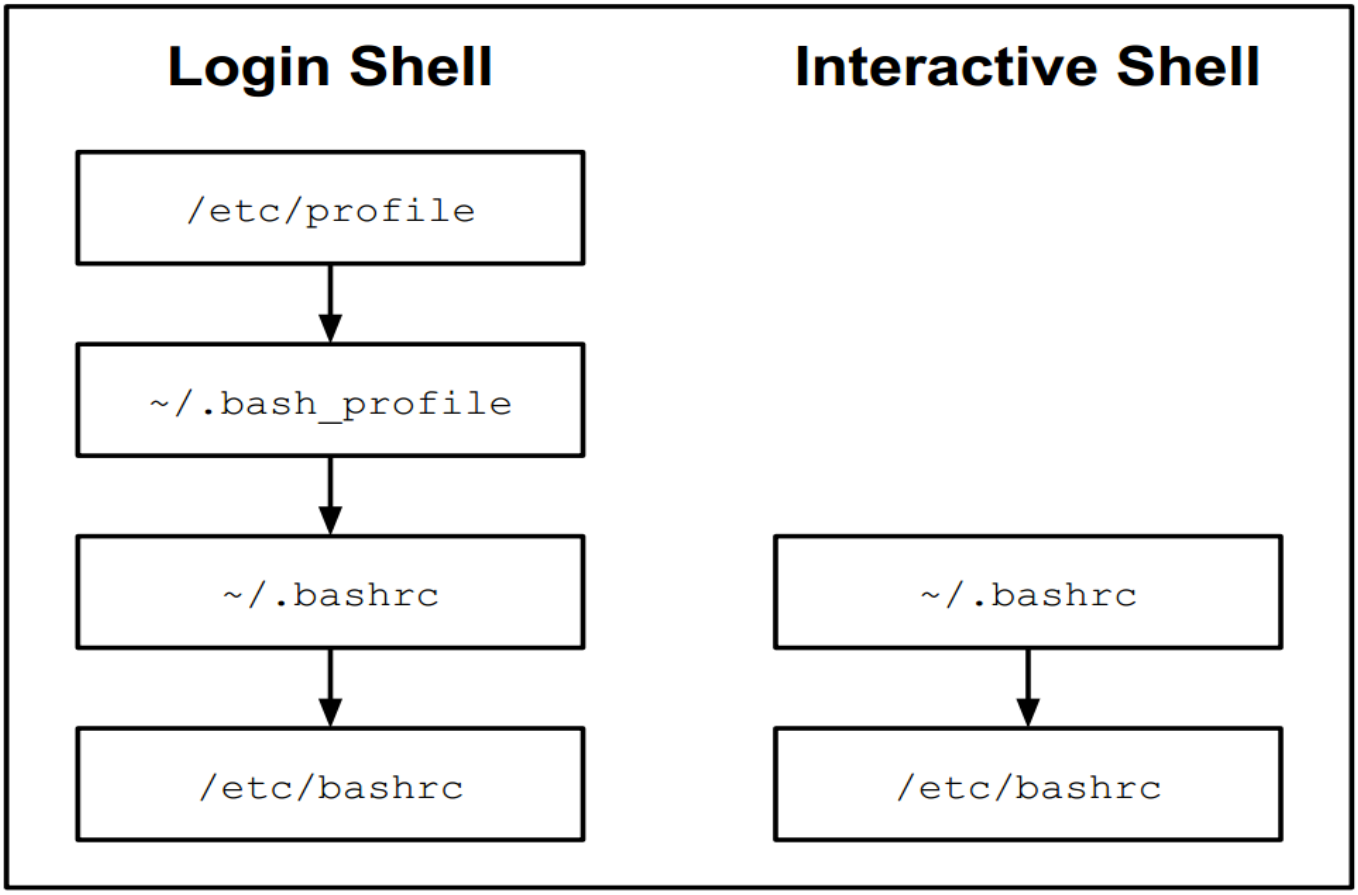
There are two types of initialization files: global initialization files that affect all users on the system and local initialization files that are specific to an individual user.

The global configuration files are located in the /etc directory. Local configuration files are stored in the user's home directory.

BASH Initialization Files

Each shell uses different initialization files. Additionally, most shells execute different initialization files when the shell is started via the login process (called a login shell) versus when a shell is started by a terminal (called a non-login shell or an interactive shell).

The following diagram illustrates the different files that are started with a typical login shell versus an interactive shell:



Note

The tilde ~ character represents the user's home directory.

Note

File names preceded by a period . character indicate hidden files. You can view these files in your home directory by using the ls command with the all -a option.

```
sysadmin@localhost:~$ ls -a
.               .bashrc   .selected_editor  Downloads  Public
..               .cache Desktop         Music   Templates
.bash_logout   .profile  Documents       Pictures   Videos
```

When Bash is started as a login shell, the `/etc/profile` file is executed first. This file typically executes all files ending in `.sh` that are found in the `/etc/profile.d` directory.

The next file that is executed is usually `~/.bash_profile` (but some users may use the `~/.bash_login` or `~/.profile` file). The `~/.bash_profile` file typically also executes the `~/.bashrc` file which in turn executes the `/etc/bashrc` file.

Consider This

Since the ~/.bash_profile file is under the control of the user, the execution of the ~/.bashrc and /etc/bashrc files are also controllable by the user.

When Bash is started as an interactive shell, it executes the ~/.bashrc file, which may also execute the /etc/bashrc file, if it exists. Again, since the ~/.bashrc file is owned by the user who is logging in, the user can prevent execution of the /etc/bashrc file.

With so many initialization files, a common question at this point is "what file am I supposed to use?" The following chart illustrates the purpose of each of these files, providing examples of what commands you might place in each file:

File	Purpose
`/etc/profile`	This file can only be modified by the administrator and will be executed by every user who logs in. Administrators use this file to create key environment variables, display messages to users as they log in, and set key system values.

`~/.bash_profile`
`~/.bash_login`
 	
`~/.profile`
	Each user has their own .bash_profile file in their home directory. The purpose of this file is the same as the /etc/profile file, but having this file allows a user to customize the shell to their own tastes. This file is typically used to create customized environment variables.

`~/.bashrc`	Each user has their own .bashrc file in their home directory. The purpose of this file is to generate items that need to be created for each shell, such as local variables and aliases.
`/etc/bashrc`	This file may affect every user on the system. Only the administrator can modify this file. Like the .bashrc file, the purpose of this file is to generate items that need to be created for each shell, such as local variables and aliases.

3.3.1 Modifying Initialization Files

The way a user's shell operates can be changed by modifying that user's initialization files. Modifying global configuration requires administrative access to the system as the files under the /etc directory can only be modified by an administrator. A user can only modify the initialization files in their home directory.

The best practice is to create a backup before modifying configuration files as insurance against errors; a backup can always be restored if something should go wrong. The following example references CentOS, a community-supported Linux distribution based on source code from Red Hat Enterprise Linux.

In some distributions, the default ~/.bash_profile file contains the following two lines which customize the PATH environment variable:

```
PATH=$PATH:$HOME/bin

export PATH
```

The first line sets the PATH variable to the existing value of the PATH variable with the addition of the bin subdirectory of the user's home directory. The $HOME variable refers to the user's home directory. For example, if the user logging in is joe,
then $HOME/bin is /home/joe/bin.

The second line converts the local PATH variable into an environment variable.

The default ~/.bashrc file executes /etc/bashrc using a statement like:

```
.  /etc/bashrc
```

The period . character is used to source a file in order to execute it. The period character also has a synonym command, the source command, but the use of the period character is more common than using the source command.

Sourcing can be an effective way to test changes made to initialization files, enabling you to correct any errors without having to log out and log in again. If you update
the ~/.bash_profile file to alter the PATH variable, you could verify that your changes are correct by executing the following:

```
. ~/.bash_profile

echo $PATH
```

Warning

Our virtual environment is not persistent, therefore, any changes made to the configuration files will not be saved if the VM is reset.

3.3.2 BASH Exit Scripts

Just as Bash executes one or more files upon starting up, it also may execute one or more files upon exiting. As Bash exits, it will execute the ~/.bash_logout and /etc/bash_logout files, if they exist. Typically, these files are used for "cleaning up" tactics as the user exits the shell. For example, the default ~/.bash_logout file executes the clear command to remove any text present in the terminal screen.

Consider This

When a new user is created, the files from the /etc/skel directory are automatically copied into the new user's home directory. As an administrator, you can modify the files in the/etc/skel directory to provide custom features to new users.

Just as Bash executes one or more files upon starting up, it also may execute one or more files upon exiting. As Bash exits, it will execute the ~/.bash_logout and /etc/bash_logout files, if they exist. Typically, these files are used for "cleaning up" tactics as the user exits the shell. For example, the default ~/.bash_logout executes the clear command to remove any text present in the terminal screen.

Consider This

When a new user is created, the files from the /etc/skel directory are automatically copied into the new user's home directory. As an administrator, you can modify the files in the/etc/skel directory to provide custom features to new users.

3.4 Command History

In a sense, the ~/.bash_history file could also be considered to be an initialization file, since Bash also reads this file as it starts up. By default, this file contains a history of the commands that a user has executed within the Bash shell. When a user exits the Bash shell, it writes out the recent history to this file.

There are several ways that this command history is advantageous to the user:

- You can use the **Up ↑** and **Down ↓ Arrow Keys** to review your history and select a previous command to execute again.
- You can select a previous command and modify it before executing it.
- You can do a reverse search through history to find a previous command to select, edit, and execute it. To start the search, press **Ctrl+R** and then begin typing a portion of a previous command.
- You execute a command again, based upon a number that is associated with the command.

3.4.1 Executing Previous Commands

The Bash shell will allow a user to use the **Up ↑ Arrow Key** to view previous commands. With each press of the up arrow, the shell displays one more command back in the history list.

If the user goes back too far, the **Down ↓ Arrow Key** can be used to return to a more recent command. Once the correct command is displayed, the **Enter** key can be pressed to execute it.

The **Left ←** and **Right → Arrow Keys** can also be used to position the cursor within the command. The **Backspace** and **Delete** keys are used to remove text, and additional characters can be typed into the command line to modify it before pressing the **Enter** key to execute it.

There are more keys that can be used for editing a command on the command line. The following table summarizes some helpful editing keys:

Action	Key	Alternate Key Combination
Previous history item	↑	**Ctrl+P**
Next history item	↓	**Ctrl+N**
Reverse history search		**Ctrl+R**
Beginning of line	**Home**	**Ctrl+A**
End of line	**End**	**Ctrl+E**
Delete current character	**Delete**	**Ctrl+D**
Delete to the left of the cursor	**Backspace**	**Ctrl+X**
Move cursor left	←	**Ctrl+B**
Move cursor right	→	**Ctrl+F**

3.4.2 Changing Editing Keys

The keys that are available for editing a command are determined by the settings of a library called **Readline**. The keys are typically set to match the key assignments found within the emacs text editor (a popular Linux editor) by default.

To bind the keys to match the key assignments found with another popular text editor, the vi editor, the shell can be configured with the set -o vi command. To set the key bindings back to the emacs text editor, use the set -o emacs command.

> **Note**
>
> If you don't know how to use the vi editor yet, you may want to stick with the emacs keys as the vi editor has a bigger learning curve.
>
> The vi editor will be covered in greater detail later in the course.

To automatically configure the edit history options at login, edit the ~/.inputrc file. If this file doesn't exist, the /etc/inputrc file is used instead. Key bindings are set differently in configuration files than on the command line; for example, to enable the vi key binding mode for an individual, add the following lines to the ~/.inputrc file:

```
set editing-mode vi
set keymap vi
```

If the situation is reversed, perhaps due to the above two lines being added to the /etc/inputrc file, a user can enable the emacs mode by adding the following lines to the ~/.inputrc file:

```
set editing-mode emacs
set keymap emacs
```

3.4.3 Using the history Command

The history command can be used to re-execute previously executed commands. When used with no arguments, the history command provides a list of previously executed commands:

```
sysadmin@localhost:~$ history
        1  ls
        2  cd test
        3  cat alpha.txt
        4  ls -l
        5  cd ..
        6  ls
        7  history
```

Note that each command is assigned a number that a user can use to re-execute the command.

The history command has numerous options; the most common of these options are listed below:

Option	Purpose
-c	Clear the list
-r	Read the history file and replace the current history
-w	Write the current history list to the history file

As the history list commonly contains five hundred or more commands, it is often helpful to filter the list. The history command accepts a number as an argument to indicate how many commands to list. For example, executing the following command will only show the last three commands from your history.

```
sysadmin@localhost:~$ history 3
        5  cd ..
        6  ls
        7  history
```

Note

The grep command is very useful for filtering the output of commands that produce copious output. For example, to view all of the commands in your history that contain the ls command, search for the ls pattern using the following command:

```
sysadmin@localhost:~$ history | grep "ls"
    1  ls
    4  ls -l
    6  ls
    9  history | grep "ls"
```

The grep command will be covered in greater detail later in the course.

3.4.4 Configuring the history Command

When you close the shell program, it takes commands in the history list and stores them in the ~/.bash_history file, also called the history file. By default, five hundred commands will be stored in the history file. The HISTFILESIZE variable will determine how many commands to write to this file.

If a user wants to store the history commands in a file that is different from ~/.bash_history, then the user can specify an absolute path to the different file as the value for the HISTFILE local variable:

```
HISTFILE=/path/to/file
```

The HISTSIZE variable will determine how many commands to keep in memory for each Bash shell. If the size of HISTSIZE is greater than the size of HISTFILESIZE, then only the most recent number of commands specified by HISTFILESIZE will be written to the history file when the Bash shell exits.

Although it is not normally set to anything by default, you may want to take advantage of setting a value for the HISTCONTROL variable in an initialization file like the ~/.bash_profile file. The HISTCONTROL variable could be set to any of the following features:

```
HISTCONTROL=ignoredups
```

Prevents duplicate commands that are executed consecutively.

```
HISTCONTROL=ignorespace
```

Any command that begins with a space will not be stored. This provides the user with an easy way to execute a command that won't go into the history list.

```
HISTCONTROL=ignoreboth
```

Consecutive duplicates and any commands that begin with a space will not be stored.

```
HISTCONTROL=erasedups
```

Commands that are identical to another command in your history will not be stored. (The previous entry of the command will be deleted from the history list.)

```
HISTCONTROL=ignorespace:erasedups
```

Includes the benefit of erasedups with the advantage of ignorespace.

Another variable that will affect what gets stored in the history of commands is the HISTIGNORE variable. Most users do not want the history list cluttered with basic commands like the `ls`, `cd`, `exit`, and `history` commands. The HISTIGNORE variable can be used to tell Bash not to store certain commands in the history list.

To have commands not included in the history list, include an assignment like the following in the ~/.bash_profile file:

```
HISTIGNORE='ls*:cd*:history*:exit'
```

Note

The * character is used to indicate anything else after the command. So, ls* would match any `ls` command, such as `ls -l` or `ls` etc.

Globbing will be covered in greater detail later in the course.

3.4.5 Executing Previous Commands

The ! exclamation mark is a special character to the Bash shell to indicate the execution of a command within the history list. There are many ways to use the ! exclamation character to re-execute commands; for example, executing two exclamation characters will repeat the previous command:

```
sysadmin@localhost:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
sysadmin@localhost:~$ !!
ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
```

The following table provides some examples of using the exclamation ! character:

History Command	Meaning
!!	Repeat the last command
!-4	Execute the command that was run four commands ago
!555	Execute command number 555
!ec	Execute the last command that started with ec
!?joe	Execute the last command that contained joe

Another way to use the history list is to take advantage of the fact that the last argument of every command is stored. Often a user will type a command to refer to a file, perhaps to display some information about that file. Subsequently, the user may want to type another command to do something else with the same file. Instead of having to type the path again, a user can use a couple of keyboard shortcuts to recall that file path from the previous command line.

By pressing either **Esc+. (Escape+Period)** or **Alt+. (Alt+Period)**, the shell will bring back the last argument of the previous command. Repeatedly pressing one of these key combinations will recall, in reverse order, the last argument of each previous command.