11 – The Environment

As we discussed earlier, the shell maintains a body of information during our shell session called the *environment*. Programs use data stored in the environment to determine facts about the system's configuration. While most programs use *configuration files* to store program settings, some programs also look for values stored in the environment to adjust their behavior. Knowing this, we can use the environment to customize our shell experience.

In this chapter, we will work with the following commands:

- `printenv` – Print part or all of the environment
- `set` – Set shell options
- `export` – Export environment to subsequently executed programs
- `alias` – Create an alias for a command

What is Stored in the Environment?

The shell stores two basic types of data in the environment; though, with `bash,` the types are largely indistinguishable. They are *environment variables* and *shell variables*. Shell variables are bits of data placed there by `bash`, and environment variables are everything else. In addition to variables, the shell stores some programmatic data, namely *aliases* and *shell functions*. We covered aliases in Chapter 5, "Working with Commands." and we will cover shell functions (which are related to shell scripting) in Part 4.

Examining The Environment

To see what is stored in the environment, we can use either the `set` builtin in `bash` or the `printenv` program. The `set` command will show both the shell and environment variables, while `printenv` will only display the latter. Since the list of environment contents will be fairly long, it is best to pipe the output of either command into `less.`

```
[me@linuxbox ~]$ printenv | less
```

Doing so, we should get that looks like this:

```
USER=me
PAGER=less
LSCOLORS=Gxfxcxdxbxegedabagacad
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
PATH=/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games:/usr/local/games
DESKTOP_SESSION=ubuntu
QT_IM_MODULE=ibus
QT_QPA_PLATFORMTHEME=appmenu-qt5
JOB=dbus
PWD=/home/me
XMODIFIERS=@im=ibus
GNOME_KEYRING_PID=1850
LANG=en_US.UTF-8
GDM_LANG=en_US
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
 MASTER_HOST=linuxbox
IM_CONFIG_PHASE=1
COMPIZ_CONFIG_PROFILE=ubuntu
GDMSESSION=ubuntu
SESSIONTYPE=gnome-session
XDG_SEAT=seat0
HOME=/home/me
SHLVL=2
LANGUAGE=en_US
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LESS=-R
LOGNAME=me
COMPIZ_BIN_PATH=/usr/bin/
LC_CTYPE=en_US.UTF-8
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/
usr/share/
QT4_IM_MODULE=xim
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-IwaesmWaT0
LESSOPEN=| /usr/bin/lesspipe %s
INSTANCE=
```

What we see is a list of environment variables and their values. For example, we see a variable called `USER`, which contains the value `me`. The `printenv` command can also list the value of a specific variable.

```
[me@linuxbox ~]$ printenv USER
me
```

The `set` command, when used without options or arguments, will display both the shell and environment variables, as well as any defined shell functions. Unlike `printenv`, its output is courteously sorted in alphabetical order.

```
[me@linuxbox ~]$ set | less
```

It is also possible to view the contents of a variable using the `echo` command, like this:

```
[me@linuxbox ~]$ echo $HOME
/home/me
```

One element of the environment that neither `set` nor `printenv` displays is aliases. To see them, enter the `alias` command without arguments.

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-
dot --show-tilde'
```

Some Interesting Variables

The environment contains quite a few variables, and though the environment will differ from the one presented here, we will likely see the variables listed in Table 11-1 in our environment.

Table 11-1: Environment Variables

Variable	Contents
`DISPLAY`	The name of the display if we are running a graphical environment. Usually this is ":0", meaning the first display generated by the X server.

EDITOR	The name of the program to be used for text editing.
SHELL	The name of the user's default shell program.
HOME	The pathname of your home directory.
LANG	Defines the character set and collation order of your language.
OLDPWD	The previous working directory.
PAGER	The name of the program to be used for paging output. This is often set to /usr/bin/less.
PATH	A colon-separated list of directories that are searched when we enter the name of a executable program.
PS1	This stands for "prompt string 1." This defines the contents of the shell prompt. As we will later see, this can be extensively customized.
PWD	The current working directory.
TERM	The name of your terminal type. Unix-like systems support many terminal protocols; this variable sets the protocol to be used with your terminal emulator.
TZ	Specifies your time zone. Most Unix-like systems maintain the computer's internal clock in *Coordinated Universal Time* (UTC) and then display the local time by applying an offset specified by this variable.
USER	Your username.

Don't worry if some of these values are missing. They vary by distribution.

How Is The Environment Established?

When we log on to the system, the bash program starts, and reads a series of configuration scripts called *startup files*, which define the default environment shared by all users. This is followed by more startup files in our home directory that define our personal environment. The exact sequence depends on the type of shell session being started. There are two kinds.

- **A login shell session** A login shell session is one in which we are prompted for our username and password. This happens when we start a virtual console session, for example.
- **A non-login shell session** A non-login shell session typically occurs when we

launch a terminal session in the GUI.

Login shells read one or more startup files as shown in Table 11-2.

Table 11-2: Startup Files for Login Shell Sessions

File	Contents
/etc/profile	A global configuration script that applies to all users.
~/.bash_profile	A user's personal startup file. This can be used to extend or override settings in the global configuration script.
~/.bash_login	If ~/.bash_profile is not found, bash attempts to read this script.
~/.profile	If neither ~/.bash_profile nor ~/.bash_login is found, bash attempts to read this file. This is the default in Debian-based distributions, such as Ubuntu.

Non-login shell sessions read the startup files listed in Table 11-3.

Table 11-3: Startup Files for Non-Login Shell Sessions

File	Contents
/etc/bash.bashrc	A global configuration script that applies to all users.
~/.bashrc	A user's personal startup file. It can be used to extend or override settings in the global configuration script.

In addition to reading the startup files in Table 11-3, non-login shells inherit the environment from their parent process, usually a login shell.

Take a look and see which of these startup files are installed. Remember—since most of the filenames listed above start with a period (meaning that they are hidden), we will need to use the "-a" option when using `ls`.

The `~/.bashrc` file is probably the most important startup file from the ordinary user's point of view, since it is almost always read. Non-login shells read it by default and most startup files for login shells are written in such a way as to read the `~/.bashrc` file as well.

What's in a Startup File?

If we take a look inside a typical `.bash_profile` (taken from a CentOS 6 system), it looks something like this:

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

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

Lines that begin with a "#" are *comments* and are not read by the shell. These are there for human readability. The first interesting thing occurs on the fourth line, with the following code:

```
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

This is called an *if compound command*, which we will cover fully when we get to shell scripting in Part 4, but for now, here is a translation:

```
If the file "~/.bashrc" exists, then
   read the "~/.bashrc" file.
```

We can see that this bit of code is how a login shell gets the contents of `.bashrc`. The next thing in our startup file has to do with the `PATH` variable.

Ever wonder how the shell knows where to find commands when we enter them on the command line? For example, when we enter `ls`, the shell does not search the entire computer to find `/bin/ls` (the full pathname of the `ls` command); rather, it searches a list of directories that are contained in the `PATH` variable.

The `PATH` variable is often (but not always, depending on the distribution) set by the `/etc/profile` startup file with this code:

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

PATH is modified to add the directory $HOME/bin to the end of the list. This is an example of parameter expansion, which we touched on in Chapter 7. “Seeing the World As tthe Shell Sees It.” To demonstrate how this works, try the following:

```
[me@linuxbox ~]$ foo="This is some "
[me@linuxbox ~]$ echo $foo
This is some
[me@linuxbox ~]$ foo=$foo"text."
[me@linuxbox ~]$ echo $foo
This is some text.
```

Using this technique, we can append text to the end of a variable's contents.

By adding the string $HOME/bin to the end of the PATH variable's contents, the directory $HOME/bin is added to the list of directories searched when a command is entered. This means that when we want to create a directory within our home directory for storing our own private programs, the shell is ready to accommodate us. All we have to do is call it bin, and we’re ready to go.

> **Note**: Many distributions provide this PATH setting by default. Debian based distributions, such as Ubuntu, test for the existence of the ~/bin directory at login and dynamically add it to the PATH variable if the directory is found.

Lastly, we have:

```
export PATH
```

The export command tells the shell to make the contents of PATH available to child processes of this shell.

Modifying the Environment

Since we know where the startup files are and what they contain, we can modify them to customize our environment.

Which Files Should We Modify?

As a general rule, to add directories to your `PATH` or define additional environment variables, place those changes in `.bash_profile` (or the equivalent, according to your distribution; for example, Ubuntu uses `.profile`). For everything else, place the changes in `.bashrc`.

> **Note:** Unless you are the system administrator and need to change the defaults for all users of the system, restrict your modifications to the files in your home directory. It is certainly possible to change the files in `/etc` such as `profile`, and in many cases it would be sensible to do so, but for now, let's play it safe.

Text Editors

To edit (i.e., modify) the shell's startup files, as well as most of the other configuration files on the system, we use a program called a *text editor*. A text editor is a program that is, in some ways, like a word processor in that it allows us to edit the words on the screen with a moving cursor. It differs from a word processor by only supporting pure text and often contains features designed for writing programs. Text editors are the central tool used by software developers to write code and by system administrators to manage the configuration files that control the system.

A lot of different text editors are available for Linux; most systems have several installed. Why so many different ones? Because programmers like writing them and since programmers use them extensively, they write editors to express their own desires as to how they should work.

Text editors fall into two basic categories: graphical and text based. GNOME and KDE both include some popular graphical editors. GNOME ships with an editor called `gedit`, which is usually called "Text Editor" in the GNOME menu. KDE usually ships with three, which are (in order of increasing complexity) `kedit`, `kwrite`, and `kate`.

There are many text-based editors. The popular ones we'll encounter are `nano`, `vi`, and `emacs`. The `nano` editor is a simple, easy-to-use editor designed as a replacement for the `pico` editor supplied with the PINE email suite. The `vi` editor (which on most Linux systems replaced by a program named `vim`, which is short for "vi improved") is the traditional editor for Unix-like systems. It will be the subject of our next chapter. The `emacs` editor was originally written by Richard Stallman. It is a gigantic, all-purpose, does-everything programming environment. While readily available, it is seldom installed on most Linux systems by default.

Using a Text Editor

Text editors can be invoked from the command line by typing the name of the editor followed by the name of the file we want to edit. If the file does not already exist, the editor will assume that we want to create a new file. Here is an example using `gedit`:

```
[me@linuxbox ~]$ gedit some_file
```

This command will start the `gedit` text editor and load the file named “some_file”, if it exists.

Graphical text editors are pretty self-explanatory, so we won't cover them here. Instead, we will concentrate on our first text-based text editor, `nano`. Let's fire up `nano` and edit the `.bashrc` file. But before we do that, let's practice some “safe computing.” Whenever we edit an important configuration file, it is always a good idea to create a backup copy of the file first. This protects us in case we mess up the file while editing. To create a backup of the `.bashrc` file, do this:

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

It doesn't matter what we call the backup file; just pick an understandable name. The extensions “.bak”, “.sav”, “.old”, and “.orig” are all popular ways of indicating a backup file. Oh, and remember that `cp` will *overwrite existing files* silently.

Now that we have a backup file, we'll start the editor.

```
[me@linuxbox ~]$ nano .bashrc
```

Once `nano` starts, we’ll get a screen like this:

```
  GNU nano 2.0.3           File: .bashrc

# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

```

```
# User specific aliases and functions









                          [ Read 8 lines ]
^G Get Help^O WriteOut^R Read Fil^Y Prev Pag^K Cut Text^C Cur Pos
^X Exit    ^J Justify ^W Where Is^V Next Pag^U UnCut Te^T To Spell
```

Note: If your system does not have `nano` installed, you may use a graphical editor instead.

The screen consists of a header at the top, the text of the file being edited in the middle, and a menu of commands at the bottom. Since `nano` was designed to replace the text editor supplied with an email client, it is rather short on editing features.

The first command we should learn in any text editor is how to exit the program. In the case of `nano`, we press `Ctrl-x` to exit. This is indicated in the menu at the bottom of the screen. The notation `^X` means `Ctrl-x`. This is a common notation for control characters used by many programs.

The second command we need to know is how to save our work. With `nano` it's `Ctrl-o`. With this knowledge, we're ready to do some editing. Using the down arrow key and/or the `PageDown` key, move the cursor to the end of the file, and then add the following lines to the `.bashrc` file:

```
umask 0002
export HISTCONTROL=ignoredups
export HISTSIZE=1000
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Note: Your distribution may already include some of these, but duplicates won't hurt anything.

Table 11-4 details the meaning of our additions:

Table 11-4: Additions to Our .bashrc

Line	Meaning
`umask 0002`	Sets the `umask` to solve the problem with the shared directories we discussed in Chapter 9, "Permissions."
`export HISTCONTROL=ignoredups`	Causes the shell's history recording feature to ignore a command if the same command was just recorded.
`export HISTSIZE=1000`	Increases the size of the command history from the usual default of 500 lines to 1,000 lines.
`alias l.='ls -d .* --color=auto'`	Creates a new command called `l.`, which displays all directory entries that begin with a dot.
`alias ll='ls -l --color=auto'`	Creates a new command called `ll`, which displays a long-format directory listing.

As we can see, many of our additions are not intuitively obvious, so it would be a good idea to add some comments to our `.bashrc` file to help explain things to the humans. Using the editor, change our additions to look like this:

```
# Change umask to make directory sharing easier
umask 0002

# Ignore duplicates in command history and increase
# history size to 1000 lines
export HISTCONTROL=ignoredups
export HISTSIZE=1000

# Add some helpful aliases
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Ah, much better! With our changes complete, press `Ctrl-o` to save our modified `.bashrc` file, and press `Ctrl-x` to exit `nano`.

Why Comments Are Important

Whenever you modify configuration files it's a good idea to add some comments to document your changes. Sure, you'll probably remember what you changed tomorrow, but what about six months from now? Do yourself a favor and add some comments. While you're at it, it’s not a bad idea to keep a log of what changes you make.

Shell scripts and `bash` startup files use a “#” symbol to begin a comment. Other configuration files may use other symbols. Most configuration files will have comments. Use them as a guide.

You will often see lines in configuration files that are *commented out* to prevent them from being used by the affected program. This is done to give the reader suggestions for possible configuration choices or examples of correct configuration syntax. For example, the `.bashrc` file of Ubuntu 18.04 contains these lines:

```
# some more ls aliases
#alias ll='ls -l'
#alias la='ls -A'
#alias l='ls -CF'
```

The last three lines are valid alias definitions that have been commented out. If you remove the leading “#” symbols from these three lines, a technique called *uncommenting*, you will activate the aliases. Conversely, if you add a “#” symbol to

> the beginning of a line, you can deactivate a configuration line while preserving the information it contains.

Activating Our Changes

The changes we have made to our `.bashrc` will not take effect until we close our terminal session and start a new one because the `.bashrc` file is only read at the beginning of a session. However, we can force `bash` to reread the modified `.bashrc` file with the following command:

```
[me@linuxbox ~]$ source ~/.bashrc
```

After doing this, we should be able to see the effect of our changes. Try one of the new aliases.

```
[me@linuxbox ~]$ ll
```

Summing Up

In this chapter, we learned an essential skill—editing configuration files with a text editor. Moving forward, as we read man pages for commands, take note of the environment variables that commands support. There may be a gem or two. In later chapters, we will learn about shell functions, a powerful feature that you can also include in the `bash` startup files to add to your arsenal of custom commands.

Further Reading

- The INVOCATION section of the `bash` man page covers the `bash` startup files in gory detail.