Environment Variables

A shell is simply a program that supplies certain services to users.

As such, a shell may take parameters whose values modify or define certain behaviors.

These parameters (or *shell variables* or global *environment variables*) typically have values that are set in certain *configuration files*.

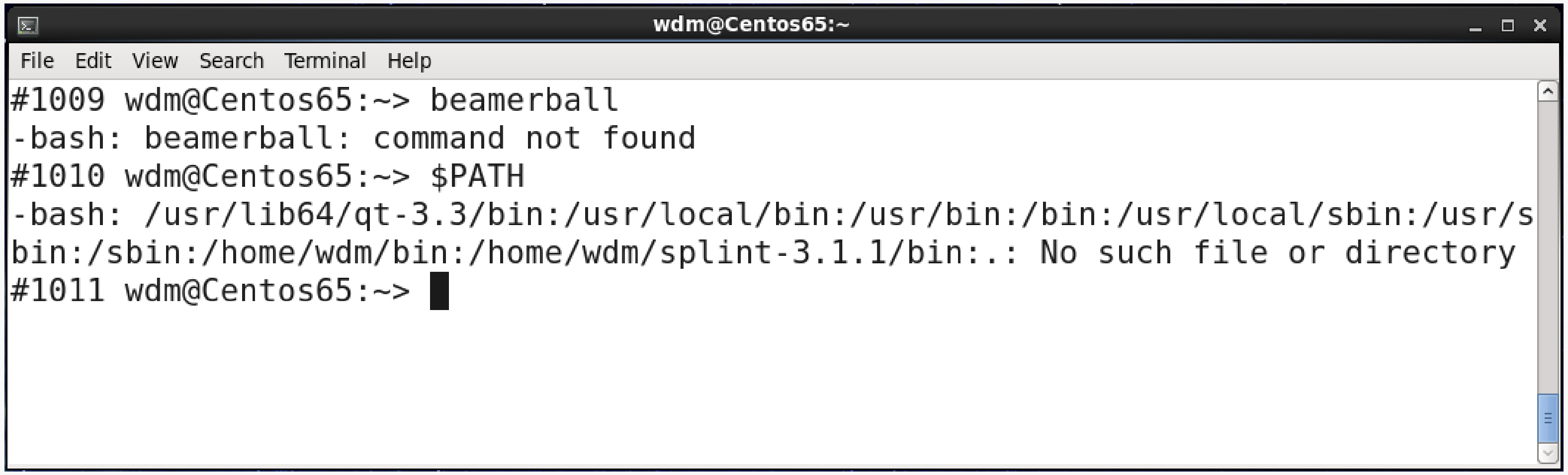
When you install Linux, or use your rlogin account, many of these parameters will have default values determined by the system administrator or by Linux installer.

You may generally modify those default values and even define new parameters by editing configuration files within your home directory.

Open a bash shell and enter the command **$HOME**… this will show the current value of the environment variable **HOME**.

The Path Variable

The environment variable that is most often encountered is the **PATH** variable, which determines which directories the shell will search (and in what order) when the shell attempts to locate programs you are attempting to execute.



```
#1009 wdm@Centos65:~> beamerball
-bash: beamerball: command not found
#1010 wdm@Centos65:~> $PATH
-bash: /usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/s
bin:/sbin:/home/wdm/bin:/home/wdm/splint-3.1.1/bin:.: No such file or directory
#1011 wdm@Centos65:~>
```

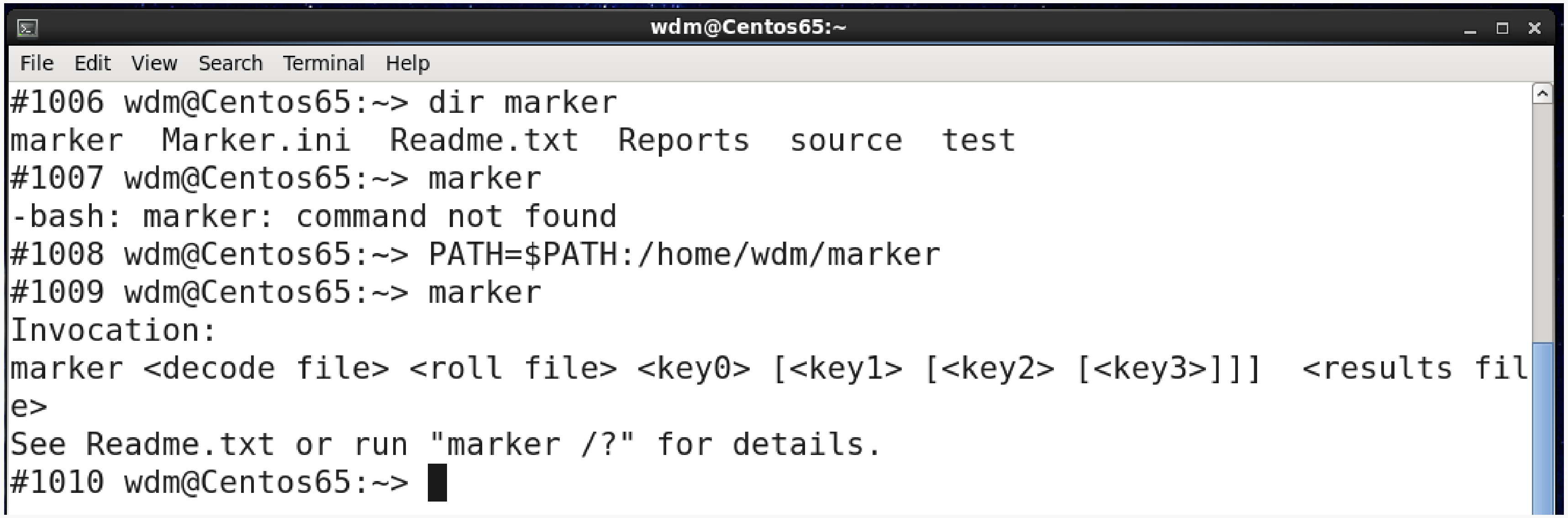
We see that the default **PATH** for this CentOS installation contains the directories:

/usr/lib64/qt-3.3/bin
/usr/local/bin
/usr/bin
/bin
/usr/local/sbin
/usr/sbin
/sbin
/home/wdm/bin
/home/wdm/splint-3.1.1/bin
(which apparently does not exist!)
./

Setting a Variable

You can change the value of a shell variable from the command line.

Let’s add the directory for one of my tools to the **PATH**:



```
#1006 wdm@Centos65:~> dir marker
marker  Marker.ini  Readme.txt  Reports  source  test
#1007 wdm@Centos65:~> marker
-bash: marker: command not found
#1008 wdm@Centos65:~> PATH=$PATH:/home/wdm/marker
#1009 wdm@Centos65:~> marker
Invocation:
marker <decode file> <roll file> <key0> [<key1> [<key2> [<key3>]]]  <results fil
e>
See Readme.txt or run "marker /?" for details.
#1010 wdm@Centos65:~> 
```

Note that we can now run the user program **marker** without specifying the path.

But… this only resets **PATH** for the current shell session.

Bash Shell Startup

When a bash shell is started, it automatically executes commands stored in certain files.

There are three kinds of shells:

(interactive) login shells	(sets values for various shell variables)
/etc/profile	a system file that only the **root** user can modify
~/.bash_profile	files in your **HOME** directory that you can change
~/.bash_login	
~/.profile	
interactive non-login shells	(inherits login shell variables from files above)
/etc/bashrc	another system file
~/.bashrc	another file in your **HOME** directory
non-interactive shells	(inherits login shell variables from files above)

files named by the environment variable **BASH_ENV**

Side Note: Hidden Files

If you try the **ls** command in your home directory, you will (probably) notice that the file **.bash_profile** is not listed.

Filenames that begin with a period are hidden by default.

You can use the **ls -a** command will show hidden files as well as non-hidden files.

Common Variance

When you open an interactive terminal session in Linux, the sequence described on the preceding slide is probably NOT followed by default.

In particular, **~/.bash_profile** is not executed automatically, and therefore changes you make to it will not be effective.

There is a simple fix for the issue:

- open a terminal session and go to Edit/Profile Preferences
- select the Title and Command tab
- check the box for “Run command as a login shell”

In fact, in my rlogin installation, **~/.bash_profile** did not exist initially; I had to create it with a text editor.

~/.bash_profile

You should use **~/.bash_profile** to set changes to the PATH variable because **~/.bash_profile** is only executed once.

Here is a sample **.bash_profile** taken from Sobell:

```
if [ -f ~/.bashrc ]; then    # if .bashrc exists
                             #   in the home directory
   source ~/.bashrc          #    run it
fi

PATH=$PATH:.                # add working directory to the path

export PS1='[\h \W \!]\$ '    # configure the shell prompt
```

Normally, **~/.bashrc** is invoked from another configuration file, as shown here.

See the note in Sobell regarding adding the working directory to the path; NEVER add it at the beginning of the path!

Sobell has a good discussion of the various options for the appearance of the prompt.

~/.bashrc

Here is a sample **~/.bashrc** adapted from Sobell:

```
if [ -f /etc/bashrc ]; then   # if  global bashrc exists, run it
   source /etc/bashrc         #   note:  no period in file name
fi
if [ -d "$HOME/bin" ] ; then     # add user's bin directory to path
    PATH="$HOME/bin:$PATH"
fi
set -o noclobber                 # prevent silent overwriting of files
                                 #     (by redirection)

alias rm='rm -i'                 # always use interactive rm cmd
alias cp='cp -i'                 #     and interactive cp cmd
alias recent='history | tail'    # displays last few cmds run
alias ll='ls -alF'
```

alias commands are a convenient way to create mnemonics for specialized execution of system commands.

Defining Aliases

alias commands are a convenient way to create mnemonics for specialized execution of system commands.

The syntax (for the bash shell) is:

```
alias <mnemonic>='command to be run'
```

There are no spaces around the equal sign, and the specification of the command to be run must be enclosed in single quotes if it contains spaces.

Defining Aliases

```
alias list='ls -gAFG -t -r --time-style=long-iso'
```

```
#1017 wdm@Centos65:code> list
total 12
-rw-rw-r--. 1 3442 2014-10-23 22:34 BinaryInt.c
-rw-rw-r--. 1  816 2014-10-23 22:34 driver.c
-rw-rw-r--. 1 2183 2014-10-23 22:34 BinaryInt.h
```

```
-A, --almost-all
      do not list implied . and ..

-F, --classify
      append indicator (one of */=>@|) to entries

-g    like -l, but do not list owner

-G, --no-group
      in a long listing, don't print group names

-r, --reverse
      reverse order while sorting

-t    sort by modification time

--time-style=STYLE
      with  -l, show times using style STYLE: full-iso, long-iso, iso, . . .
```

Shell Scripting

The shell supports a built-in programming language, called a scripting language.

(Different shells support different scripting languages.)

We will explore shell scripting in detail later in the course.

For now, we'll show a couple of examples to illustrate how the bash scripting language works.

Functions

```
first_vowel() {                  # utility fn called by piggy()

   return `expr index "$1" aeiouAEIOU`
}

piggy() {                        # translates params to pig-Latin

   for x; do                     # iterate through params
      first_vowel $x             # locate first vowel in param
      retval="$?"                # save return value from first_vowel()

      if [[ $retval -eq 1 ]]; then     # vowel is at front of param
         echo -n $x"way "
      else                             # vowel is not at front of param
         length=`expr length "$x"`
         prefix=`expr substr "$x" 1 $((retval-1))`
         suffix=`expr substr "$x" $retval $length`
         echo -n $suffix$prefix"ay "
      fi

   done                          # end of for loop body
   echo                          # bang out a newline
   return 0
}
```

Analysis

```
first_vowel() {                           # utility fn called by piggy()
                                          # no return type, param list
   return `expr index "$1" aeiouAEIOU`
}
```

```
. . .
first_vowel $x   # call passes param; syntax like command-line invocation
. . .
```

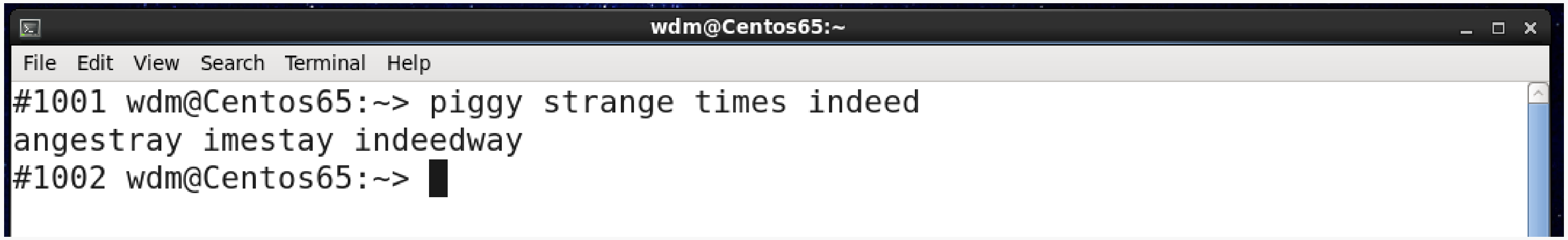
```
retval="$?"      # return value accessed as $?
```

Analysis

```
. . .
if [[ $retval -eq 1 ]]; then      # have if-then construct

   echo -n $x"way "

else                              # else is optional

   length=`expr length "$x"`
   prefix=`expr substr "$x" 1 $((retval-1))`
   suffix=`expr substr "$x" $retval $length`
   echo -n $suffix$prefix"ay "

fi                     # if/fi pair delimit if body
. . .
```

Adding Function to Shell

If we add the function(s) to the **.bashrc** file, they become available in future shell invocations:



```
#1001 wdm@Centos65:~> piggy strange times indeed
angestray imestay indeedway
#1002 wdm@Centos65:~>
```

Of course, this is just a "joke" function...

A More Useful Function

```
protect() {

   if [[ $# -eq 0 ]]; then                    # check for a parameter
      echo "Invocation:  protect filename"
      return 1;
   fi

   if [[ ! -f $1 ]]; then                     # see if it's a regular file
      echo "$1 is not a regular file."
      return 2;
   fi

   chmod g-rwx,o-rwx $1     # if so, remove all group/other access to it

   return 0
}
```