

Lecture 4

Arguments

Arguments to Shell Programs

So far, we have focused on some of the fundamentals of shell programming that are required to build more complex programs. In this discussion, I'll focus on passing information to the shell from the command line. This gives you the ability to have your program actually operate using the parameters you specify at run-time. These parameters are known as arguments. UNIX makes the arguments given on the command line available to the program that is executed as a result of that command line (the process). It does not matter whether the program is a binary executable (such as `/bin/cat` or `/bin/ls`) or an ASCII text shell program such as those you have been creating. In its simplest form, the basic syntax of a shell command line looks like:

```
$ command argument1 argument2 argument3...
```

Figure 1

When you write programs, you have to think like you are two different people – the user of the program and the programmer. These two roles has different view points. The user is concerned about *using* the program and what it ultimate *does*, and the programmer is concerned about *how* to make it happen.

The user thinks of command line arguments – those things that are passed to control or influence the running of the program. But when you are wearing the hat of the programmer, you don't think of the arguments passed on the command line as arguments. Instead, the command line arguments are referred to as **positional parameters**. From the programmer's point of view, the command line arguments were passed in a certain order, and you need to have access to them. Fortunately, the shell makes these available to you in a very simple way.

From within the shell program, the positional parameters (i.e. the command lines arguments) are know affectionately as **\$1** for the first argument, **\$2** for the second argument, **\$3** for the third, **\$4** the fourth, and so on, up to **\$9** the ninth command line argument. So why not just call them *arguments* instead of the odd sounding *positional parameters*? That's a good question. But you don't expect me to have an answer do you? Ok, I'll cough one up. Part of the answer has to do with the nomenclature used by the shell for variables – they are all called parameters (keyword or positional). The other reason is that there are only nine positional parameters (\$1 to \$9), yet there can be any number of command line arguments. And finally, from your programming point of view, you deal with them in the order that they appear, one at a time. Their position on the command line will likely have meaning to you.

What if there are more than 9 command line arguments? I'll discuss that later; for now, let's focus on the positional parameters and how to use them. Consider the shell program below, named **showargs**:

```
$ cat showargs
#!/bin/sh

echo The first positional parameter is: $1
echo The second positional parameter is: $2
```

When the **showargs** program is run, the shell will set **\$1** to contain the first argument on the command line passed to **showargs**. And you can probably guess that **\$2** will contain the second command line argument.

```
$ showargs Eat Prunes
The first positional parameter is: Eat
The second positional parameter is: Prunes
```

Now, is **\$1** really the value of some variable named **1** (one). No, not really. You already know that variables cannot start with a number (they can only start with letters or an underscore). Certainly the shell could violate its own rules and use a variable that starts with, or is even named **1**. But there really is no variable named **1** – instead, there are these things called positional parameters, and they are referred to as mentioned: **\$1**, **\$2**, etc. The author of the textbook claims that there are variables whose names are **1**, **2**, **3**, etc. This is incorrect, because a) you can not set them by direct assignment¹, and b) you cannot refer to them as **1**, **2**, **3** ... anyway. Instead, you always refer to them as **\$1**, or **\$2**, or **\$3** ... to refer to the value of that positional parameter. But it really does not matter since this is like the philosophical question "*Does a tree falling in woods make a sound if you are not there to hear it?*" I have a more pragmatic answer, "*Why would I care – it doesn't wake me up from a good sleep either way!*"

What is the value of **\$2** for example when there is only 1 command line argument? Any positional parameters that have no corresponding command line argument are left empty (i.e. **\$2** would be *null*).

```
$ showargs Yuck!
The first positional parameter is: Yuck!
The second positional parameter is:
```

And the opposite is true where there are more command line arguments than you handle in your shell program. They are available, if you would only use them... just like lost coins in your couch.

```
$ showargs An argument unused
The first positional parameter is: An
The second positional parameter is: argument
```

Below is a shell program that will start to show you how quotes work. Quotes really do cause the shell to ignore special characters (meta-characters, spaces, tabs, newlines, etc.). This time, notice that the first argument below is only one word² because the usual whitespace (tab, space, newline) word separator was disabled by the single quotes. The two words were turned into one with quotes, and of course, the same would hold true for double quotes.

```
$ showargs 'Eat Prunes'
The first positional parameter is: Eat Prunes
The second positional parameter is:
```

Even newlines can be passed unmodified to your shell program:

¹ The positional parameters can be (re-)assigned using the shell built-in **set** command.

² Remember, a word is a sequence of characters that are not special characters - the word ends when a special character is encountered, and spaces are special.

```
$ showargs 'Eat
> Prunes'
The first positional parameter is: Eat Prunes
The second positional parameter is:
```

Wait! If newlines are passed to the program, then where did the newline go in the example above? To understand this, you need to again consider your two roles – user and programmer. From the user's point of view, the user passed a single argument that contains a newline – the quotes maintained the whitespace, as was noted earlier. But from the programmer's point of view, we did not quote the **\$1** positional parameter reference to protect that embedded newline. The sub-shell running your program turned the embedded newline inside **\$1** into a space. We have seen this in the previous lecture and homework. To avoid this replacement, we need to modify our program and double quote the reference.

```
$ cat showargs
#!/bin/sh

echo The first positional parameter is: "$1"
echo The second positional parameter is: "$2"
```

Running the example again provides the expected results.

```
$ showargs 'Eat
Prunes'
The first positional parameter is: Eat
Prunes
The second positional parameter is:
```

Passing quoted material from command line to shell program, and then again from shell program to other programs can prove a challenge. You will discover this through the duration of the course. For example, look at the following example:

```
$ showargs 'backslash \c'
The first positional parameter is: backslash The second positional parameter is:
```

The **\c** was passed to the **echo** command within **showargs**, and **echo** uses **\c** to suppress a newline at the end of the output. You probably did not want the results above. This is a more difficult problem to resolve.

\$# – Number of Arguments

The shell provides another built-in variable that tells a shell program how many command line arguments were passed when it was called. The special symbol **\$#** is the number of arguments passed to the program, and can be used just like any other read-only variable. The modified **showargs** program below demonstrates this:

```
$ cat showargs
#!/bin/sh

echo Number of command line arguments: $#
echo The first positional parameter is: "$1"
echo The second positional parameter is: "$2"
```

Notice that there are two command line arguments for the *Enjoy Kiwi* run below and that `$#` confirms this as expected:

```
$ showargs Enjoy Kiwi
Number of command line arguments: 2
The first positional parameter is: Enjoy
The second positional parameter is: Kiwi
```

Quoting the two words, making a single command line argument shows expected results as well:

```
$ showargs 'Enjoy Kiwi'
Number of command line arguments: 1
The first positional parameter is: Enjoy Kiwi
The second positional parameter is:
```

`$*` – All Arguments

The shell also provides another convenience variable that gives you access to all of the arguments passed on the command line. The variable `$*` contains all of the command line arguments, in the order they were passed. The modified `showargs` program demonstrates this:

```
$ cat showargs
#!/bin/sh

echo Number of command line arguments: $#
echo The first positional parameter is: "$1"
echo The second positional parameter is: "$2"
echo All: $*
$
$ showargs How much wood could a wood chuck chuck if a wood chuck could chuck
wood?
Number of command line arguments: 15
The first positional parameter is: How
The second positional parameter is: much
All: How much wood could a wood chuck chuck if a wood chuck could chuck wood?
```

Notice above that there were 15 arguments, as reported by `$#` , and these are all available in `$*` . But how would you get access to them? You can immediately reference the first nine arguments through the positional parameters `$1` to `$9` . To answer this, we need a new section.

The shift Built-in Command

Because the shell only provides you with nine positional parameters, there needs to be another way to access command line arguments 10 and beyond³. This is accomplished with the `shift` built-in command. To understand how `shift` works, think of the positional parameters `$1` to `$9` as a small window that allows you to view only nine command line arguments at a time. Let's look at our previous example command:

```
$ showargs How much wood could a wood chuck chuck if a wood chuck could chuck
wood?
```

³ Which silly person came up with the idea that in proper English you must write out *one* through *nine*, but can use the numbers 10 and above using digits?

Figure 2 shows the window of nine positional parameters. The current view only allows access to the first nine arguments.

```
$ showargs How much wood could a wood chuck chuck if a wood chuck could chuck wood?
      $1  $2  $3  $4  $5  $6  $7  $8  $9
```

Figure 2

The **shift** command moves the window down (to the right) by one argument. Figure 3 shows the effect of **shift** being run once. Notice that **\$1** is now what **\$2** used to be, **\$2** is now what **\$3** used to be, and so on. The first command line argument *How* is now no longer available and cannot be accessed again. If you need it, you must store it in your own variables before you do a **shift**.

```
$ showargs How much wood could a wood chuck chuck if a wood chuck could chuck wood?
      $1  $2  $3  $4  $5  $6  $7  $8  $9
```

Figure 3

If a numeric argument is supplied with **shift**, as in **shift 5**, the shift will move the window that many places down the command line. Figure 4 demonstrates this.

```
$ showargs How much wood could a wood chuck chuck if a wood chuck could chuck wood?
      $1  $2  $3  $4  $5  $6  $7  $8  $9
```

Figure 4

In addition, the **\$#** and **\$*** variables are updated to reflect the change caused by shift. Let's look again at the **showargs** program, which has been modified to show the effects of different **shift** operations:

```
$ cat showargs
#!/bin/sh

echo Number of command line arguments: $#
echo The first positional parameter is: "$1"
echo The second positional parameter is: "$2"
echo All: $*

shift
echo
echo After shift
echo Number of command line arguments: $#
echo The first positional parameter is: "$1"
echo The second positional parameter is: "$2"
echo All: $*
```

```

shift 5
echo
echo After shift 5
echo Number of command line arguments: $#
echo The first positional parameter is: "$1"
echo The second positional parameter is: "$2"
echo All: $*

```

And here is the output:

```

$ showargs How much wood could a wood chuck chuck if a wood chuck could chuck
  wood?
Number of command line arguments: 15
The first positional parameter is: How
The second positional parameter is: much
All: How much wood could a wood chuck chuck if a wood chuck could chuck wood?

After shift
Number of command line arguments: 14
The first positional parameter is: much
The second positional parameter is: wood
All: much wood could a wood chuck chuck if a wood chuck could chuck wood?

After shift 5
Number of command line arguments: 9
The first positional parameter is: chuck
The second positional parameter is: chuck
All: chuck chuck if a wood chuck could chuck wood?

```

Note that shifting beyond the last available argument produces an error, so the above code is not very good, since it will generate errors when **showargs** is called with fewer six arguments.

Using vi Effectively

Now that your shell programs are getting more complex, I am going to show you a way to go through the edit-test-edit more quickly. I think you will like this. All of the UNIX shells except **sh** give you the ability to stop the currently running program, and return control to the shell again so that you can enter more commands. And since **vi** gives you the ability to automatically write a file when you stop the edit session, you can use this to your advantage. Note: if you are using **sh**, this will not work, because **sh** does not support what is called **job control**.

First, follow the steps below to see how this works.

1. Start a **vi** session, creating a new file called **go**. The command is **vi go**.
2. Now, enter the command **echo hello** in the file. Make sure to hit Escape to leave insert mode.
3. Now, type **^Z** (that's the Control-Z key). You should notice that your editor seemed to have just stopped, and you are now back in the shell.
4. Type **jobs** - this shows you the list of stopped jobs. Your editor is still there, waiting to be continued.
5. Do an **ls** and look for the file **go**. You should notice that it does not exist. That is because when the **vi** session was stopped, **vi** did not write out the file or changes to the file. You have to tell **vi** to save changes when it is stopped using **^Z**. You do this with a setting called **autowrite** within **vi**.
6. Return to the editor session using the shell's built-in **fg** command. This command will restart the most recently stopped job. You should be back in **vi** exactly where you left it.

7. Set the **autowrite** option with the **vi** command **:set aw** and hit enter. The **:** takes you to **vi**'s command mode and **set aw** sets the **autowrite** option (**aw** is an abbreviation allowed by **vi** for **autowrite**).
8. Now, stop the **vi** session as you did before with **^Z**. The editor should be stopped immediately, but this time, notice that **vi** output a message indicating that it wrote out the file. This is good!
9. Now, make **go** executable with **chmod** and then run the **go** program. It should have run correctly.
10. All of this was to demonstrate how **autowrite** and stopped jobs work with together with **vi**. Now let's see the real power. Foreground your **vi** session again with the **fg** command as before. You should be back in the editor.
11. Add a new command to the file: **echo it works** and hit Escape to leave insert mode.
12. Now stop the job again with **^Z** and re-run the **go** program. Notice that you did not have save the file
13. Go back into **vi** with **fg** to continue your work. When you are done, quit **vi** your usual way (with **ZZ** or **:wq**).

To set the **autowrite** option so that it is set every time you start **vi**, add the line **set aw** to your **.exrc** file in your home directory. If the file does not exist yet, create it. Next time you start **vi**, autowrite mode will be automatically set for you. Every time you stop the **vi**, your file will be written out for you. If you do not want to write out the changes before you stop the job, then unset **autowrite** mode with **:set noaw**, or just quit **vi** your usual way without saving changes.

What is so nice about this method of working is that when returning to stopped **vi** session, all of **vi**'s state is exactly where it was when you left. Your cursor is where you last left it, your buffers are intact, and nicest of all, this means that **undo** works correctly. You can quickly try out a change, and if you don't like it, just use **vi**'s undo command to undo the change. This is even more powerful when you use **vim** instead of **vi** because **vim** has infinite undos.

I have been working in this mode for a very long time, and find it indispensable. It may take you a little time to get used to the idea of **edit-^Z-fg**, but once you do, you'll never go back. Try it for the next week or two - if you're not completely satisfied, I'll give you your money back. And you can keep the tote back as our free gift.

Have a good week.