Shell configuration and scripts

Last time, we learned about shells and, towards the end, about environment variables. We saw that some environment variables are used to communicate with programs, other contain information that helps the shell do its job (for example, the `PATH` environment variable). Some commands can be affected by specific environment variables. To pick an example out of thin air, consider the `less` command. If you look at the man pages for `less`, you see that if the environment variable `LESS` is defined, it is used to communicate command line options passed automatically to `less`. For example, the command line option `-M` makes `less` use a very verbose prompt as its command prompt. To automatically have `less` do this, you can use (in Bourne shells):

```
export LESS=-M
```

Since many programs have such conventions, it would be nice to have a place where all of these configuration options can be specified. It turns out there are a few such places.

Every shell is set up to read from configuration files whenever it is started up. These configuration files can, among other things, define environment variables (and local variables). Read the appropriate shell man pages for actual documentation. The shell `bash` reads the file `.bashrc` when it is started, and when it is used as a login shell, it attempts to read `.bash_profile`, `.bash_login`, or `.profile` (all in your home directory). The shell `tcsh` read the file `.tcshrc` (`.cshrc` if `.tcshrc` doesn't exist) in your home directory everytime it is started. Moreover, when it is started as the login shell, it also reads the file `.login` in your home directory.

Those configuration files can contain any sequence of shell commands. Typically, what you find in such files are environment variable settings (such as the `LESS` variable above.)

Another thing you may find are *aliases*. An alias is simply a name that you use as a command, but that *expands* into a more complicated command before being executed. For example, in my case, I always get confused by `ls`: when I learned Unix originally, I was using `lc`, another directory listing program, to list my directories. On system where `lc` is not available, I typically alias `lc` to mean `ls -F`. In `bash`, I do this with `alias lc='ls -F'`. Then, whenever I use `lc` *`args`* as a command name, the shell translates it into `ls -F` *`args`*. Here are the alias-related commands for the Bourne shell family:

- `alias` lists all the aliases on the system

- `alias` *`name`* list the alias (if any) for *`name`*
- `alias` *`name=value`* sets an alias
- `unalias` *`name`* deletes an alias

For the C shell family, it's similar:

- `alias` lists all the aliases on the system
- `alias` *`name`* list the alias (if any) for *`name`*
- `alias` *`name value`* sets an alias
- `unalias` *`name`* deletes an alias

Aliases are also standard things that occur in a shell configuration files. In fact, any shell command can occur in a shell configuration file. You can put in `echo` commands to see the progress through the configuration file as it is loaded.

Scripts

The example of shell configuration files given above is quite useful. The idea that we can put shell commands all in one place and executed all at once is attractive. For instance, if you have the same sequence of operations you want to perform on many files, it would be nice to put all those operations in one place and invoke them repeatedly.

The example of shell configuration files is an instance of what is called a *shell script*. Shell scripts contains sequences of commands executed by the shell. For the time being, we will understand a shell script to be a sequence of shell commands, one per line. Here's a shell script, a fairly simple one at that (we'll assume these lines have been put in a file `test.sh`):

```
fgrep 'hi' file1 > tmp1
fgrep 'hello' file3 > tmp2
cat tmp1 tmp2 | sort
```

Executing this script will basically search for all the lines in `file1`containing `hi`, all the lines in `file2` containing `hello`, sort all those lines, and print them to `stdout`.

Before being able to execute a script, you need to prepare it first. To do that, you put *as the first line of the script*, a line such as:

```
#!/bin/bash
```

Following the `#!` should be the path to the shell to use to execute this script. After you've added this line, you turn the script into an executable file by changing its execute permission, as in `chmod u+x test.sh`. Now you can simply invoke the script by using its name as a command, i.e. by calling `test.sh`. The effect of this will be to start the shell specified by the first line of the script, and execute the content of the script.

You can redirect the standard input and standard output of a script as you would any other command, although it gets a bit touchy to understand how the commands in the script will behave with respect to `stdin`. For example, every command of the script sending to `stdout` will send to the `stdout` of the script, but every command reading from `stdin` will require a different file.