

## Shell Script Assignments

In assignments you will be writing shell programs, commonly called *shell scripts*. The first section this handout is a suggested method of creating shell scripts. The second section discusses goals and qualities of a successful shell script, including how to document a shell script. The last section discusses other issues: including debugging, using functions, and using temporary files.

### CREATING A SHELL SCRIPT

There are two golden rules to writing shell scripts. Whether you follow the suggested steps in this section or not, you will save yourself a lot of grief if you observe these two key rules:

1. *Whenever possible, test the commands you want to use **outside** of your shell script. Incrementally editing and running your script can be very tedious. Polish the commands at the command line.*
2. **Do some design!** *Think the problem through before you begin writing code. A minute of thought in the design stage can save you hours in the debugging stage. Shell scripts are not easy to debug.*

Here are some more general steps that implement the above and more:

1. Before you begin, write an algorithm for your shell script. This can be at whatever level you want. A good starting point is large functional blocks like 'Process Arguments'. Get as specific as you want at this point, but you want to at least have the general structure of your program ready before you start writing it. You can add details during the coding process, but changing the structure can create a mess. I should add a note here on shell functions: although they can be useful, they are very limited and misuse can obscure your program more than it can help it. (see below under *Using Functions*) Until you understand these issues, don't use shell functions.
2. Create a file for the shell script using `vi`. Add the important `#!/bin/bash` line as the first line in your shell script. and use `chmod` to make the shell script executable. Now you can execute the shell script directly by typing its name (note: on linux, the current directory may not be in your search path. You will have to prefix your shell script name with `./`) DO NOT start writing your shell script yet.
3. Figure out the first set of commands your shell script needs by typing them into the shell at the `$` prompt. You should place them in your shell script only after you are sure they are working. You may want to do this piecemeal, by testing the commands and adding them to the script step by step. As you become more comfortable with writing shell scripts, you will be able to add some of the commands without testing them (such as to handle the arguments). You should ALWAYS experiment with commands outside of your shell script.
4. Add `echo` statements to the file to print any descriptive information you may need. These `echo` statements should describe *what* the output is, not how the program works. You can also use `echo` statements for debugging purposes (see below under *Debugging Shell Scripts*)
5. Repeat these steps piecemeal until each of the functional blocks in your original algorithm is complete.
6. Add documentation to your shell script. This is described in the next section under *Documentation*.

### A Note on Your environment

Unix provides many ways to customize your environment. It is possible to customize your environment and change the names and the behavior of commands (e.g., with aliases). You are urged to experiment with these capabilities on your own. However, you must operate in a standard environment when you are doing your homework to enable me to grade it. If I can't follow what you are doing you won't get a grade.

Your aim is to write shell scripts that others can use. Thus, your shell scripts must run the same in my standard environment as in yours. This is your responsibility.

## GOALS AND QUALITIES OF A SUCCESSFUL SHELL SCRIPT

You probably think your goal in writing a shell script is to solve a problem. That is part of it. However, as with any computer program, a shell script that simply solves a problem is not necessarily a good one - in fact, it can be worse than none at all! If you write a poor shell script that solves a problem and it is difficult to follow, modify, or has bugs, the tendency will be to try to keep it around or to try to fix it. Professional programs are written with an eye to these issues: how easy will this be to understand? modify? what kind of unexpected things might happen and how will they affect the performance of the shell script?

Our goal is to write professional quality shell scripts that people would actually want to use, that are easy to maintain and to understand. To do this, we have certain goals for them:

### Completeness and Correctness:

Does the shell script solve the problem? Does it perform correctly even if unexpected things happen?

### Documentation:

When the issue of documentation is raised, the questions that arise are "Why document?" and "What do I document?". The answer to the first question was discussed above. You might wonder what that has to do with a school assignment; the answer here is simple: to get some practice! Knowing how to do documentation may not get you a job, but not doing it can certainly lose you a job.

The second question is more interesting. The answer depends on who is reading the shell script. There are two general audiences: *the user of the shell script* and *someone who may need to modify it*. These two kinds of readers have different needs and require different kinds of documentation..

The **user** of the shell script is someone who is familiar with Unix at a user level. The user needs to know how to run the shell script, what it needs, and what it does. They don't want Unix details, and are not necessarily a programmer. They will look for documentation in two places: external documentation (as in a manual page) and in the beginning of the program. Thus, the beginning of your shell script (the program header) should have the following information:

- *a general statement* about what the shell script does. Notice that this is **what** it does, not **how** it does it.
- *how to run the shell script*, including any information on options and arguments.
- *any input the shell script needs*. This includes whether the shell script functions in command or interactive mode, and any input the user must provide.
- *what output the shell script generates* and where it goes.
- *any assumptions the shell script makes about its environment*. For example, if your shell script expects to be able to write to the current directory you should say so. If it needs to use standard data from a particular place this must be documented, including where it must be, and its format. (However, if it uses a standard system file, such as `passwd`, its format does not need to be documented.)

The goal of the documentation in the program header is to get user to read it. This means that *too much documentation can be as bad as insufficient documentation*. Avoid documenting things that are obvious. Keep documentation concise and organize it so that it is *easy to read*, with the most important points at the beginning or otherwise highlighted. Examples are better than a lengthy description: a synopsis of how to run your program saves many words of explanation, as does an example of what the input (or output) file looks like.

The second audience is someone who might need to modify your program. This is another Unix programmer with expertise as extensive as your own. This person needs to know how your program does what it does. *This type of documentation should be in the body of the code, with* the pieces of code that it documents. Remember, this programmer has expertise also, so the only things necessary to document are things that are not immediately obvious. If you are using a filter like `cut`, don't explain how it works. A comment reminding the reader the contents of the fields you are cutting would be more helpful.

Documentation in the body of the code (called *inline documentation*) should *not* be added to the right of the code. Instead, add entire lines of documentation above the section of code being documented.

Although this should be obvious, **ensure that your documentation is correct**. If you modify your shell script, alter the documentation if necessary!

**Documentation is so important, that it will count 15% of the grade of each program.** Again, if you get a job writing programs, knowing how to write documentation may determine whether you keep it or not.

### **Style:**

Your shell script should be written so that it is easy to read. This means that the flow of control should be obvious, the names of variables descriptive and the design as *simple as possible*.

Proper style requires the use of consistent indentation and white space. You should also take care that comments are easy to see and do not disappear into the code.

Style also includes coding style. I will be asking such questions as *Did you do unnecessary work? Is your solution well-designed? Did you complete one part of the task before starting another? Are your error messages and other output done consistently and are they understandable? Did they give the user sufficient information to readily correct the problem?*

### **Robustness:**

Your shell script should detect and diagnose errors when they occur. It should then take "appropriate" action, which will depend on the circumstances. It is impossible to check for every contingency. Some situations may be taken for granted - for example you can usually assume that you can write to the /tmp area (unless your file could be very large). Others cannot - like assuming an argument exists. The task is to anticipate situations that could result in an error and give the user enough information to enable him or her to remedy it. **All error messages should be written to standard error.**

One note about checking for errors: some beginning shell programmers take the approach of trying to predict a problem before it happens. For example, if you are going to create a file, you could do the necessary checks to see if you can write to it. This usually requires more code and is much more difficult than the more standard Unix mechanism: just do it and see if it succeeded. This sounds sloppy, but in the Unix realm it is more elegant, faster, and a lot easier. However, this is not appropriate for all situations: You will need to strike a balance between these two philosophies.

No matter what the input, your shell script should be free of syntax errors. If a user forgets to provide an argument and your shell script aborts with a seemingly unrelated error, you have not done your job. In addition, your shell script should only abort if a problem prevents it from doing any more meaningful work. If it is possible to continue, do so. Don't just give up at the first sign of trouble. For example, if your shell script is processing a series of items and it finds one that is inaccessible or that it can not process, give a message and continue with the next item if possible.

Robustness also includes maintaining control over the output of any command used in your shell script.

In addition to errors, you should also anticipate error messages when they may occur. Many times system-generated error messages resulting from some command your shell script uses will simply confuse the user. Diagnose the error and generate your own message, discarding the system message. You should also be aware of when a command may produce some normal output. A common mistake is to use an `if` statement and `grep` to check if something contains a pattern and forget that `grep` outputs the lines that match.

- continued on the next page -

## OTHER ISSUES

### Temporary Files

From time to time your shell script will need to use temporary or scratch files. Although it may be tempting to create a temporary file in the current directory with a descriptive name like `employee_names` you should do so since this file may overwrite an existing file. You must create a path to a non-existent (temporary) file. Unique temporary file paths are easily generated using `mktemp(1)`. It ensures that the path is unique.

Follow these instructions:

- assign the current directory to an environment variable `TMPDIR` like this: `export TMPDIR=.` Of course you only need to do this once at the beginning of your shell script. (The `TMPDIR` environment variable is used by `mktemp(1)` to find the directory to hold temporary files.)

Then, for each temporary file you need, do the following:

- select a descriptive variable name to hold the path to the temporary file. A variable to hold the path for a generic temporary file might be `TMPFILE`.
- create the temporary file path using `mktemp` with command substitution like this:  
`TMPFILE=$(mktemp)`
- create the temporary file using `touch`: `touch "$TMPFILE"`
- Use `$TMPFILE` as your filename in your code.

Don't forget to remove your temporary files whenever your shell script exits. **Your shell script may only remove the temporary files that it creates!** CS260A students should also ensure your temporary files are removed if your shell script is killed.

Real-world shell scripts use the system temporary directory to hold temporary files. Once your shell script is working you can use the system temporary directory if you like. First, verify that your shell script is removing all of your temporary files correctly. (It is important to the efficient function of the system that `/tmp` remain free of clutter, otherwise its contents may be unexpectedly removed.) Then simply change `TMPDIR` to `/tmp`, or just delete its assignment above (as `/tmp` is the default temporary directory).

### Shell Functions

Students with programming backgrounds who learn to program in shell want to use functions for everything. You must resist this temptation. Functions in any programming language serve two purposes: organization and abstraction. The achievement of these goals requires encapsulation. This is what is difficult in the shell. If a function is not properly encapsulated the result of using the function can be to obscure, rather than to clarify, the overall task.

The shell is a language in which declarations are implicit and variables are global by default. If you are not careful designing your functions, they will have side-effects, changing the value of global variables in ways that are only obvious if you read the function carefully. Side-effects by their nature are hidden when you see the function being used. This results in one of two situations: the reader will miss an important event due to the side-effect OR the (more conservative) reader will resort to re-analyzing the function whenever it is used - negating any abstraction.

When you are using functions in the shell, you must restrict their use to specific, well-defined, repetitive tasks, and carefully define them so as to avoid side-effects (or, in the very least, advertise them).

Let's look at a simple example: You are writing a shell script to work on a file with a particular record format. You will be processing this file a line (record) at a time, and one of the tasks you have is to analyze the record to see if it is in the correct (legal) format. Here are two students' approach to this task. Both use a variable `line` to hold the current record.

<i>The function</i>	<i>The Main Program</i>
<pre>checkline() {   nfields=\$(echo "\$line"       tr '#' '\n'   wc -l)   if [ \$nfields -ne 4 ]; then     islegal=false   else     islegal=true   fi }</pre>	<pre># line holds the current record checkline if [ "\$islegal" = "false" ]; then   echo "line \$lineno is illegal" &gt;&amp;2   continue fi</pre>

This looks like a very simple function, and a useful one. However, if you are reading this shell script, here are some questions that might arise when you read the definition of the function `checkline`:

- where did `line` come from?
- is `nfields` used outside of `checkline`? is `islegal` used outside of `checkline` and how?

When you see the use of `checkline` in the main program you should ask the following questions:

- how does `checkline` get the line to check?
- what can the values of `islegal` be?
- is `islegal` used later in the shell script? was it used previously and did it have a value?

Furthermore, unless you go back and reread `checkline`, you would not remember that it alters the global variable `nfields`, which you may need to know about. **Thus, even though this function is well-behaved, defining it poorly leaves open questions which must be re-asked each time you see the function used.**

Let's rewrite this function slightly:

<pre>checkline() {   # returns success if the argument   # has 4 #-separated fields   local nfields   nfields=\$(echo "\$1"       tr '#' '\n'   wc -l)   [ \$nfields -eq 4 ] }</pre>	<pre># line holds the current record if ! checkline "\$line"; then   echo "line \$lineno is illegal" &gt;&amp;2   continue fi</pre>
--	---

This time, when you read the function `checkline`, you see that it only uses arguments and local variables. Now you know later when you see it that it isn't suspicious **and the function is shorter** (note here that I have made use of the shell rule that the return value of a function is the status of the last command executed to avoid a verbose `if/then` construct.)

## Debugging Shell Scripts

For all its power, debugging in shell is rudimentary. The simplest technique consists of adding `echo` statements to trace the execution of the program. For more global debugging, you can get the shell to output a trace of each of the commands it executes as it does so by use of the shell trace option `-x`. Simply add the line `set -x` to the beginning of your shell script, or run your shell script by giving it to `bash` with the option `-x`. As an example, if your shell script is normally invoked as `xxx arg1 arg2 arg3`, use

```
bash -x xxx arg1 arg2 arg3
```

to run it with debugging output. Turning on the `-x` option in the shell can generate a lot of output. You might want to redirect the debugging output (which comes out on standard error) to a file.