Troubleshooting

Syntactic Errors

- One general class of errors is syntactic. Syntactic errors involve mistyping some element of shell syntax. The shell will stop executing a script when it encounters this type of error.
- In the following discussions, we will use this script to demonstrate common types of errors:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

- As written, this script runs successfully.

```
[me@linuxbox ~]$ trouble
Number is equal to 1.
```

Missing Quotes

- Let's edit our script and remove the trailing quote from the argument following the first echo command.

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
    echo "Number is equal to 1.
else
    echo "Number is not equal to 1."
fi
```

Here is what happens:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 10: unexpected EOF while looking for
matching `"'
/home/me/bin/trouble: line 13: syntax error: unexpected end of file
```

- It generates two errors. Interestingly, the line numbers reported by the error messages are not where the missing quote was removed but rather much later in the program.
- If we follow the program after the missing quote, we can see why. bash will continue looking for the closing quote until it finds one, which it does, immediately after the second echo command.
- After that, bash becomes very confused. The syntax of the subsequent if command is broken because the fi statement is now inside a quoted (but open) string.
- In long scripts, this kind of error can be quite hard to find. Using an editor with syntax highlighting will help since, in most cases, it will display quoted strings in a distinctive manner from other kinds of shell syntax.
- If a complete version of vim is installed, syntax highlighting can be enabled by entering this command:

```
:syntax on
```

Missing or Unexpected Tokens

- Another common mistake is forgetting to complete a compound command, such as if or while. Let’s look at what happens if we remove the semicolon after test in the if command:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ] then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

- The result is this:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 9: syntax error near unexpected token
`else'
/home/me/bin/trouble: line 9: `else'
```

- Again, the error message points to an error that occurs later than the actual problem.
- What happens is really pretty interesting. As we recall, if accepts a list of commands and evaluates the exit code of the last command in the list.
- In our program, we intend this list to consist of a single command, [, a synonym for test.
- The [command takes what follows it as a list of arguments; in our case, that's four arguments: $number, 1, =, and].
- With the semicolon removed, the word then is added to the list of arguments, which is syntactically legal.
- The following echo command is legal, too. It's interpreted as another command in the list of commands that if will evaluate for an exit code.
- The else is encountered next, but it's out of place since the shell recognizes it as a reserved word (a word that has special meaning to the shell) and not the name of a command, which is the reason for the error message.

Unanticipated Expansions

- It's possible to have errors that occur only intermittently in a script. Sometimes the script
- will run fine and other times it will fail because of the results of an expansion. If we return our missing semicolon and change the value of number to an empty variable, we can demonstrate.

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=

if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

- Running the script with this change results in the following output:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 7: [: =: unary operator expected
 Number is not equal to 1.
```

- We get this rather cryptic error message, followed by the output of the second echo command. The problem is the expansion of the number variable within the test command. When the following command:

```
[ $number = 1 ]
```

- undergoes expansion with number being empty, the result is this:

```
[  = 1 ]
```

- which is invalid and the error is generated. The = operator is a binary operator (it requires a value on each side), but the first value is missing, so the test command expects a unary operator (such as -z) instead. Further, since the test failed (because of the error), the if command receives a non-zero exit code and acts accordingly, and the secondecho command is executed.

- This problem can be corrected by adding quotes around the first argument in the test command.

```
[ "$number" = 1 ]
```

- Then when expansion occurs, the result will be this:

```
[ "" = 1 ]
```

- This yields the correct number of arguments. In addition to empty strings, quotes should be used in cases where a value could expand into multiword strings, as with filenames containing embedded spaces.

Logical Errors

- Unlike syntactic errors, logical errors do not prevent a script from running. The script will run, but it will not produce the desired result, because of a problem with its logic.
- There are countless numbers of possible logical errors, but here are a few of the most common kinds found in scripts:
    1. Incorrect conditional expressions. It’s easy to incorrectly code an if/then/else and have the wrong logic carried out. Sometimes the logic will be reversed, or it will be incomplete.
    2. “Off by one” errors. When coding loops that employ counters, it is possible to overlook that the loop may require that the counting start with zero, rather than one, for the count to conclude at the correct point. These kinds of errors result in either a loop “going off the end” by counting too far or a loop missing the last iteration by terminating one iteration too soon.
    3. Unanticipated situations. Most logic errors result from a program encountering data or situations that were unforeseen by the programmer. As we have seen, this can also include unanticipated expansions, such as a filename that contains embedded spaces that expands into multiple command arguments rather than a single filename.

Defensive Programming

- It is important to verify assumptions when programming. This means a careful evaluation of the exit status of programs and commands that are used by a script. Here is an example, based on a true story. An unfortunate system administrator wrote a script to perform a maintenance task on an important server. The script contained the following two lines of code:

```
cd $dir_name
rm *
```

- There is nothing intrinsically wrong with these two lines, as long as the directory named in the variable, dir_name, exists.
- But what happens if it does not? In that case, the cd command fails, and the script continues to the next line and deletes the files in the current working directory.
- Not the desired outcome at all! The hapless administrator destroyed an important part of the server because of this design decision.

- Let’s look at some ways this design could be improved. First, it might be wise to ensure that the dir_name variable expands into only one word by quoting it and make the execution of rm contingent on the success of cd.

```
cd "$dir_name" && rm *
```

- This way, if the cd command fails, the rm command is not carried out. This is better but still leaves open the possibility that the variable, dir_name, is unset or empty, which would result in the files in the user’s home directory being deleted. This could also be avoided by checking to see that dir_name actually contains the name of an existing directory.

```
[[ -d "$dir_name" ]] && cd "$dir_name" && rm *
```

- Often, it is best to include logic to terminate the script and report an error when a situation such as the one shown previously occurs.

```
# Delete files in directory $dir_name
if [[ ! -d "$dir_name" ]]; then
   echo "No such directory: '$dir_name'" >&2
   exit 1
fi
if ! cd "$dir_name"; then
   echo "Cannot cd to '$dir_name'" >&2
   exit 1
fi
if ! rm *; then
   echo "File deletion failed. Check results" >&2
   exit 1
fi
```

- Here, we check both the name, to see that it is an existing directory, and the success of the cd command. If either fails, a descriptive error message is sent to standard error, and the script terminates with an exit status of one to indicate a failure.

set -e, set -u, and set -o PIPEFAIL

- One thing we notice about bash is that when a script executes and a command fails (not including a syntax error in the script itself), the script will happily continue to the next command.
- Often this is undesirable and the POSIX standard and subsequently, bash attempts to address this issue. bash offers a setting that will to attempt handle errors automatically, which simply means that with this setting enabled, a script will terminate if any command (with some necessary exceptions) returns a non-zero exit status.
- To invoke this setting, we place the command set -e near the beginning of the script. Several bash coding standards insist on using this feature along with a couple of related settings, set -u which terminates a script if there is an uninitialized variable, and set –o PIPEFAIL which causes script termination if the final element in a pipeline fails.

- Using these features is not recommended. It is better to design proper error handling and not rely on set -e as a substitute for good coding practices.
- The Bash FAQ #105 provides the following opinion on this:
  - `set -e` was an attempt to add "automatic error detection" to the shell.
  - Its goal was to cause the shell to abort any time an error occurred, so you don't have to put `|| exit 1` after each important command.
  - That goal is non-trivial, because many commands intentionally return non-zero. For example,

    `if [ -d /foo ]; then ...; else ...; fi`
  - Clearly we don't want to abort when the `[ -d /foo ]` command returns non-zero (because the directory does not exist) -- our script wants to handle that in the else part. So the implementers decided to make a bunch of special rules, like "commands that are part of an if test are immune", or "commands in a pipeline, other than the last one, are immune."
  - These rules are extremely convoluted, and they still fail to catch even some remarkably simple cases. Even worse, the rules change from one Bash version to another, as bash attempts to track the extremely slippery POSIX definition of this "feature." When a subshell is involved, it gets worse still -- the behavior changes depending on whether bash is invoked in POSIX mode.

ShellCheck is Your Friend

- There is a program available in most distribution repositories called shellcheck that performs analysis of shell scripts and will detect many kinds of faults and poor scripting practices. It is well worth using it to check the quality of our scripts.
- To use it with a script that has a shebang, we simply do this:

```
shellcheck my_script
```

- ShellCheck will automatically detect which shell dialect to use based on the shebang.
- For testing script code that does not contain a shebang, such as function libraries, we can use ShellCheck this way:

```
shellcheck -s bash my_library
```

- Use the -s option to specify the desired shell dialect

Watch Out for Filenames

- There is another problem with this file deletion script that is more obscure but could be very dangerous. Unix (and Unix-like operating systems) has, in the opinion of many, a serious design flaw when it comes to filenames. Unix is extremely permissive about them.
- In fact, there are only two characters that cannot be included in a filename. The first is the / character since it is used to separate elements of a pathname, and the second is the null character (a zero byte), which is used internally to mark the ends of strings.
- Everything else is legal including spaces, tabs, line feeds, leading hyphens, carriage returns, and so on.

- Of particular concern are leading hyphens. For example, it's perfectly legal to have a file named "-rf ~". Consider for a moment what happens when that filename is passed to rm.
- To defend against this problem, we want to change our rm command in the file deletion script from this:

```
rm *
```

- to the following:

```
rm ./*
```

- This will prevent a filename starting with a hyphen from being interpreted as a command option. As a general rule, always precede wildcards (such as * and ?) with ./ to prevent misinterpretation by commands. This includes things like *.pdf and ???.mp3, for example

Verifying Input

- A general rule of good programming is that if a program accepts input, it must be able to deal with anything it receives.
- This usually means that input must be carefully screened to ensure that only valid input is accepted for further processing.
- We saw an example of this in the previous chapter when we studied the read command. One script contained the following test to verify a menu selection:

```
[[ $REPLY =~ ^[0-3]$ ]]
```

- This test is very specific. It will return a zero exit status only if the string entered by the user is a numeral in the range of zero to three. Nothing else will be accepted. Sometimes these kinds of tests can be challenging to write, but the effort is necessary to produce a high-quality script.

Testing

- Testing is an important step in every kind of software development, including scripts.
- There is a saying in the open-source world, “release early, release often,” that reflects this fact. By releasing early and often, software gets more exposure to use and testing.
- Experience has shown that bugs are much easier to find, and much less expensive to fix, if they are found early in the development cycle.

- Let’s look at the file-deletion problem shown previously and see how this could be coded for easy testing. Testing the original fragment of code would be dangerous since its purpose is to delete files, but we could modify the code to make the test safe.

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        echo rm * # TESTING
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
exit # TESTING
```

- Since the error conditions already output useful messages, we don't have to add any.
- The most important change is placing an echo command just before the rm command to allow the command and its expanded argument list to be displayed, rather than the command actually being executed. This change allows safe execution of the code.
- At the end of the code fragment, we place an exit command to conclude the test and prevent any other part of the script from being carried out. The need for this will vary according to the design of the script.

Test Cases

- To perform useful testing, it's important to develop and apply good test cases. This is done by carefully choosing input data or operating conditions that reflect edge and corner cases. In our code fragment (which is simple), we want to know how the code performs under three specific conditions:
    1. dir_name contains the name of an existing directory.
    2. dir_name contains the name of a nonexistent directory.
    3. dir_name is empty.
- By performing the test with each of these conditions, good test coverage is achieved.
- Just as with design, testing is a function of time, as well. Not every script feature needs to be extensively tested. It's really a matter of determining what is most important.
- Since it could be so potentially destructive if it malfunctioned, our code fragment deserves careful consideration during both its design and testing.

Debugging

- If testing reveals a problem with a script, the next step is debugging. “A problem” usually means that the script is, in some way, not performing to the programmer's expectations.
- If this is the case, we need to carefully determine exactly what the script is actually doing and why. Finding bugs can sometimes involve a lot of detective work.
- A well-designed script will try to help. It should be programmed defensively, to detect abnormal conditions and provide useful feedback to the user. Sometimes, however, problems are quite strange and unexpected, and more involved techniques are required.

Finding the Problem Area

- In some scripts, particularly long ones, it is sometimes useful to isolate the area of the script that is related to the problem. This won't always be the actual error, but isolation will often provide insights into the actual cause.
- One technique that can be used to isolate code is "commenting out" sections of a script. For example, our file deletion fragment could be modified to determine whether the removed section was related to an error.

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
         rm *
    else
         echo "cannot cd to '$dir_name'" >&2
         exit 1
    fi
# else
#   echo "no such directory: '$dir_name'" >&2
#   exit 1
fi
```

- By placing comment symbols at the beginning of each line in a logical section of a script, we prevent that section from being executed. Testing can then be performed again, to see whether the removal of the code has any impact on the behavior of the bug.

Tracing

- Bugs are often cases of unexpected logical flow within a script. That is, portions of the script either are never being executed or are being executed in the wrong order or at the wrong time. To view the actual flow of the program, we use a technique called tracing.
- One tracing method involves placing informative messages in a script that display the location of execution. We can add messages to our code fragment.

```
echo "preparing to delete files" >&2
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
echo "deleting files" >&2
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
echo "file deletion complete" >&2
```

- We send the messages to standard error to separate them from normal output. We also do not indent the lines containing the messages, so it is easier to find when it's time to remove them.
- Now when the script is executed, it's possible to see that the file deletion has been performed.

```
[me@linuxbox ~]$ deletion-script
preparing to delete files
deleting files
file deletion complete
[me@linuxbox ~]$
```

- bash also provides a method of tracing, implemented by the -x option and the set command with the -x option. Using our earlier trouble script, we can activate tracing for the entire script by adding the -x option to the first line.

```
#!/bin/bash -x

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

- When executed, the results look like this:

```
[me@linuxbox ~]$ trouble
+ number=1

+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.
```

- With tracing enabled, we see the commands performed with expansions applied. The leading plus signs indicate the display of the trace to distinguish them from lines of regular output. The plus sign is the default character for trace output. It is contained in the PS4 (prompt string 4) shell variable. The contents of this variable can be adjusted to make the prompt more useful.
- Here, we modify the contents of the variable to include the current line number in the script where the trace is performed. Note that single quotes are required to prevent expansion until the prompt is actually used.

```
[me@linuxbox ~]$ export PS4='$LINENO + '
[me@linuxbox ~]$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

- To perform a trace on a selected portion of a script, rather than the entire script, we can use the set command with the -x option.

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

set -x # Turn on tracing
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

- We use the set command with the -x option to activate tracing and the +x option to de- activate tracing. This technique can be used to examine multiple portions of a troublesome script.

Examining Values During Execution

- It is often useful, along with tracing, to display the content of variables to see the internal workings of a script while it is being executed. Applying additional echo statements will usually do the trick.
- In this trivial example, we simply display the value of the variable number and mark the added line with a comment to facilitate its later identification and removal. This technique is particularly useful when watching the behavior of loops and arithmetic within scripts.

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

echo "number=$number" # DEBUG
set -x # Turn on tracing
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```