# Errors And Signals And Traps (Oh My!) - Part 1

In this lesson, we're going to look at handling errors during the execution of your scripts.

The difference between a good program and a poor one is often measured in terms of the program's *robustness*. That is, the program's ability to handle situations in which something goes wrong.

## Exit Status

As you recall from previous lessons, every well-written program returns an exit status when it finishes. If a program finishes successfully, the exit status will be zero. If the exit status is anything other than zero, then the program failed in some way.

It is very important to check the exit status of programs you call in your scripts. It is also important that your scripts return a meaningful exit status when they finish. I once had a Unix system administrator who wrote a script for a production system containing the following 2 lines of code:

```
# Example of a really bad idea

cd $some_directory
rm *
```

Why is this such a bad way of doing it? It's not, if nothing goes wrong. The two lines change the working directory to the name contained in `$some_directory` and delete the files in that directory. That's the intended behavior. But what happens if the directory named in `$some_directory` doesn't exist? In that case, the `cd` command will fail and the script executes the `rm` command on the current

working directory. Not the intended behavior!

By the way, my hapless system administrator's script suffered this very failure and it destroyed a large portion of an important production system. Don't let this happen to you!

The problem with the script was that it did not check the exit status of the `cd` command before proceeding with the `rm` command.

# Checking The Exit Status

There are several ways you can get and respond to the exit status of a program. First, you can examine the contents of the `$?` environment variable. `$?` will contain the exit status of the last command executed. You can see this work with the following:

```
[me] $ true; echo $?
0
[me] $ false; echo $?
1
```

The `true` and `false` commands are programs that do nothing except return an exit status of zero and one, respectively. Using them, we can see how the `$?` environment variable contains the exit status of the previous program.

So to check the exit status, we could write the script this way:

```
# Check the exit status
```

```
cd $some_directory
if [ "$?" = "0" ]; then
        rm *
else
        echo "Cannot change directory!" 1>&2
        exit 1
fi
```

In this version, we examine the exit status of the `cd` command and if it's not zero, we print an error message on standard error and terminate the script with an exit status of 1.

While this is a working solution to the problem, there are more clever methods that will save us some typing. The next approach we can try is to use the `if` statement directly, since it evaluates the exit status of commands it is given.

Using `if`, we could write it this way:

```
# A better way

if cd $some_directory; then
        rm *
else
        echo "Could not change directory! Aborting." 1>&2
        exit 1
fi
```

Here we check to see if the `cd` command is successful. Only then does `rm` get executed; otherwise an error message is output and the program exits with a code of 1, indicating that an error has occurred.

# An Error Exit Function

Since we will be checking for errors often in our programs, it makes sense to write a function that will display error messages. This will save more typing and promote laziness.

```
# An error exit function

error_exit()
{
        echo "$1" 1>&2
        exit 1
}

# Using error_exit

if cd $some_directory; then
        rm *
else
        error_exit "Cannot change directory!  Aborting."
fi
```

# AND And OR Lists

Finally, we can further simplify our script by using the AND and OR control operators. To explain how they work, I will quote from the **bash** man page:

"The control operators && and ll denote AND lists and OR lists, respectively. An AND list has the form

```
command1 && command2
```

`command2` is executed if, *and only if*, `command1` returns an exit status of zero.

An OR list has the form

```
command1 || command2
```

`command2` is executed if, and only if, `command1` returns a non-zero exit status. The exit status of AND and OR lists is the exit status of the last command executed in the list."

Again, we can use the `true` and `false` commands to see this work:

```
[me] $ true || echo "echo executed"
[me] $ false || echo "echo executed"
echo executed
[me] $ true && echo "echo executed"
echo executed
[me] $ false && echo "echo executed"
[me] $
```

Using this technique, we can write an even simpler version:

```
# Simplest of all
```

```
cd $some_directory || error_exit "Cannot change directory! Aborting"
rm *
```

If an exit is not required in case of error, then you can even do this:

```
# Another way to do it if exiting is not desired

cd $some_directory && rm *
```

I want to point out that even with the defense against errors we have introduced in our example for the use of `cd`, this code is still vulnerable to a common programming error, namely, what happens if the name of the variable containing the name of the directory is misspelled? In that case, the shell will interpret the variable as empty and the `cd` succeed, but it will change directories to the user's home directory, so beware!

# Improving The Error Exit Function

There are a number of improvements that we can make to the `error_exit` function. I like to include the name of the program in the error message to make clear where the error is coming from. This becomes more important as your programs get more complex and you start having scripts launching other scripts, etc. Also, note the inclusion of the `LINENO` environment variable which will help you identify the exact line within your script where the error occurred.

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

```
# A slicker error handling routine

# I put a variable in my scripts named PROGNAME which
# holds the name of the program being run.  You can get this
# value from the first item on the command line ($0).

PROGNAME=$(basename $0)

error_exit()
{

#       ----------------------------------------------------------------
#       Function for exit due to fatal program error
#               Accepts 1 argument:
#                       string containing descriptive error message
#       ----------------------------------------------------------------


        echo "${PROGNAME}: ${1:-"Unknown Error"}" 1>&2
        exit 1
}

# Example call of the error_exit function.  Note the inclusion
# of the LINENO environment variable.  It contains the current
# line number.

echo "Example of error with line number and message"
error_exit "$LINENO: An error has occurred."
```

The use of the curly braces within the `error_exit` function is an example of *parameter expansion*. You can surround a variable name with curly braces (as with `${PROGNAME}`) if you need to be sure it is separated from surrounding text. Some people just put them around every variable out of habit.

That usage is simply a style thing. The second use, `${1:-"Unknown Error"}` means that if parameter 1 (`$1`) is undefined, substitute the string "Unknown Error" in its place. Using parameter expansion, it is possible to perform a number of useful string manipulations. You can read more about parameter expansion in the **bash** man page under the topic "EXPANSIONS".

---