Shell Script Exercises (1)

Compile helper exercise

Write a shell script in a file called `b` (for build) that does the following:

- Your script should run under any Bourne-compatible shell (e.g. not just `bash`), and it should be written so that you can call it with `./b`.
- `./b compile NAME` should compile the file of the given name, so for example `./b compile hello` should run `gcc -Wall -std=c11 -g hello.c -o hello`.
- However, your script should accept both `./b compile hello` and `./b compile hello.c` as input, and do the same thing in both cases, namely compile `hello.c`. The output file for gcc in both cases should be called just `hello`.
- If the source file you provided as an argument does not exist (adding `.c` if necessary) then the script should print an error message and return a nonzero exit status - *not* invoke the C compiler.
- `./b run NAME` should run the program, assuming it exists in the current folder, so both `./b run hello` and `./b run hello.c` should run `./hello`. If it does not exist, again print an error message and exit with a nonzero status, don't try and run the program.
- `./b build NAME` should first compile the C source file, and then if the compile was successful it should run the program. If the compile failed, it should not try and run the program.
- If you call `./b` without any parameters, or `./b COMMAND` with a command other than compile or run or build, it should print some information on how to use it. If you call `./b compile` or another command with no filename at all, then the script should print an error message and exit with a nonzero exit status.

You now have a useful tool for when you are developing C programs. Of course you can add other features of your own like a `debug` command that compiles the file and launches it in `gdb`.

```
#!/bin/sh

compile() {
    # Remove the .c extension if present, then append .c to ensu
    local src="${1%.c}.c"
    local out="${1%.c}"

    # Check if the source file exists
    if [ ! -f "$src" ]; then
        echo "Error: Source file $src does not exist."
        return 1 # Return a non-zero status to indicate failure
    fi

    # Compile the source file
    gcc -Wall -std=c11 -g "$src" -o "$out"
    return $? # Return the exit status of gcc
}

run() {
    local executable="${1%.c}"

    # Check if the executable exists
    if [ ! -x "$executable" ]; then
        echo "Error: Executable $executable does not exist."
        return 1 # Return a non-zero status to indicate failure
    fi

    # Run the executable
    ./"$executable"
}

build() {
    compile "$1" && run "$1"
}
```

```
# Main script starts here
if [ $# -lt 2 ]; then
    echo "Usage: $0 {compile|run|build} <filename>"
    exit 1
fi

command="$1"
filename="$2"

case "$command" in
    compile)
        compile "$filename"
        ;;
    run)
        run "$filename"
        ;;
    build)
        build "$filename"
        ;;
    *)
        echo "Unknown command: $command"
        echo "Usage: $0 {compile|run|build} <filename>"
        exit 1
        ;;
esac
```

Strict Mode

Some programming languages have an optional *strict mode* which treats some constructs as errors that people often do by mistake. It is similar in spirit to `-Werror` in C that treats all warnings as errors. This page suggests using the following line near the top of your shell scripts: `set -euo pipefail`. (It also talks about IFS to improve string handling with spaces, but that's a separate matter.)

You might want to use these yourself if you get into shell scripting. `set` is a shell internal command that sets shell flags which controls how commands are run.

- `set -e` makes the whole script exit if any command fails. This way, if you want to run a list of commands, you can just put them in a script with `set -e` at the top, and as long as all the commands succeed (return 0), the shell will carry on; it will stop running any further if any command returns nonzero. It is like putting `|| exit $?` on the end of every command.
- `set -u` means referencing an undefined variable is an error. This is good practice for lots of reasons.
- `set -o pipefail` changes how pipes work: normally, the return value of a pipe is that of the *last* command in the pipe. With the `pipefail` option, if any command in the pipeline fails (non-zero return) then the pipeline returns that command's exit code.

A couple of notes on `set -u`: if you write something like `rm -rf $FOLDER/` and `$FOLDER` isn't set, then you don't accidentally end up deleting the whole system! Of course, most `rm` implementations will refuse to delete `/` without the `--no-preserve-root` option, and you should not have that trailing slash in the first place. There was a bug in a beta version of Steam for linux where it tried to do `rm -rf "$STEAMROOT/"*` to delete all files in a folder (which explains the slash), but the variable in some cases got set to the *empty string*, which `-u` would not protect against. This was an installer script, so it ran as root which made things even worse.

Exercise: think of an example in a shell script where `pipefail` makes a difference, that is where the last command in a pipe could succeed even if a previous one fails. As a counter-example, `cat FILE | grep STRING` would fail even without `pipefail` if the file does not exist, because grep would immediately get end-of-file on standard input.

```
#!/bin/sh

compile() {
# Remove the .c extension if present, then append .c to ensure the filename ends with .c
local src="${1%.c}.c"
local out="${1%.c}"
```

```
# Check if the source file exists
if [ ! -f "$src" ]; then
    echo "Error: Source file $src does not exist."
    return 1 # Return a non-zero status to indicate failure
fi

# Compile the source file
gcc -Wall -std=c11 -g "$src" -o "$out"
return $? # Return the exit status of gcc
```

}

run() {
local executable="${1%.c}"

```
# Check if the executable exists
if [ ! -x "$executable" ]; then
    echo "Error: Executable $executable does not exist."
    return 1 # Return a non-zero status to indicate failure
fi

# Run the executable
./"$executable"
```

}

build() {
compile "$1" && run "$1"
}

Main script starts here

if [$# -lt 2]; then
echo "Usage: $0 {compile|run|build} <filename>"
exit 1
fi

```
command="$1"
filename="$2"

case "$command" in
compile)
compile "$filename"
;;
run)
run "$filename"
;;
build)
build "$filename"
;;
*)
echo "Unknown command: $command"
echo "Usage: $0 {compile|run|build} <filename>"
exit 1
;;
esac
```