What is shell scripting good for?

Shell scripts are the duct tape and bailing wire of computer programming.

What is shell scripting good for?

Shell scripts are the duct tape and bailing wire of computer programming.
You can use them:

- To automate repeated tasks
- For jobs that require a lot of interaction with files
- To set up the environment for big, complicated programs
- When you need to stick a bunch of programs together into something useful
- To add customizations to your environment

A practical example `runit1.sh`

```
#!/bin/bash

fg++ *.cpp
./a.out
```

Special Variables

- `$?` Exit code of the last command run

Special Variables

- `$?` Exit code of the last command run
- `$0` Name of command that started this script (almost always the script's name)
- `$1, $2, ..., $9` Comand line arguments 1-9
- `$@` All command line arguments except $0
- `$#` The number of command line arguments in `$@`

Special Variables

- `$?` Exit code of the last command run
- `$0` Name of command that started this script (almost always the script's name)
- `$1, $2, ..., $9` Comand line arguments 1-9
- `$@` All command line arguments except $0
- `$#` The number of command line arguments in `$@`

And now, a brief message from our sponsors:

- Bash really likes splitting things up into words.
- `for arg in $@` will NOT do what you want.

Special Variables

- `$?` Exit code of the last command run
- `$0` Name of command that started this script (almost always the script's name)
- `$1, $2, ..., $9` Comand line arguments 1-9
- `$@` All command line arguments except $0
- `$#` The number of command line arguments in `$@`

And now, a brief message from our sponsors:

- Bash really likes splitting things up into words.
- `for arg in $@` will NOT do what you want.
- `for arg in "$@"` correctly handles args with spaces.
- In general, when using the value of a variable you don't control, it is wise to put `"` s around the variable.

A Spiffier Example `runit2.sh`

```
#!/bin/bash

fg++ *.cpp -o "$1"
./"$1"
```

Conditional Statements `if.sh`

```
#!/bin/bash

# Emit the appropriate greeting for various people

if [[ $1 = "Jeff" ]]; then
        echo "Hi, Jeff"
elif [[ $1 == "Maggie" ]]; then
        echo "Hello, Maggie"
elif [[ $1 == *.txt ]]; then
        echo "You're a text file, $1"
elif [ "$1" = "Stallman" ]; then
        echo "FREEDOM!"
else
        echo "Who in blazes are you?"
fi
```

Conditional Operators

- `[ ]` is shorthand for the `test` command.
- `[[ ]]` is a `bash` keyword.
- `[ ]` works on most shells, but `[[ ]]` is less confusing.

Conditional Operators

- `[ ]` is shorthand for the `test` command.
- `[[ ]]` is a `bash` keyword.
- `[ ]` works on most shells, but `[[ ]]` is less confusing.
- `(( ))` is another `bash` keyword. It does arithmetic.

String Comparison Operators for `[[ ]]`

- `=` String equality OR pattern matching if the RHS is a pattern.
- `!=` String ineqaulity.

String Comparison Operators for `[[ ]]`

- `=` String equality OR pattern matching if the RHS is a pattern.
- `!=` String ineqaulity.
- `<` The LHS sorts before the RHS.
- `>` The LHS sorts after the RHS.

String Comparison Operators for `[[ ]]`

- `=` String equality OR pattern matching if the RHS is a pattern.
- `!=` String ineqaulity.
- `<` The LHS sorts before the RHS.
- `>` The LHS sorts after the RHS.
- `-z` The string is empty (length is **z**ero).
- `-n` The string is **n**ot empty (e.g. `[[ -n "$var" ]]`).

Numeric Comparison Operators for `[[ ]]`

- `-eq` Numeric equality (e.g. `[[ 5 -eq 5 ]]`).
- `-ne` Numeric inequality.

Numeric Comparison Operators for `[[ ]]`

- `-eq` Numeric equality (e.g. `[[ 5 -eq 5 ]]`).
- `-ne` Numeric inequality.
- `-lt` Less than
- `-gt` Greater than
- `-le` Less than or equal to
- `-ge` Greater than or equal to

File Operators for `[[ ]]`

- `-e` True if the file exists (e.g. `[[ -e story.txt ]]`)
- `-f` True if the file is a regular file
- `-d` True if the file is a directory

There are a lot more file operators that deal with even fancier stuff.

General Operators for `[[ ]]`

- `&&` Logical AND
- `||` Logical OR
- `!` Logical NOT

General Operators for `[[ ]]`

- `&&` Logical AND
- `||` Logical OR
- `!` Logical NOT
- You can use parentheses to group statements too.

Shell Arithmetic with `(( ))`

- This mostly works just like C++ arithmetic does.
- `**` does exponentiation
- You can do ternaries! `(( 3 < 5 ?  3 :  5 ))`
- You don't need `$` on the front of normal variables.
- Shell Arithmetic Manual

Spiffy++ Example `runit3.sh`

```
#!/bin/bash

if (( $# > 0 )); then
        g++ *.cpp -o "$1"
        exe="$1"
else
        g++ *.cpp
        exe=a.out
fi

if [[ $? -eq 0 ]]; then
        ./"$exe"
fi
```

(Could you spiff it up even more with file checks?)

Case statements

```
#!/bin/bash
case $1 in
        a)
                echo "a, literally"
                ;;
        b*)
                echo "Something that starts with b"
                ;;
        *c)
                echo "Something that ends with c"
                ;;
        "*d")
                echo "*d, literally"
                ;;
        *)
                echo "Anything"
                ;;
esac
```

For Looping `for.sh`

```
#!/bin/bash

echo C-style:
for (( i=1; i < 9; i++ )); do
        echo $i;
done

echo BASH-style:
for file in *.sh; do
        echo $file
done
```

While Looping `while.sh`

```
#!/bin/bash

input=""
while [[ $input != "4" ]]; do
        echo "Please enter the random number: "
        read input
done
```

Reading Files `quine.sh`

```
#!/bin/bash

IFS= # Inter-field separator.
     # Unset to prevent word splitting

while read f; do
        echo "$f"
done < "$0"
```

What is a quine?

Functions `function.sh`

```
#!/bin/bash

parrot() {
        while (( $# > 0 )); do
                echo "$1"
                shift
        done
}

parrot These are "several arguments"
```

Miscellany

- Escaping characters: use `\` on `\, `, $, ", ', #`

Miscellany

- Escaping characters: use `\` on `\, `, $, ", ', #`

- `pushd` and `popd` create a stack of directories
- `dirs` lists the stack
- Use these instead of `cd`

Miscellany

- Escaping characters: use `\` on `\, `, $, ", ', #`
- `pushd` and `popd` create a stack of directories
- `dirs` lists the stack
- Use these instead of `cd`
- `set -u` gives an error if you try to use an unset variable.
- `set -x` prints out commands as they are run.

Miscellany

- Escaping characters: use `\` on `` \, `, $, ", ', # ``
- `pushd` and `popd` create a stack of directories
- `dirs` lists the stack
- Use these instead of `cd`
- `set -u` gives an error if you try to use an unset variable.
- `set -x` prints out commands as they are run.
- `help COMMAND` gives you help with builtins.