Bash conditionals and loops

ComS 252 — Iowa State University

Andrew Miner

Using shell logic

- We have already seen "shell logic" using `&&` and `||`
- We can implement a conditional in shell logic, e.g
  if (`cmdA` succeeds) then (run `cmdB`) else (run `cmdC`):

Using shell logic

- We have already seen “shell logic” using `&&` and `||`
- We can implement a conditional in shell logic, e.g

  if (`cmdA` succeeds) then (run `cmdB`) else (run `cmdC`):

```
(cmdA && cmdB) || cmdC
```

Using shell logic

- We have already seen "shell logic" using `&&` and `||`
- We can implement a conditional in shell logic, e.g

  if (`cmdA` succeeds) then (run `cmdB`) else (run `cmdC`):

```
(cmdA && cmdB) || cmdC
```

- Some people use this in shell scripts

Using shell logic

- We have already seen "shell logic" using `&&` and `||`
- We can implement a conditional in shell logic, e.g

  if (`cmdA` succeeds) then (run `cmdB`) else (run `cmdC`):

```
(cmdA && cmdB) || cmdC
```

- Some people use this in shell scripts
- There are also "proper" `if` statements . . .

bash if statement

If–then syntax

```
if command1 args
then command2
command3
command4
fi
```

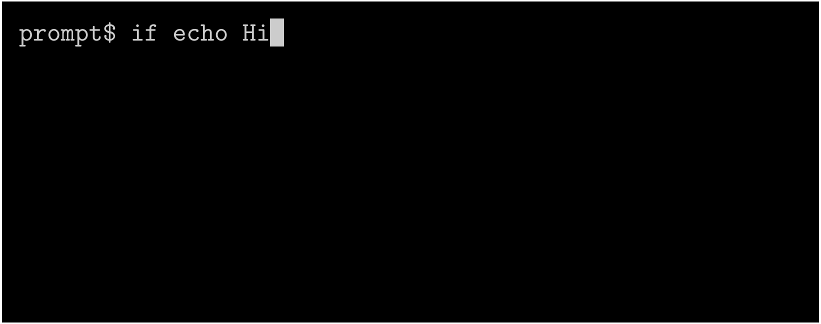
- There must be a newline before "`then`"
  - Or a semicolon
  - Otherwise the shell thinks "`then`" is an argument
- If `command1 args` executes successfully, then

  all commands between "`then`" and "`fi`" will execute
- Otherwise, execution jumps to command following "`fi`"

Simple `if` examples

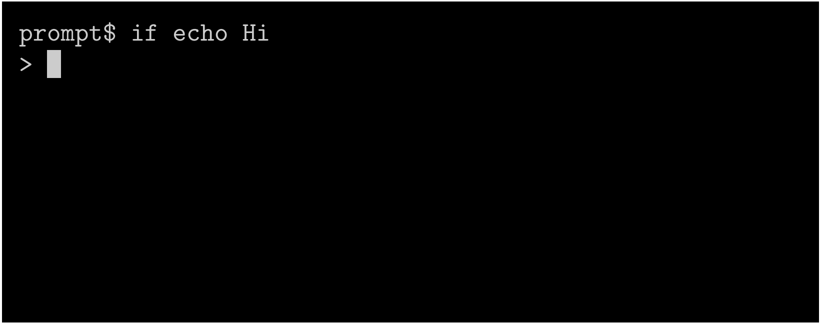
```
prompt$ █
```

Simple if examples



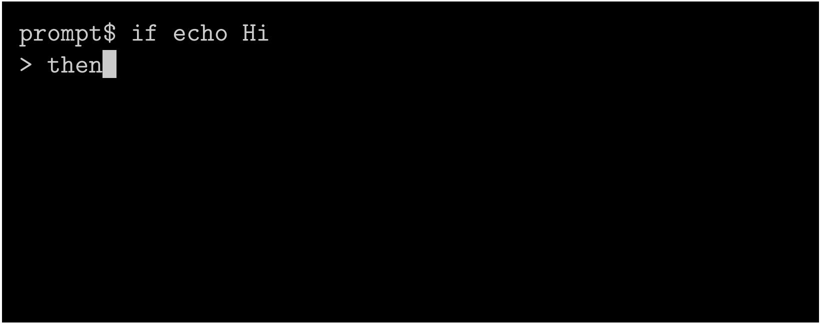
```
prompt$ if echo Hi
```

Simple `if` examples



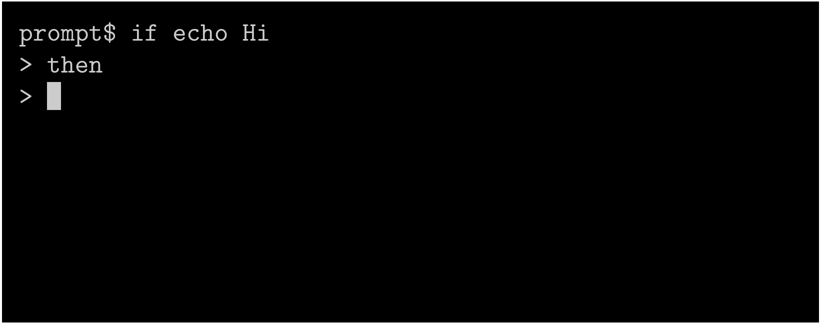
```
prompt$ if echo Hi
> 
```

Simple `if` examples



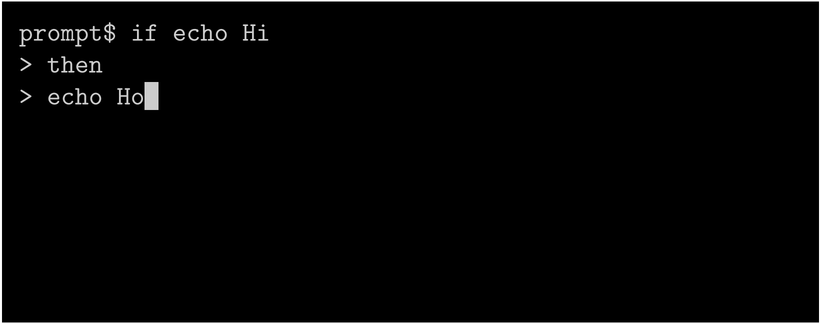
```
prompt$ if echo Hi
> then
```

Simple `if` examples



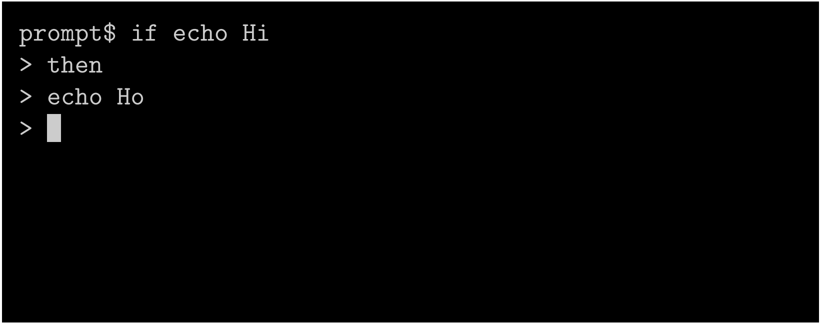
```
prompt$ if echo Hi
> then
> 
```

Simple if examples



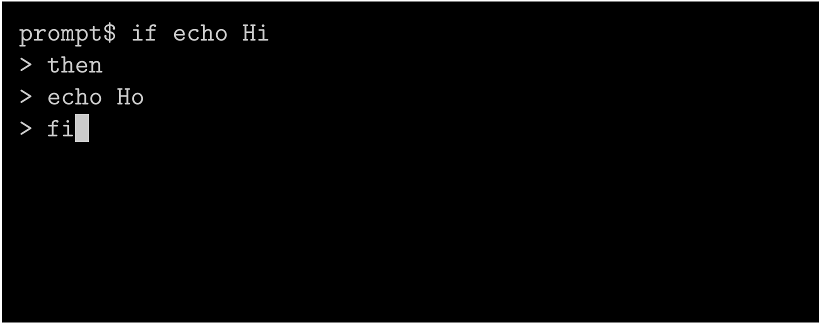
```
prompt$ if echo Hi
> then
> echo Ho
```

Simple if examples



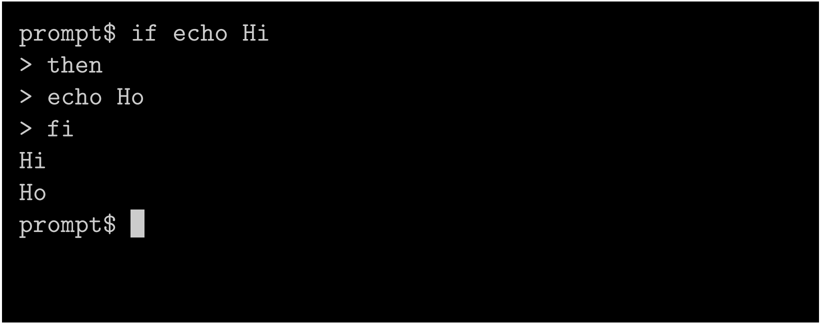
```
prompt$ if echo Hi
> then
> echo Ho
> 
```

Simple if examples



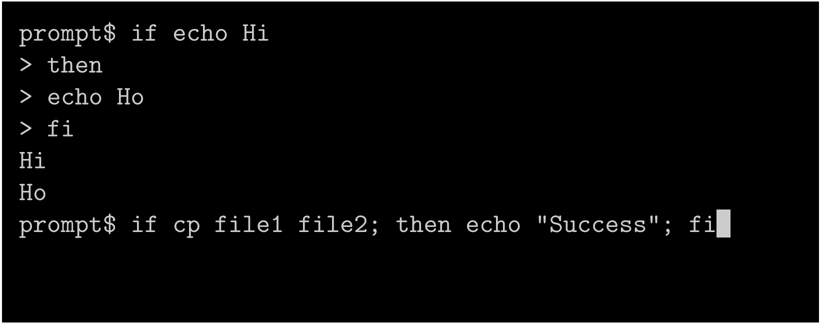
```
prompt$ if echo Hi
> then
> echo Ho
> fi
```

Simple `if` examples



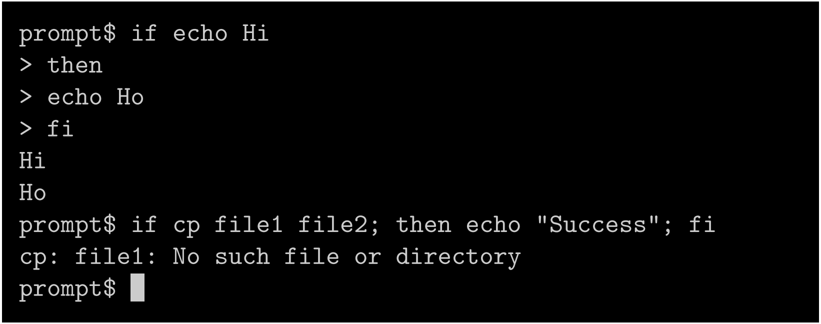
```
prompt$ if echo Hi
> then
> echo Ho
> fi
Hi
Ho
prompt$ █
```

Simple if examples



```
prompt$ if echo Hi
> then
> echo Ho
> fi
Hi
Ho
prompt$ if cp file1 file2; then echo "Success"; fi
```

Simple if examples



```
prompt$ if echo Hi
> then
> echo Ho
> fi
Hi
Ho
prompt$ if cp file1 file2; then echo "Success"; fi
cp: file1: No such file or directory
prompt$ 
```

bash if–then–else

If–then–else syntax

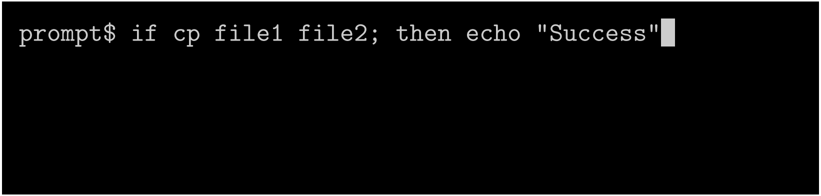
```
if command
then
  commands to run on success
else
  commands to run on failure
fi
```

```
prompt$
```

bash if–then–else

If–then–else syntax

```
if command
then
  commands to run on success
else
  commands to run on failure
fi
```



```
prompt$ if cp file1 file2; then echo "Success"
```

bash if–then–else

If–then–else syntax

```
if command
then
  commands to run on success
else
  commands to run on failure
fi
```

```
prompt$ if cp file1 file2; then echo "Success"
> █
```

bash if–then–else

If–then–else syntax

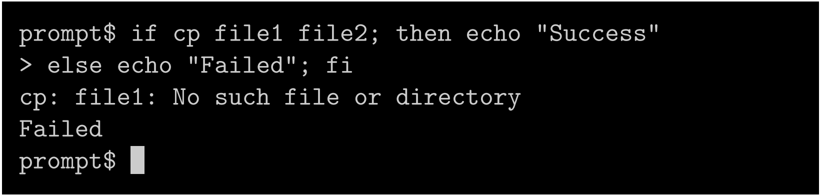
```
if command
then
  commands to run on success
else
  commands to run on failure
fi
```

```
prompt$ if cp file1 file2; then echo "Success"
> else echo "Failed"; fi
```

bash if–then–else

If–then–else syntax

```
if command
then
  commands to run on success
else
  commands to run on failure
fi
```



```
prompt$ if cp file1 file2; then echo "Success"
> else echo "Failed"; fi
cp: file1: No such file or directory
Failed
prompt$ █
```

A sequence of tests

```
if command1
then
  cmds to run on success
else
  if command2
  then
    cmds when command1 fails and command2 succeeds
  else
    if command3
    then
      ⋮
    fi
  fi
fi
```

A sequence of tests: alternate syntax

```
if command1
then
  cmds to run on success
elif command2
then
  cmds when command1 fails and command2 succeeds
elif command3
then
  ⋮
else
  cmds when all fail
fi
```

Comparisons and other "proper" tests

- Shell conditionals are based on process exit status
- What about "programming style" tests?
  - E.g., check if two variables are equal

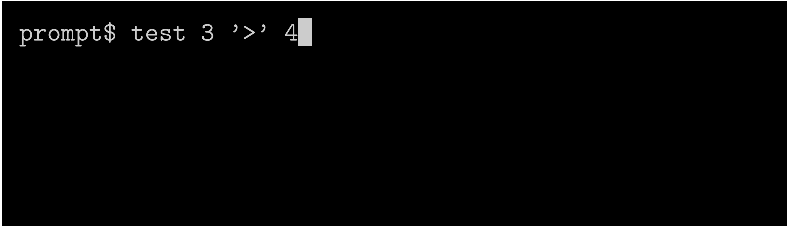
Comparisons and other “proper” tests

- Shell conditionals are based on process exit status
- What about “programming style” tests?
  - E.g., check if two variables are equal
- There is a utility for that: `test`
  - Analogous to “`expr`” for expressions
  - Returns “successful” exit status for true statements

```
prompt$ █
```

Comparisons and other “proper” tests

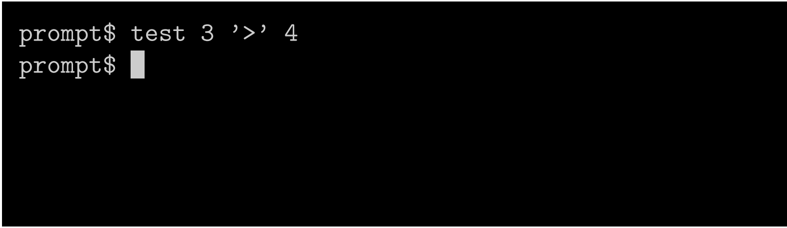
- Shell conditionals are based on process exit status
- What about “programming style” tests?
  - E.g., check if two variables are equal
- There is a utility for that: `test`
  - Analogous to “`expr`” for expressions
  - Returns “successful” exit status for true statements



```
prompt$ test 3 ’>’ 4
```

Comparisons and other “proper” tests

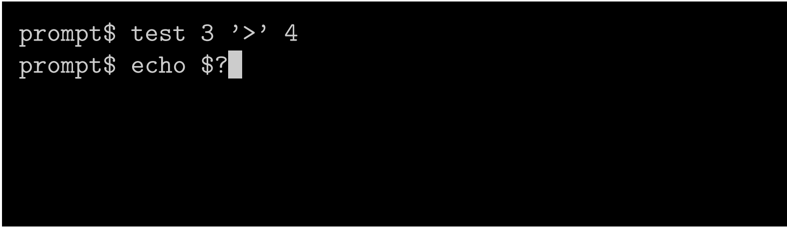
- Shell conditionals are based on process exit status
- What about “programming style” tests?
  - E.g., check if two variables are equal
- There is a utility for that: `test`
  - Analogous to “`expr`” for expressions
  - Returns “successful” exit status for true statements



```
prompt$ test 3 ’>’ 4
prompt$ █
```

Comparisons and other “proper” tests

- Shell conditionals are based on process exit status
- What about “programming style” tests?
  - E.g., check if two variables are equal
- There is a utility for that: `test`
  - Analogous to “`expr`” for expressions
  - Returns “successful” exit status for true statements



```
prompt$ test 3 ’>’ 4
prompt$ echo $?
```

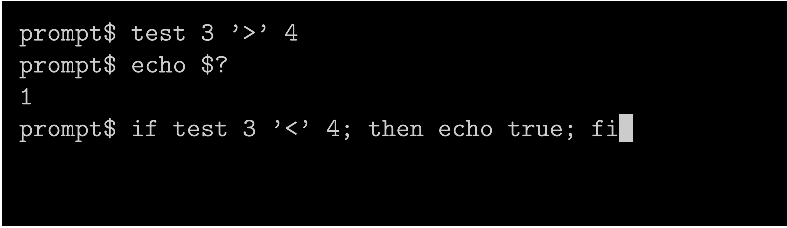
Comparisons and other “proper” tests

- Shell conditionals are based on process exit status
- What about “programming style” tests?
  - E.g., check if two variables are equal
- There is a utility for that: `test`
  - Analogous to “`expr`” for expressions
  - Returns “successful” exit status for true statements

```
prompt$ test 3 '>' 4
prompt$ echo $?
1
prompt$ 
```

Comparisons and other “proper” tests

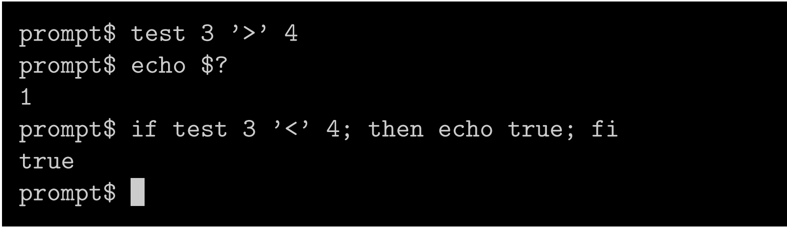
- Shell conditionals are based on process exit status
- What about “programming style” tests?
  - E.g., check if two variables are equal
- There is a utility for that: `test`
  - Analogous to “`expr`” for expressions
  - Returns “successful” exit status for true statements



```
prompt$ test 3 '>' 4
prompt$ echo $?
1
prompt$ if test 3 '<' 4; then echo true; fi
```

Comparisons and other “proper” tests

- Shell conditionals are based on process exit status
- What about “programming style” tests?
  - E.g., check if two variables are equal
- There is a utility for that: `test`
  - Analogous to “`expr`” for expressions
  - Returns “successful” exit status for true statements



```
prompt$ test 3 '>' 4
prompt$ echo $?
1
prompt$ if test 3 '<' 4; then echo true; fi
true
prompt$ █
```

Shorthand for test

```
test arg1 arg2 arg3 ...argn
```

is equivalent to

```
[ arg1 arg2 arg3 ...argn ]
```

- The spaces are important
- Not as nice as “$[...]” environment
- Can be used anywhere we need a process exit status

Shorthand for test

```
test arg1 arg2 arg3 ...argn
```

is equivalent to

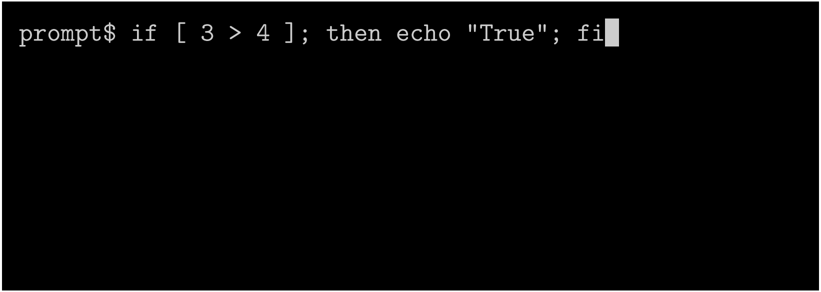
```
[ arg1 arg2 arg3 ...argn ]
```

- The spaces are important
- Not as nice as “$[...]” environment
- Can be used anywhere we need a process exit status
- Fun fact: there is no shell magic here
    - There is an executable file named “[”
        - Try “`which [`”
    - Sometimes, it is a link to “`test`”

Simple example

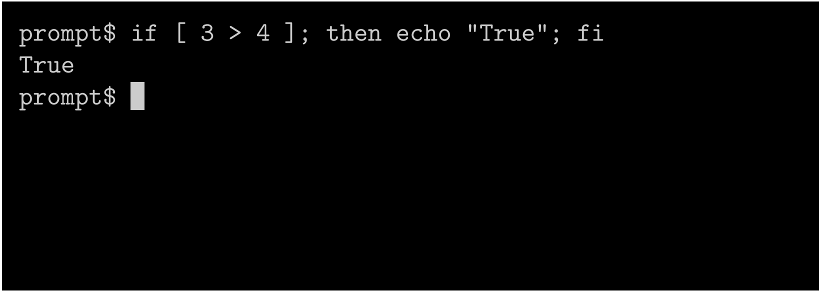
```
prompt$ █
```

Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
```

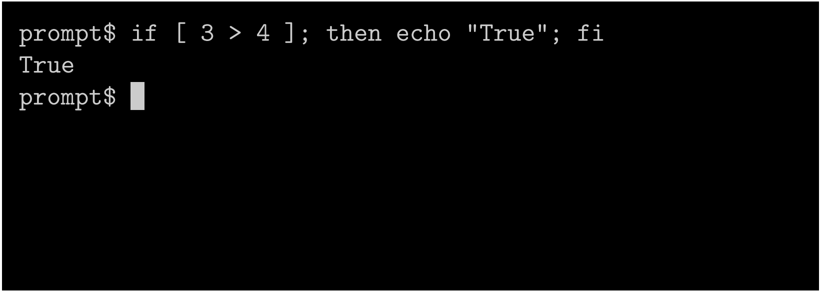
Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ █
```

- What happened?

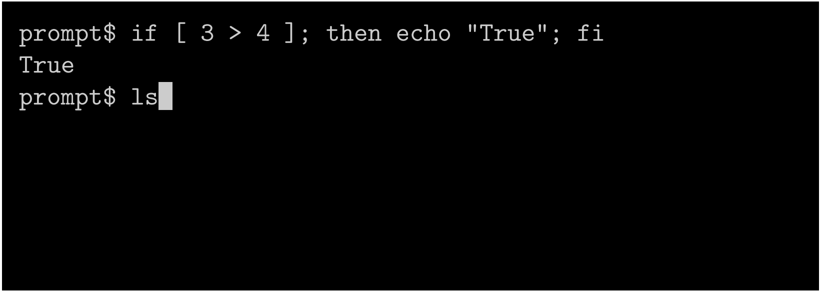
Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ █
```

- What happened?
  - “>” still means redirection inside “[ ]”
  - And “`test 3`” gives a successful exit status of 0

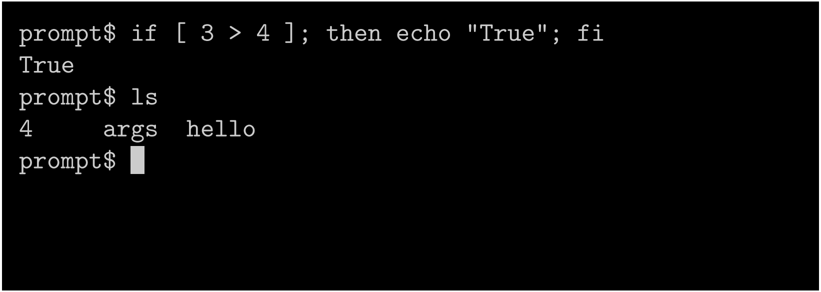
Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ ls
```

- What happened?
  - “>” still means redirection inside “[ ]”
  - And “`test 3`” gives a successful exit status of 0

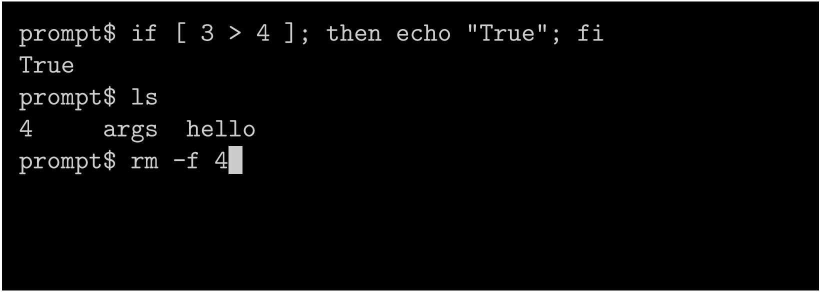
Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ ls
4     args  hello
prompt$ █
```

- What happened?
  - “>” still means redirection inside “[ ]”
  - And “`test 3`” gives a successful exit status of 0

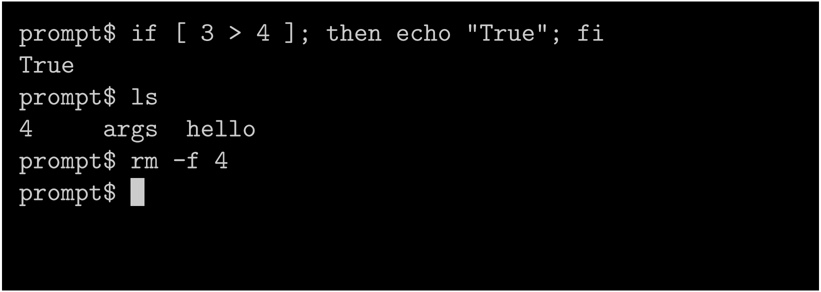
Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ ls
4      args  hello
prompt$ rm -f 4
```

- What happened?
  - “>” still means redirection inside “[ ]”
  - And “`test 3`” gives a successful exit status of 0

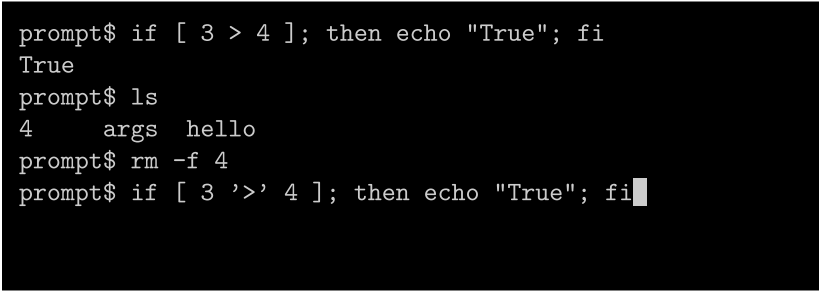
Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ ls
4      args  hello
prompt$ rm -f 4
prompt$ █
```

- What happened?
  - “>” still means redirection inside “[ ]”
  - And “`test 3`” gives a successful exit status of 0

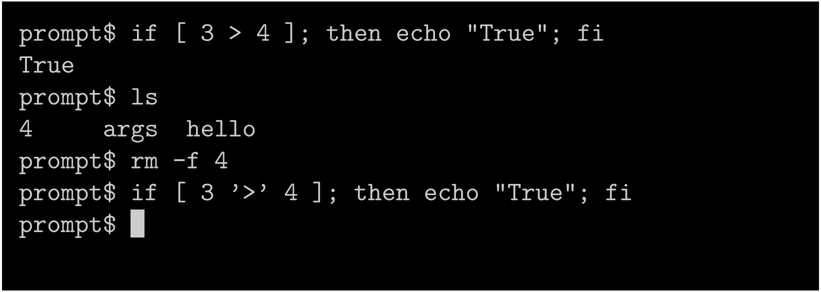
Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ ls
4      args  hello
prompt$ rm -f 4
prompt$ if [ 3 '>' 4 ]; then echo "True"; fi
```

- What happened?
  - ">" still means redirection inside "[ ]"
  - And "`test 3`" gives a successful exit status of 0

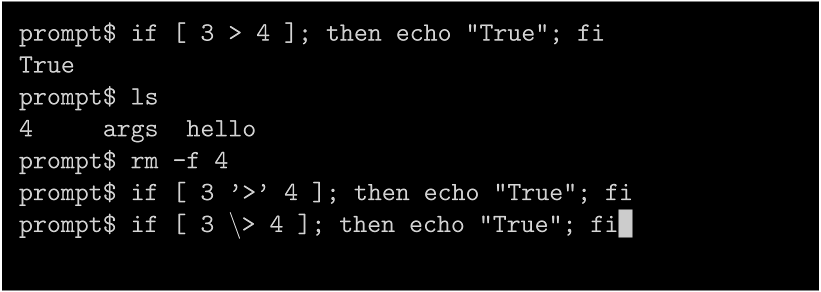
Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ ls
4      args  hello
prompt$ rm -f 4
prompt$ if [ 3 ’>’ 4 ]; then echo "True"; fi
prompt$
```

- What happened?
  - “>” still means redirection inside “[ ]”
  - And “`test 3`” gives a successful exit status of 0

Simple example



```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ ls
4      args  hello
prompt$ rm -f 4
prompt$ if [ 3 '>' 4 ]; then echo "True"; fi
prompt$ if [ 3 \> 4 ]; then echo "True"; fi
```

- What happened?
  - “>” still means redirection inside “[ ]”
  - And “`test 3`” gives a successful exit status of 0

Simple example

```
prompt$ if [ 3 > 4 ]; then echo "True"; fi
True
prompt$ ls
4     args  hello
prompt$ rm -f 4
prompt$ if [ 3 ’>’ 4 ]; then echo "True"; fi
prompt$ if [ 3 \> 4 ]; then echo "True"; fi
prompt$ █
```

- What happened?
  - “>” still means redirection inside “[ ]”
  - And “`test 3`” gives a successful exit status of 0
- Remember to quote or escape the special characters

Using `test`

What can we do with `test`?

- Lots of things
- I will only cover a few useful things
- Read the `man` pages for `test` for more details

Using `test`

What can we do with `test`?

- Lots of things
- I will only cover a few useful things
- Read the `man` pages for `test` for more details

Remember

- Everything in the shell is a string
- Strings that are numbers sometimes need special treatment
  - Assuming you want to treat them as numbers

String comparisons

`=` : Check for equality

- More precisely:
  “`test str1 = str2`” exists successfully if and only if strings `str1` and `str2` are equal

`==` : Check for equality, same as =

`!=` : Check for inequality

`<` : Checks less than

- Uses “alphabetical” ordering

`<=` : Checks less or equal

`>` : Checks greater than

`>=` : Checks less or equal

Example: prompt for name, unless given as arguments

Example: prompt for name, unless given as arguments

hello3

```
#!/bin/bash
if [ "$#" = "0" ]; then
  read -p "What is your name?  " name
else
  name="$@"
fi
printf 'Hello, %s!\n' "$name"
```

Example: special greeting by name

Example: special greeting by name

hello4

```
#!/bin/bash
read -p "What is your name?  " name
if [ "$name" = "Voltron" ]; then
  echo "Hello Voltron, defender of the universe"
elif [ "$name" = "Megatron" ]; then
  echo "All hail Megatron, leader of the Decepticons"
elif [ "$name" = "She-Ra" ]; then
  echo "Hello She-Ra: Princess of Power"
else
  echo "Hello $name"
fi
```

Example: integer comparison

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ █
```

Example: integer comparison

intcmp

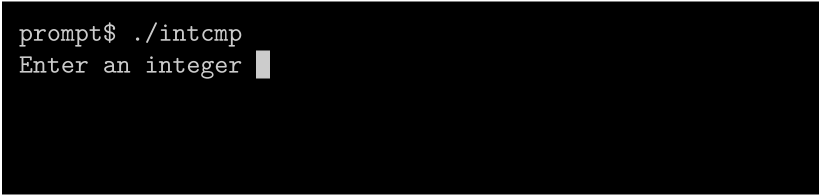
```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ ./intcmp
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```



```
prompt$ ./intcmp
Enter an integer █
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ ./intcmp
Enter an integer 1
```

Example: integer comparison

intcmp

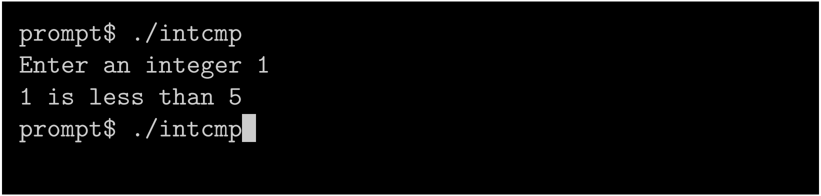
```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ ./intcmp
Enter an integer 1
1 is less than 5
prompt$ █
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```



```
prompt$ ./intcmp
Enter an integer 1
1 is less than 5
prompt$ ./intcmp
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ ./intcmp
Enter an integer 1
1 is less than 5
prompt$ ./intcmp
Enter an integer █
```

Example: integer comparison

intcmp

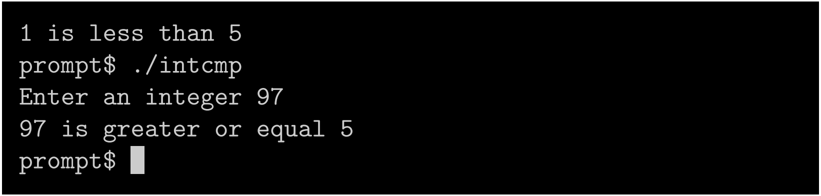
```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" '<' '5' ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ ./intcmp
Enter an integer 1
1 is less than 5
prompt$ ./intcmp
Enter an integer 97
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

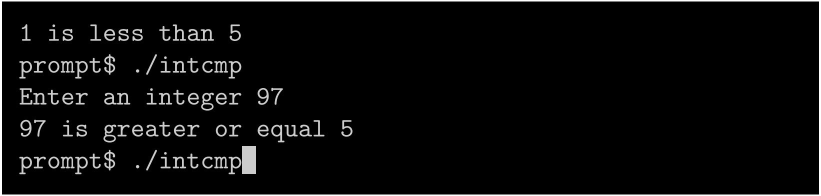


```
1 is less than 5
prompt$ ./intcmp
Enter an integer 97
97 is greater or equal 5
prompt$ 
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

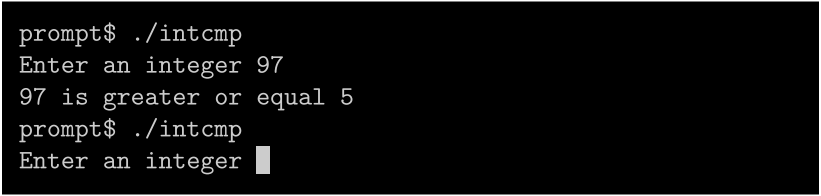


```
1 is less than 5
prompt$ ./intcmp
Enter an integer 97
97 is greater or equal 5
prompt$ ./intcmp
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" '<' '5' ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

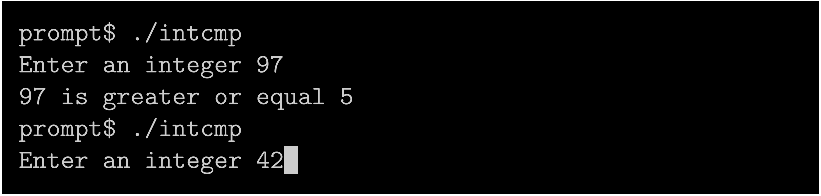


```
prompt$ ./intcmp
Enter an integer 97
97 is greater or equal 5
prompt$ ./intcmp
Enter an integer 
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```



```
prompt$ ./intcmp
Enter an integer 97
97 is greater or equal 5
prompt$ ./intcmp
Enter an integer 42
```

Example: integer comparison

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
97 is greater or equal 5
prompt$ ./intcmp
Enter an integer 42
42 is less than 5
prompt$ █
```

String comparisons on numbers

- May not give the intended result
- Remember, `test` uses alphabetical ordering
  - In alphabetical order, we have $1 < 12 < 123456 < 2 < 204 < 25 < \dots$
- How to compare in numerical order?
  - Do not use >, <, =, etc.
- Wait, what is wrong with =?
  - Strings `"3"` and `"003"` are not equal
  - Numbers 3 and 003 are equal

Comparing numbers with `test`

- `-eq` : Check numerical equality (replaces =)
  - More precisely:
    “`test str1 -eq str2`” exists successfully if and only if strings `str1` and `str2` are equal when interpreted as numbers
- `-ne` : Check numerical inequality (replaces `!=`)
- `-lt` : Check numerical less than (replaces <)
- `-le` : Check numerical less or equal (replaces <=)
- `-gt` : Check numerical greater than (replaces >)
- `-ge` : Check numerical greater or equal (replaces >=)

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" ’<’ ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

Fixing the integer comparison example

intcmp

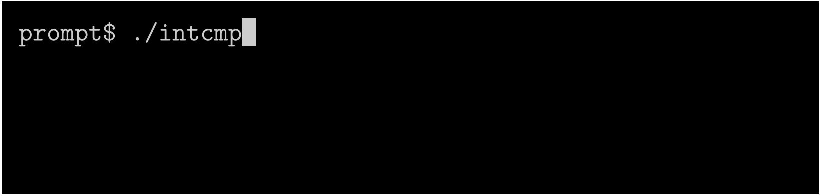
```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ █
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

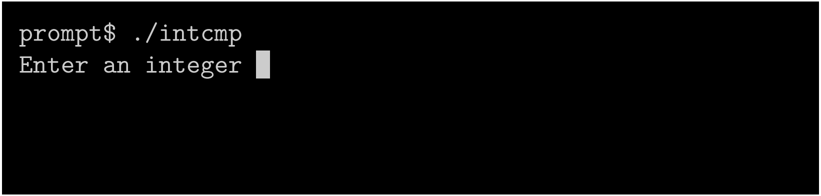


```
prompt$ ./intcmp
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```



```
prompt$ ./intcmp
Enter an integer
```

Fixing the integer comparison example

intcmp

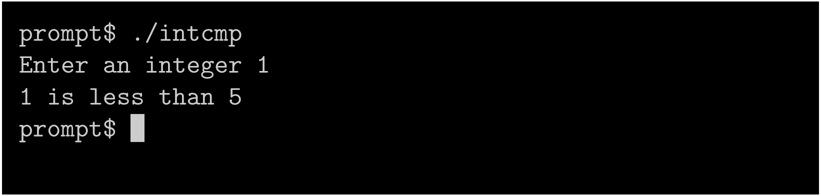
```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ ./intcmp
Enter an integer 1
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

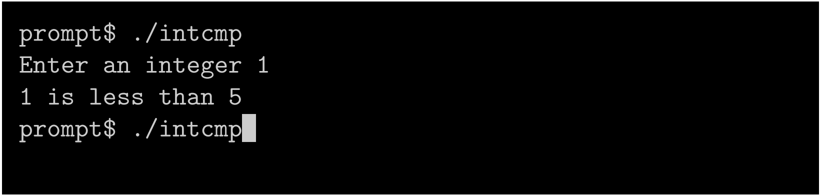


```
prompt$ ./intcmp
Enter an integer 1
1 is less than 5
prompt$ █
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```



```
prompt$ ./intcmp
Enter an integer 1
1 is less than 5
prompt$ ./intcmp
```

Fixing the integer comparison example

intcmp

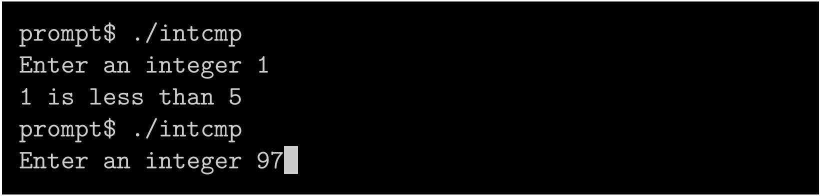
```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ ./intcmp
Enter an integer 1
1 is less than 5
prompt$ ./intcmp
Enter an integer █
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt '5' ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

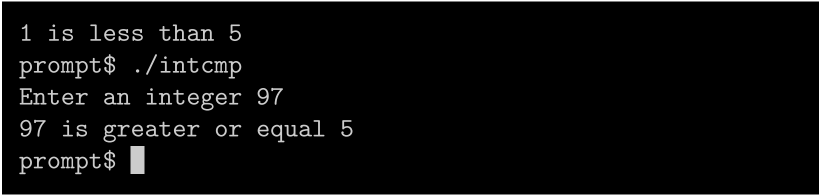


```
prompt$ ./intcmp
Enter an integer 1
1 is less than 5
prompt$ ./intcmp
Enter an integer 97
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```



```
1 is less than 5
prompt$ ./intcmp
Enter an integer 97
97 is greater or equal 5
prompt$ █
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
1 is less than 5
prompt$ ./intcmp
Enter an integer 97
97 is greater or equal 5
prompt$ ./intcmp
```

Fixing the integer comparison example

intcmp

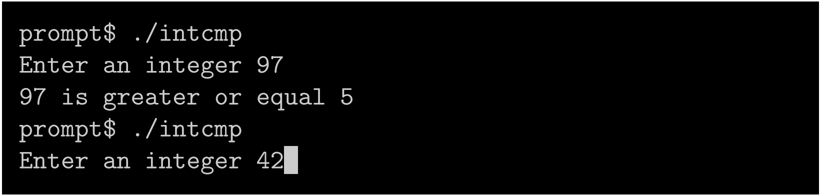
```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
prompt$ ./intcmp
Enter an integer 97
97 is greater or equal 5
prompt$ ./intcmp
Enter an integer █
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```



```
prompt$ ./intcmp
Enter an integer 97
97 is greater or equal 5
prompt$ ./intcmp
Enter an integer 42
```

Fixing the integer comparison example

intcmp

```
#!/bin/bash
read -p "Enter an integer " i
if [ "$i" -lt ’5’ ]; then
  echo "$i is less than 5"
else
  echo "$i is greater or equal 5"
fi
```

```
97 is greater or equal 5
prompt$ ./intcmp
Enter an integer 42
42 is greater or equal 5
prompt$ █
```

File tests

We can answer questions about files using `test`:

- `-d file` : Check if `file` exists, and is a directory
- `-e file` : Check if `file` exists (any type)
- `-f file` : Check if `file` exists, and is a regular file
- `-r file` : Check if `file` exists, and we can read it
- `-w file` : Check if `file` exists, and we can write it
- `-x file` : Check if `file` exists, and we can execute it

Example: what can we execute

canX

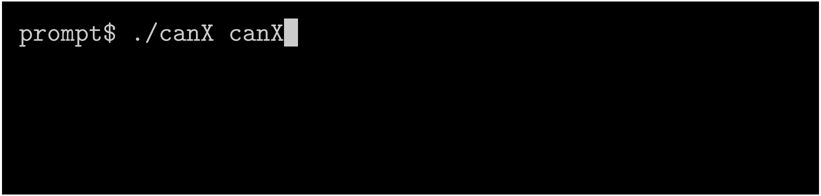
```
#!/bin/bash
if [ -x $1 ]; then
  echo "We can execute $1"
else
  echo "We cannot execute $1"
fi
```

```
prompt$ █
```

Example: what can we execute

canX

```
#!/bin/bash
if [ -x $1 ]; then
  echo "We can execute $1"
else
  echo "We cannot execute $1"
fi
```

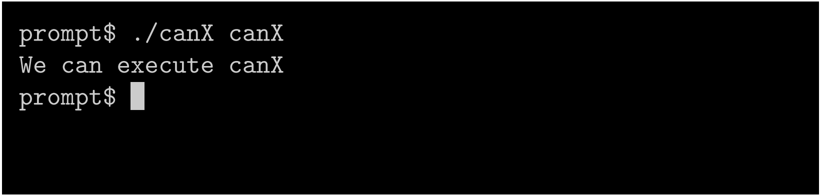


```
prompt$ ./canX canX
```

Example: what can we execute

canX

```
#!/bin/bash
if [ -x $1 ]; then
  echo "We can execute $1"
else
  echo "We cannot execute $1"
fi
```



```
prompt$ ./canX canX
We can execute canX
prompt$ █
```

Example: what can we execute

canX

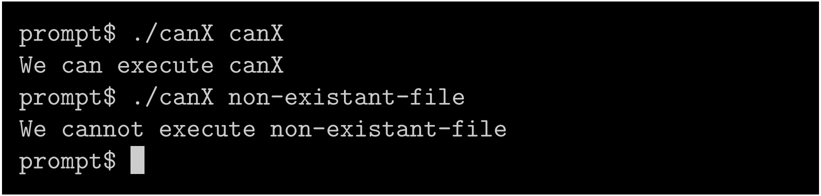
```
#!/bin/bash
if [ -x $1 ]; then
  echo "We can execute $1"
else
  echo "We cannot execute $1"
fi
```

```
prompt$ ./canX canX
We can execute canX
prompt$ ./canX non-existant-file
```

Example: what can we execute

canX

```
#!/bin/bash
if [ -x $1 ]; then
  echo "We can execute $1"
else
  echo "We cannot execute $1"
fi
```

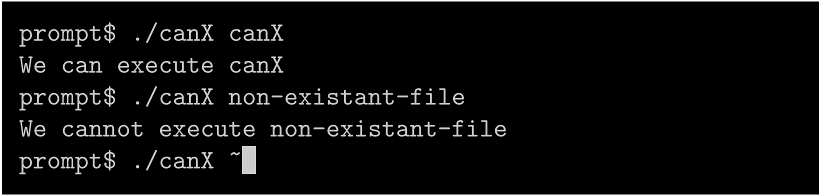


```
prompt$ ./canX canX
We can execute canX
prompt$ ./canX non-existant-file
We cannot execute non-existant-file
prompt$ 
```

Example: what can we execute

canX

```
#!/bin/bash
if [ -x $1 ]; then
  echo "We can execute $1"
else
  echo "We cannot execute $1"
fi
```



```
prompt$ ./canX canX
We can execute canX
prompt$ ./canX non-existant-file
We cannot execute non-existant-file
prompt$ ./canX ~
```

Example: what can we execute

canX

```
#!/bin/bash
if [ -x $1 ]; then
  echo "We can execute $1"
else
  echo "We cannot execute $1"
fi
```

```
prompt$ ./canX non-existant-file
We cannot execute non-existant-file
prompt$ ./canX ~
We can execute /home/alice
prompt$
```

Example: what can we execute

canX

```
#!/bin/bash
if [ -x $1 ]; then
  echo "We can execute $1"
else
  echo "We cannot execute $1"
fi
```

```
prompt$ ./canX non-existant-file
We cannot execute non-existant-file
prompt$ ./canX ~
We can execute /home/alice
prompt$ ./canX ~bob
```

Example: what can we execute

canX

```
#!/bin/bash
if [ -x $1 ]; then
  echo "We can execute $1"
else
  echo "We cannot execute $1"
fi
```

```
prompt$ ./canX ~
We can execute /home/alice
prompt$ ./canX ~bob
We cannot execute /home/bob
prompt$ █
```

Test logic

`! expr` : Negate an expression

`( expr )` : Grouping

`expr1 -a expr2` : Check if `expr1` AND `expr2` holds

`expr1 -o expr2` : Check if `expr1` OR `expr2` holds

Test logic

`! expr` : Negate an expression

`( expr )` : Grouping

`expr1 -a expr2` : Check if `expr1` AND `expr2` holds

`expr1 -o expr2` : Check if `expr1` OR `expr2` holds

Remember to protect characters that mean something to the shell

Loops in bash: “while”

While syntax

```
while command1 args
do command2
command3
command4
done
```

- Remember, we need a newline after each `command`
  - Or, a semicolon
  - Otherwise the shell assumes things are arguments
- If `command1 args` executes successfully, then
  - All commands between “`do`” and “`done`” execute
  - Execution jumps to the beginning (`command1` is executed again)
- Otherwise, execution jumps to command following “`done`”

Example: counting from first to second argument

(With nice error checking for the number of arguments)

Example: counting from first to second argument

(With nice error checking for the number of arguments)

counter

```
#!/bin/bash
if [ $# -ne 2 ]; then
  echo Usage: $0 start stop
  exit 1
fi
N=$1
while [ $N -le $2 ]; do
  echo $N
  N=$[N+1]
done
```

Powers of two

Powers of two

power

```
#!/bin/bash
N=0
P=1
while [ $P -ge 1 ]; do
  echo "2^$N = $P"
  N=$[N+1]
  P=$[2*P]
done
```

Powers of two

power

```
#!/bin/bash
N=0
P=1
while [ $P -ge 1 ]; do
  echo "2^$N = $P"
  N=$[N+1]
  P=$[2*P]
done
```

Will this run forever?

Powers of two

power

```
#!/bin/bash
N=0
P=1
while [ $P -ge 1 ]; do
  echo "2^$N = $P"
  N=$[N+1]
  P=$[2*P]
done
```

Will this run forever?

- Theory: yes
- Practice: no

Cool read trick

Remember read?

- read var: will read a line from stdin, store in var
- read returns a status of 0

Cool `read` trick

Remember `read`?

- `read var`: will read a line from stdin, store in `var`
- `read` returns a status of 0 . . .
- unless we're at the end of file

Cool read trick

Remember read?

- read var: will read a line from stdin, store in var
- read returns a status of 0 . . .
- unless we're at the end of file

We can read all input lines with a loop:

```
while read line; do
...
done
```

Example: read names until EOF, say hello to all

Example: read names until EOF, say hello to all

helloall

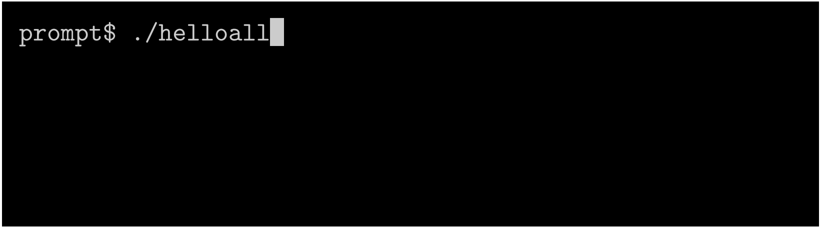
```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```

```
prompt$ █
```

Example: read names until EOF, say hello to all

helloall

```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```

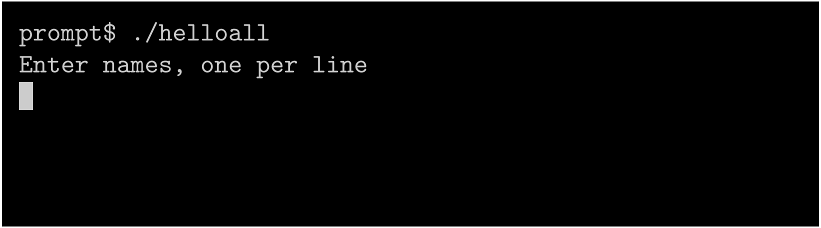


```
prompt$ ./helloall
```

Example: read names until EOF, say hello to all

helloall

```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```



```
prompt$ ./helloall
Enter names, one per line
█
```

Example: read names until EOF, say hello to all

helloall

```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```

```
prompt$ ./helloall
Enter names, one per line
Madonna
```

Example: read names until EOF, say hello to all

helloall

```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```

```
prompt$ ./helloall
Enter names, one per line
Madonna
Hello, Madonna
█
```

Example: read names until EOF, say hello to all

helloall

```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```

```
prompt$ ./helloall
Enter names, one per line
Madonna
Hello, Madonna
Bob Roberts█
```

Example: read names until EOF, say hello to all

helloall

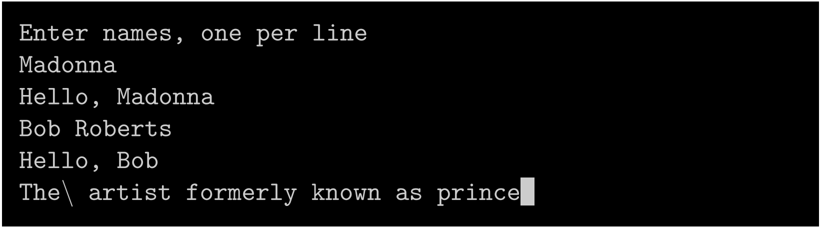
```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```

```
Enter names, one per line
Madonna
Hello, Madonna
Bob Roberts
Hello, Bob
█
```

Example: read names until EOF, say hello to all

helloall

```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```

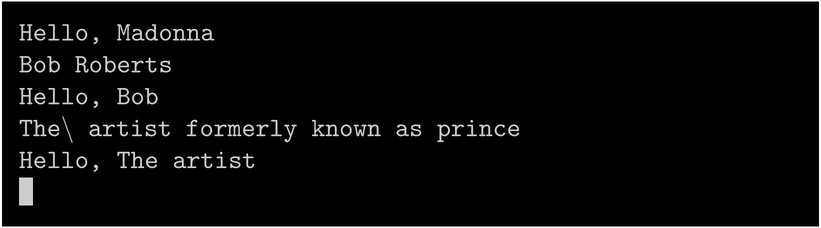


```
Enter names, one per line
Madonna
Hello, Madonna
Bob Roberts
Hello, Bob
The\ artist formerly known as prince
```

Example: read names until EOF, say hello to all

helloall

```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```



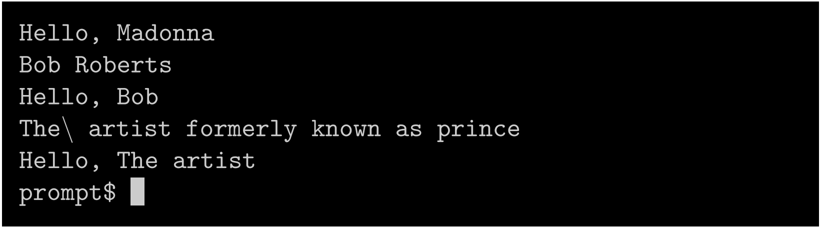
```
Hello, Madonna
Bob Roberts
Hello, Bob
The\ artist formerly known as prince
Hello, The artist
```

(Hit Ctrl-D for end of file)

Example: read names until EOF, say hello to all

helloall

```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```



```
Hello, Madonna
Bob Roberts
Hello, Bob
The\ artist formerly known as prince
Hello, The artist
prompt$ █
```

Example: read names until EOF, say hello to all

helloall

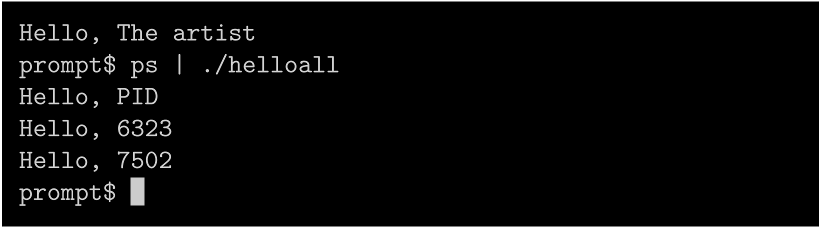
```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```

```
Hello, Madonna
Bob Roberts
Hello, Bob
The\ artist formerly known as prince
Hello, The artist
prompt$ ps | ./helloall
```

Example: read names until EOF, say hello to all

helloall

```
#!/bin/bash
echo Enter names, one per line
while read first rest; do
  echo Hello, $first
done
```



```
Hello, The artist
prompt$ ps | ./helloall
Hello, PID
Hello, 6323
Hello, 7502
prompt$
```

Loops in `bash`: "for"

For syntax

```
for var in list of items
do
commands as usual
done
```

- Not your basic `for` loop from C/C++/Java
- Newline characters or semicolons are required as usual

1. Assigns `var` to first item in the list
   and executes commands between `do` and `done`
2. Assigns `var` to second item in the list and executes commands

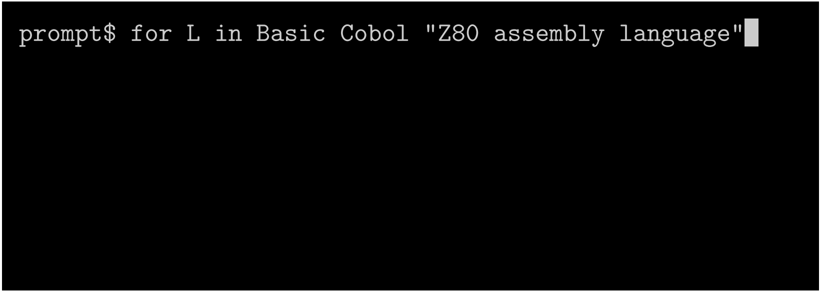
   ⋮

n. Assigns `var` to last item in the list and executes commands

Simple `for` example

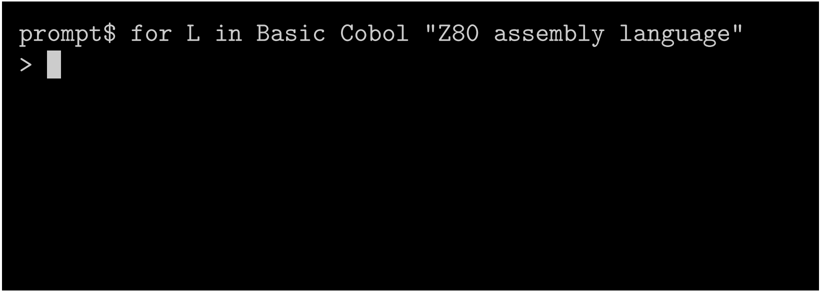
```
prompt$ 
```

Simple for example



```
prompt$ for L in Basic Cobol "Z80 assembly language"
```

Simple for example

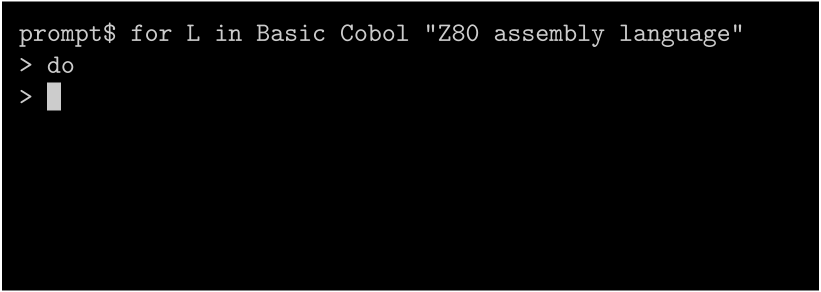


```
prompt$ for L in Basic Cobol "Z80 assembly language"
>
```

Simple for example

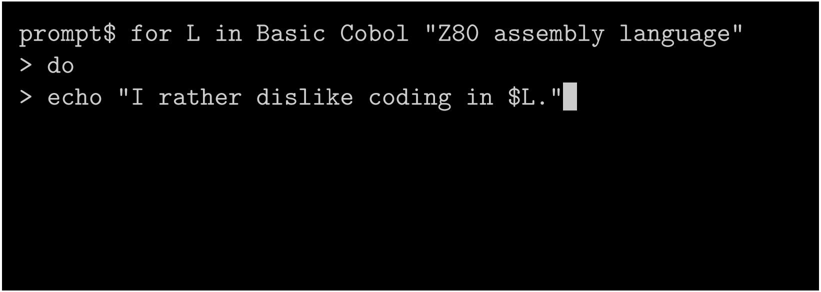
```
prompt$ for L in Basic Cobol "Z80 assembly language"
> do
```

Simple for example



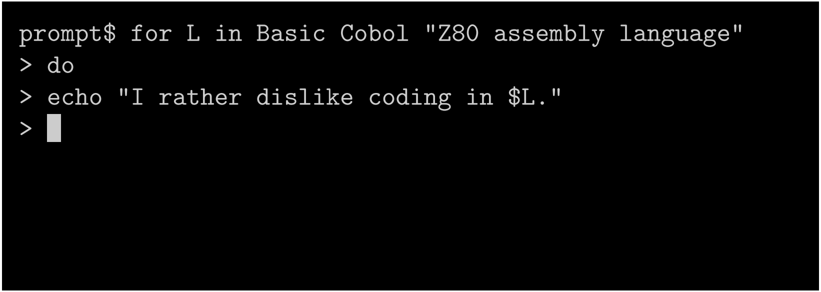
```
prompt$ for L in Basic Cobol "Z80 assembly language"
> do
> 
```

Simple for example



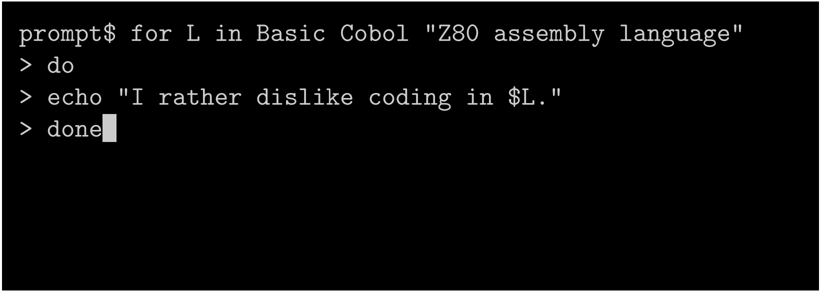
```
prompt$ for L in Basic Cobol "Z80 assembly language"
> do
> echo "I rather dislike coding in $L."
```

Simple for example



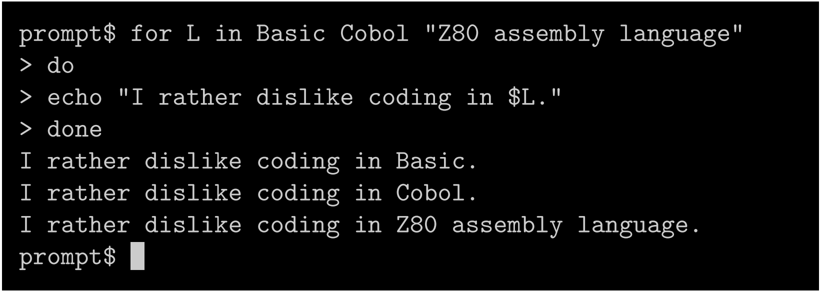
```
prompt$ for L in Basic Cobol "Z80 assembly language"
> do
> echo "I rather dislike coding in $L."
> 
```

Simple for example



```
prompt$ for L in Basic Cobol "Z80 assembly language"
> do
> echo "I rather dislike coding in $L."
> done
```

Simple for example



```
prompt$ for L in Basic Cobol "Z80 assembly language"
> do
> echo "I rather dislike coding in $L."
> done
I rather dislike coding in Basic.
I rather dislike coding in Cobol.
I rather dislike coding in Z80 assembly language.
prompt$ █
```

Ok, time for a quiz

How can I change all my "`.c`" files into "`.cc`" files?

Ok, time for a quiz

How can I change all my “.c” files into “.cc” files?

- `mv *.c *.cc`

Ok, time for a quiz

How can I change all my “`.c`” files into “`.cc`” files?

- `mv *.c *.cc` NO, sorry

Ok, time for a quiz

How can I change all my "`.c`" files into "`.cc`" files?

- `mv *.c *.cc` NO, sorry
- We just learned `for` loops, so that must be how to do it

Ok, time for a quiz

How can I change all my “.c” files into “.cc” files?

- `mv *.c *.cc` NO, sorry
- We just learned `for` loops, so that must be how to do it

```
prompt$ for f in *.c; do mv $f "$f"c; done
```

- I think that will fail if some filename contains spaces
  - But you know better than to put spaces in a filename...

Ok, time for a quiz

How can I change all my "`.c`" files into "`.cc`" files?

- `mv *.c *.cc` NO, sorry
- We just learned `for` loops, so that must be how to do it

```
prompt$ for f in *.c; do mv $f "$f"c; done
```

- I think that will fail if some filename contains spaces
  - But you know better than to put spaces in a filename...

How do I change all my "`.cc`" files back to "`.c`" files?

Ok, time for a quiz

How can I change all my "`.c`" files into "`.cc`" files?

- `mv *.c *.cc` NO, sorry
- We just learned `for` loops, so that must be how to do it

```
prompt$ for f in *.c; do mv $f "$f"c; done
```

- I think that will fail if some filename contains spaces
  - But you know better than to put spaces in a filename...

How do I change all my "`.cc`" files back to "`.c`" files?

- That's more involved
- We need some utilities or other shell tricks to help

How can I loop over the arguments of a script?

How can I loop over the arguments of a script?

Method 1: use a `for` loop

How can I loop over the arguments of a script?

Method 1: use a `for` loop

```
for arg in $@
do
  ...
done
```

Example: arguments are names, say hello to all

Example: arguments are names, say hello to all

helloargs

```
#!/bin/bash
for name in $@; do
  echo Hello, $name
done
```

```
prompt$ █
```

Example: arguments are names, say hello to all

helloargs

```
#!/bin/bash
for name in $@; do
  echo Hello, $name
done
```

```
prompt$ ./helloargs Madonna Prince
```

Example: arguments are names, say hello to all

helloargs

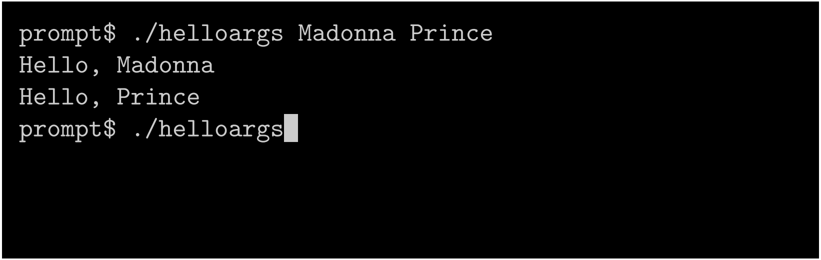
```
#!/bin/bash
for name in $@; do
  echo Hello, $name
done
```

```
prompt$ ./helloargs Madonna Prince
Hello, Madonna
Hello, Prince
prompt$ 
```

Example: arguments are names, say hello to all

helloargs

```
#!/bin/bash
for name in $@; do
  echo Hello, $name
done
```



```
prompt$ ./helloargs Madonna Prince
Hello, Madonna
Hello, Prince
prompt$ ./helloargs
```

Example: arguments are names, say hello to all

helloargs

```
#!/bin/bash
for name in $@; do
  echo Hello, $name
done
```

```
prompt$ ./helloargs Madonna Prince
Hello, Madonna
Hello, Prince
prompt$ ./helloargs
prompt$ 
```

Example: arguments are names, say hello to all

helloargs

```
#!/bin/bash
for name in $@; do
  echo Hello, $name
done
```

```
prompt$ ./helloargs Madonna Prince
Hello, Madonna
Hello, Prince
prompt$ ./helloargs
prompt$ ./helloargs Madonna Prince "Bob Roberts" Bjork
```

Example: arguments are names, say hello to all

helloargs

```
#!/bin/bash
for name in $@; do
  echo Hello, $name
done
```

```
prompt$ ./helloargs Madonna Prince "Bob Roberts" Bjork
Hello, Madonna
Hello, Prince
Hello, Bob
Hello, Roberts
Hello, Bjork
prompt$ █
```

Example: arguments are names, say hello to all

helloargs

```
#!/bin/bash
for name in $@; do
  echo Hello, $name
done
```

```
prompt$ ./helloargs Madonna Prince "Bob Roberts" Bjork
Hello, Madonna
Hello, Prince
Hello, Bob
Hello, Roberts
Hello, Bjork
prompt$ ./helloargs Madonna Prince Bob\ Roberts Bjork
```

Example: arguments are names, say hello to all

helloargs

```
#!/bin/bash
for name in $@; do
  echo Hello, $name
done
```

```
prompt$ ./helloargs Madonna Prince Bob\ Roberts Bjork
Hello, Madonna
Hello, Prince
Hello, Bob
Hello, Roberts
Hello, Bjork
prompt$
```

How can I loop over the arguments of a script?

Method 1: use a `for` loop

```
for arg in $@
do
   ...
done
```

How can I loop over the arguments of a script?

Method 1: use a for loop

```
for arg in $@
do
  ...
done
```

But this will drop any quotes; can we fix that?

How can I loop over the arguments of a script?

Method 1: use a `for` loop

```
for arg in $@
do
   ...
done
```

But this will drop any quotes; can we fix that?

Method 1 improved:

```
for arg
do
   ...
done
```

Let’s fix the `helloargs` script

Let's fix the `helloargs` script

`helloargs`

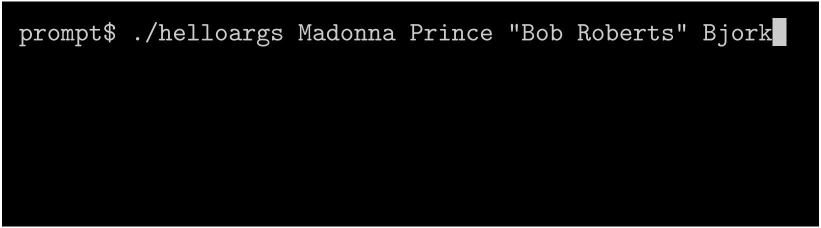
```
#!/bin/bash
for name; do
  echo Hello, $name
done
```

```
prompt$ █
```

Let’s fix the helloargs script

helloargs

```
#!/bin/bash
for name; do
  echo Hello, $name
done
```



```
prompt$ ./helloargs Madonna Prince "Bob Roberts" Bjork
```

Let's fix the `helloargs` script

helloargs

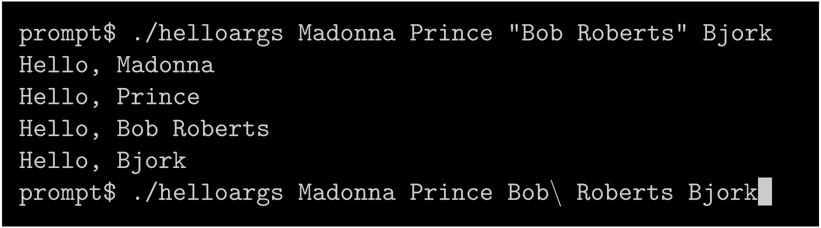
```
#!/bin/bash
for name; do
  echo Hello, $name
done
```

```
prompt$ ./helloargs Madonna Prince "Bob Roberts" Bjork
Hello, Madonna
Hello, Prince
Hello, Bob Roberts
Hello, Bjork
prompt$ █
```

Let's fix the helloargs script

helloargs

```
#!/bin/bash
for name; do
  echo Hello, $name
done
```



```
prompt$ ./helloargs Madonna Prince "Bob Roberts" Bjork
Hello, Madonna
Hello, Prince
Hello, Bob Roberts
Hello, Bjork
prompt$ ./helloargs Madonna Prince Bob\ Roberts Bjork
```

Let’s fix the helloargs script

helloargs

```
#!/bin/bash
for name; do
  echo Hello, $name
done
```

```
prompt$ ./helloargs Madonna Prince Bob\ Roberts Bjork
Hello, Madonna
Hello, Prince
Hello, Bob Roberts
Hello, Bjork
prompt$ █
```

Another way to deal with arguments

`shift`

- Argument 0 remains the same
- All other arguments “shift to the left”
  - Old argument 2 is now argument 1
  - Old argument 3 is now argument 2
  - ...
- The argument count decreases by one
- Works for function and script arguments

How can I loop over arguments of a script?

How can I loop over arguments of a script?

Method 2: use `while` and `shift`

How can I loop over arguments of a script?

Method 2: use `while` and `shift`

```
while [ $# -gt 0 ]
do
  ...
  shift
done
```

Example: arguments are names, say hello to all

Example: arguments are names, say hello to all

helloargs2

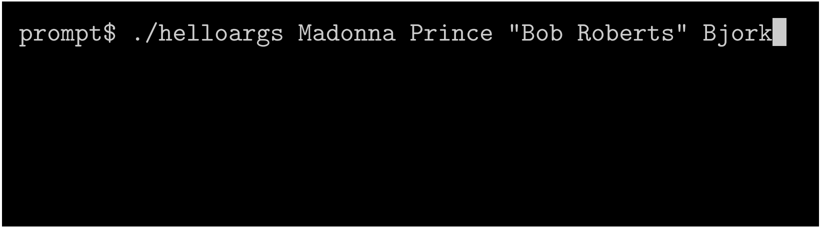
```
#!/bin/bash
while [ $# -gt 0 ]; do
  echo Hello, $1
  shift
done
```

```
prompt$ █
```

Example: arguments are names, say hello to all

helloargs2

```
#!/bin/bash
while [ $# -gt 0 ]; do
  echo Hello, $1
  shift
done
```



```
prompt$ ./helloargs Madonna Prince "Bob Roberts" Bjork
```

Example: arguments are names, say hello to all

helloargs2

```
#!/bin/bash
while [ $# -gt 0 ]; do
  echo Hello, $1
  shift
done
```

```
prompt$ ./helloargs Madonna Prince "Bob Roberts" Bjork
Hello, Madonna
Hello, Prince
Hello, Bob Roberts
Hello, Bjork
prompt$ █
```

Summary: looping over arguments

Method 1: use a `for` loop

```
for arg; do
  ...
done
```

Method 2: use `while` and `shift`

```
while [ $# -gt 0 ]; do
  ...
  shift
done
```

Differences:

- Method 1 does not “consume” arguments
- Method 2 does “consume” arguments

Ways to handle switches

1. “By hand”
2. `getopts` command

Ways to handle switches

1. "By hand"
2. `getopts` command

`getopts`

- Usage: `getopts string-of-switches VAR`
- Only handles short switches
- Can handle switches with arguments
- Details to follow…

`getopts string-of-switches VAR`

`string-of-switches` : string containing switch letters

- Put ":" after letters that take arguments
- Put ":" at the beginning for no error messages

`VAR` :

- If there was a matching switch, gets the letter
- If no matching switch, gets "?"

exit status : Success iff there was a matching switch

`OPTARG` : a "global variable"

- Holds the switch argument, if there was one
- Otherwise, will be the empty string

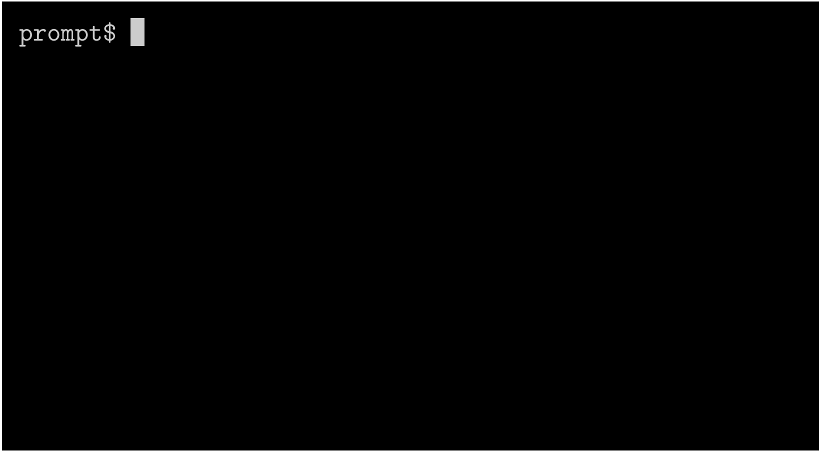
`OPTIND` : another "global variable"

- Position of next parameter to examine

opts

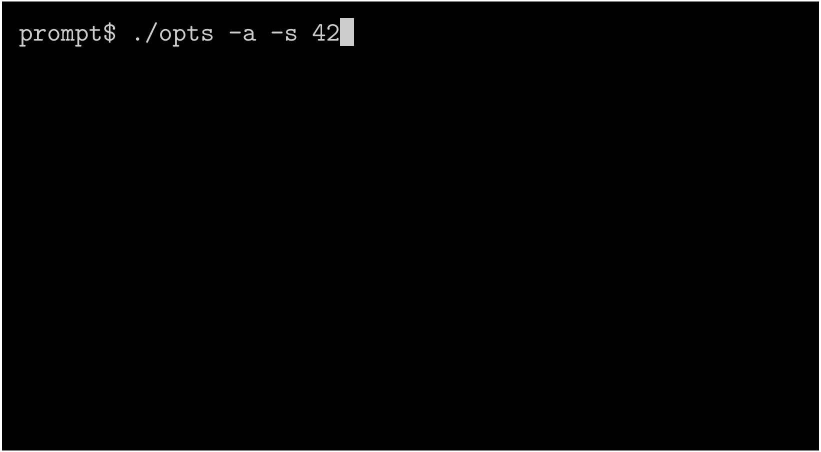
```
#!/bin/bash
while getopts "habs:" SW; do
  if [ "$SW" = "h" ]; then
    echo "Legal switches: -h -a -b -s size"
    exit 1
  elif [ "$SW" = "a" ]; then
    echo "Switch a was set"
  elif [ "$SW" = "b" ]; then
    echo "Switch b was set"
  elif [ "$SW" = "s" ]; then
    echo "Switch s with argument $OPTARG"
  else
    echo "Unknown switch: $SW"
  fi
done
```

Running getopts



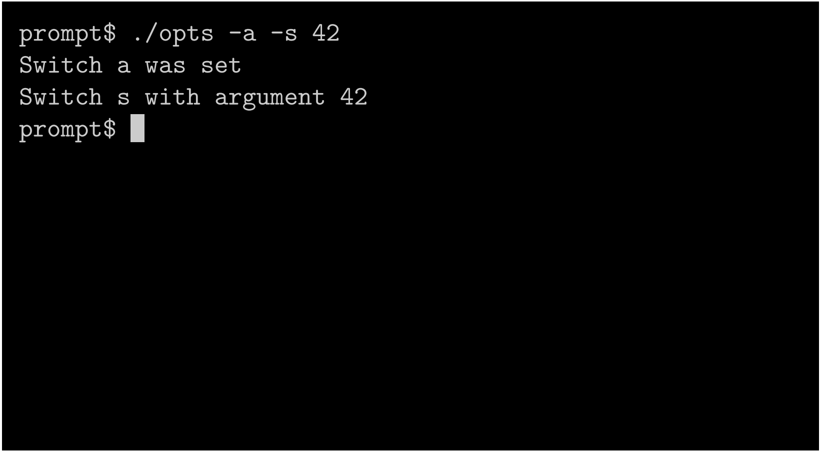
```
prompt$
```

Running getopts



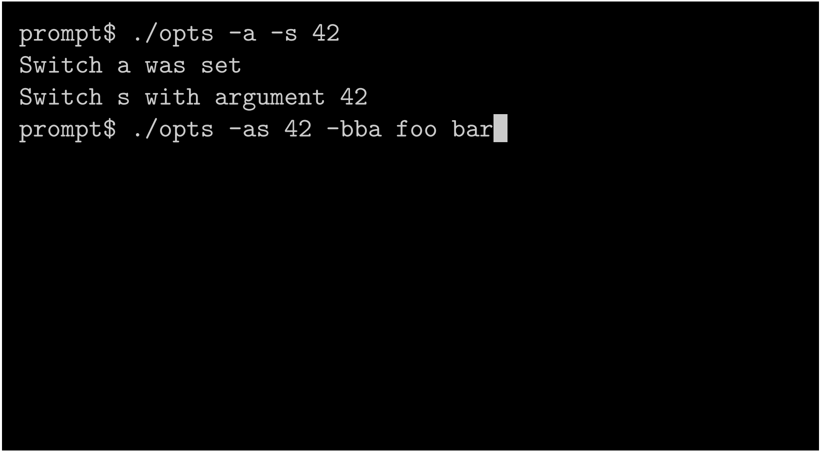
```
prompt$ ./opts -a -s 42
```

Running getopts



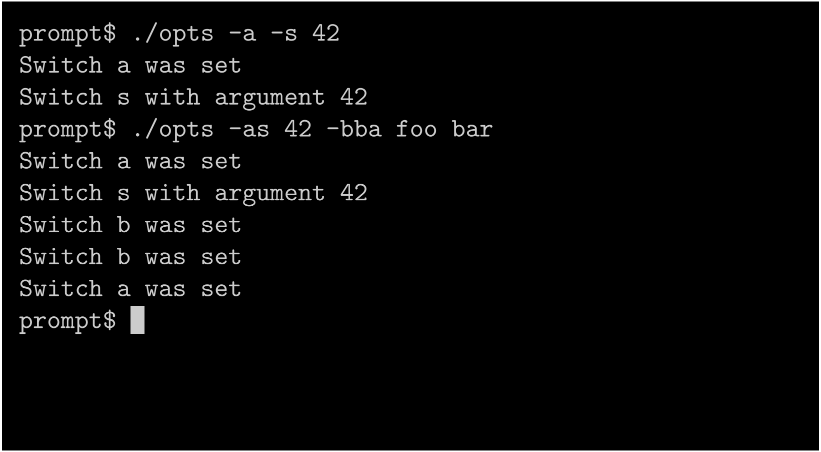
```
prompt$ ./opts -a -s 42
Switch a was set
Switch s with argument 42
prompt$ █
```

Running getopts



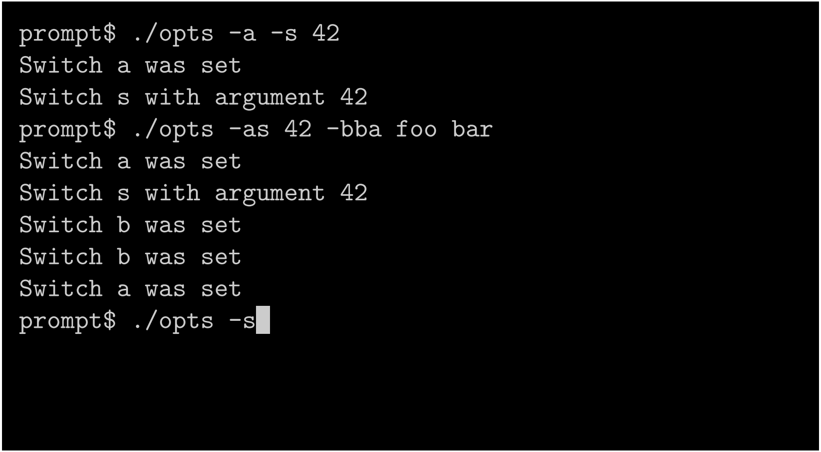
```
prompt$ ./opts -a -s 42
Switch a was set
Switch s with argument 42
prompt$ ./opts -as 42 -bba foo bar
```

Running getopts



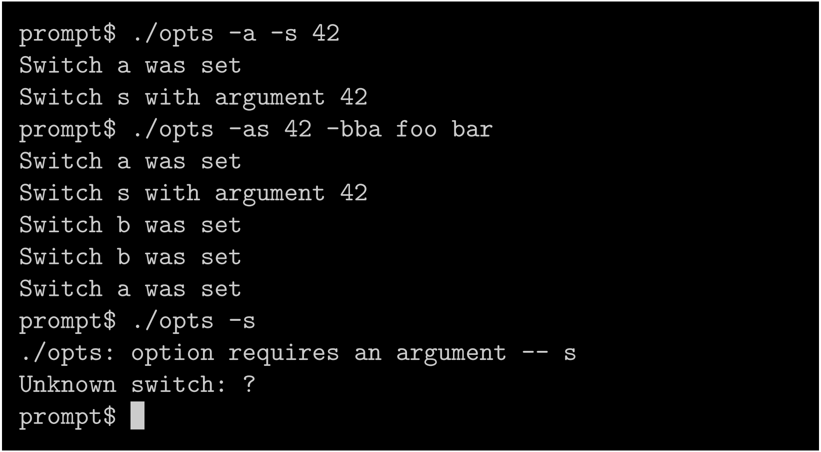
```
prompt$ ./opts -a -s 42
Switch a was set
Switch s with argument 42
prompt$ ./opts -as 42 -bba foo bar
Switch a was set
Switch s with argument 42
Switch b was set
Switch b was set
Switch a was set
prompt$
```

Running getopts



```
prompt$ ./opts -a -s 42
Switch a was set
Switch s with argument 42
prompt$ ./opts -as 42 -bba foo bar
Switch a was set
Switch s with argument 42
Switch b was set
Switch b was set
Switch a was set
prompt$ ./opts -s
```

Running getopts



```
prompt$ ./opts -a -s 42
Switch a was set
Switch s with argument 42
prompt$ ./opts -as 42 -bba foo bar
Switch a was set
Switch s with argument 42
Switch b was set
Switch b was set
Switch a was set
prompt$ ./opts -s
./opts: option requires an argument -- s
Unknown switch: ?
prompt$
```

Case statement

- Similar to "switch" construct in C
- Operates on strings (of course)
- Can use globbing to match case labels
- Cases do not "fall through", like C

Case syntax

```
case string in
pattern) list; of; statements ;;
⋮
pattern) list; of; statements ;;
esac
```

Let's rewrite `hello4` using case

`hello4`

```
#!/bin/bash
read -p "What is your name?  " name
if [ "$name" = "Voltron" ]; then
  echo "Hello Voltron, defender of the universe"
elif [ "$name" = "Megatron" ]; then
  echo "All hail Megatron, leader of the Decepticons"
elif [ "$name" = "She-Ra" ]; then
  echo "Hello She-Ra: Princess of Power"
else
  echo "Hello $name"
fi
```

hello4case

```
#!/bin/bash
read -p "What is your name? " name
case "$name" in
  Voltron)
      echo "Hello $name, defender of the universe"
  ;;
  Megatron)
      echo "All hail Megatron,"
      echo "leader of the Decepticons"
  ;;
  She-[Rr]a)
      echo "Hello $name: Princess of Power"
  ;;
  *)
      echo "Hello $name"
  ;;
esac
```

Some super handy utilities (1)

`dirname path`

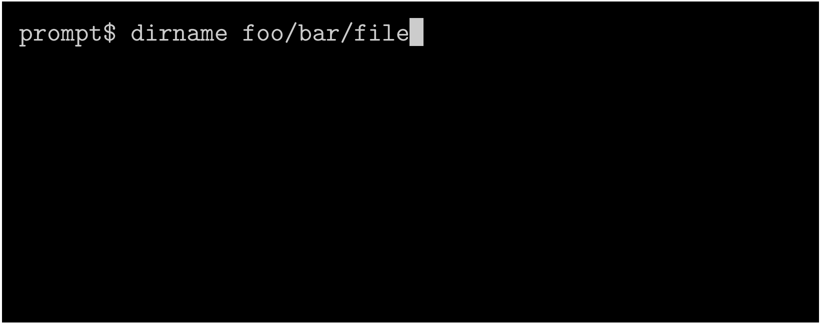
- For the given `path`, write the directory to standard output
  - Deletes everything after the last “/” character
  - Based on the string; does not check that the directory exists
- Exit status 0 on success

```
prompt$ █
```

Some super handy utilities (1)

`dirname path`

- For the given `path`, write the directory to standard output
  - Deletes everything after the last “/” character
  - Based on the string; does not check that the directory exists
- Exit status 0 on success

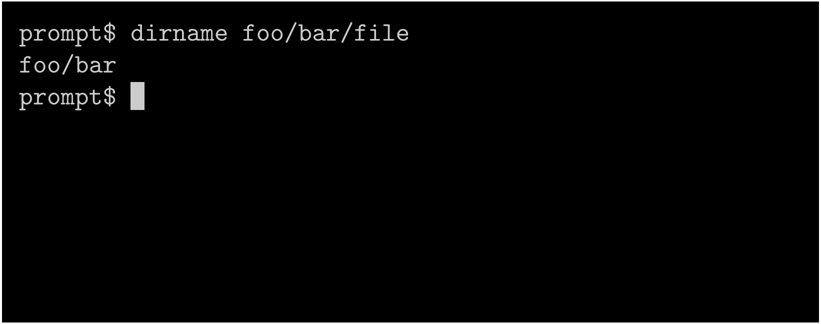


```
prompt$ dirname foo/bar/file
```

Some super handy utilities (1)

`dirname path`

- For the given `path`, write the directory to standard output
    - Deletes everything after the last "/" character
    - Based on the string; does not check that the directory exists
- Exit status 0 on success

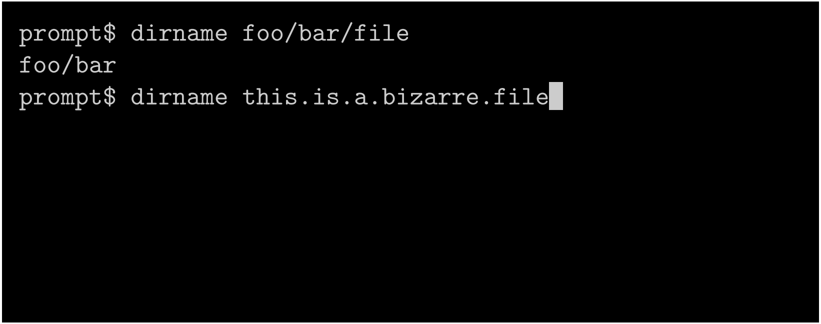


```
prompt$ dirname foo/bar/file
foo/bar
prompt$
```

Some super handy utilities (1)

`dirname path`

- For the given `path`, write the directory to standard output
  - Deletes everything after the last “/” character
  - Based on the string; does not check that the directory exists
- Exit status 0 on success

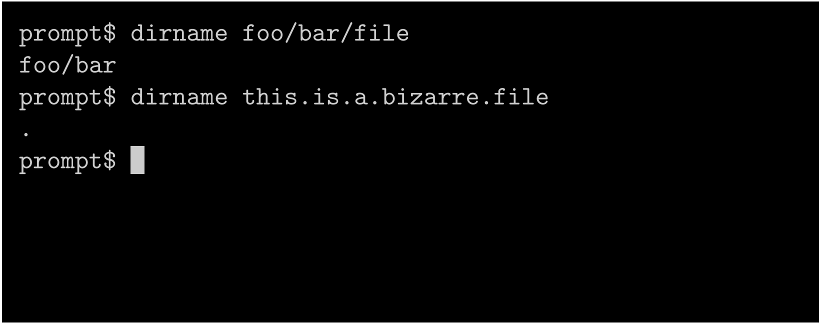


```
prompt$ dirname foo/bar/file
foo/bar
prompt$ dirname this.is.a.bizarre.file
```

Some super handy utilities (1)

`dirname path`

- For the given `path`, write the directory to standard output
  - Deletes everything after the last "/" character
  - Based on the string; does not check that the directory exists
- Exit status 0 on success

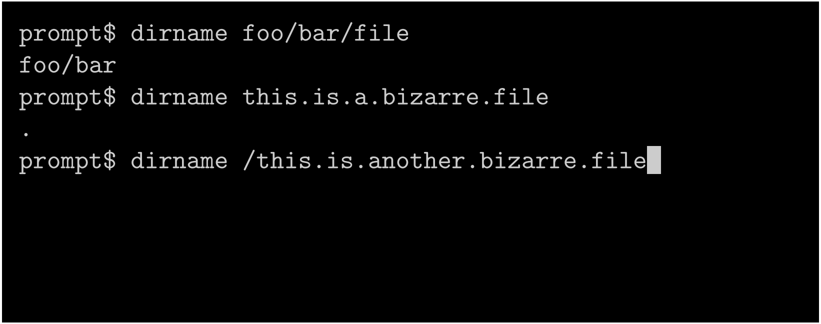


```
prompt$ dirname foo/bar/file
foo/bar
prompt$ dirname this.is.a.bizarre.file
.
prompt$ 
```

Some super handy utilities (1)

`dirname path`

- For the given `path`, write the directory to standard output
  - Deletes everything after the last "/" character
  - Based on the string; does not check that the directory exists
- Exit status 0 on success



```
prompt$ dirname foo/bar/file
foo/bar
prompt$ dirname this.is.a.bizarre.file
.
prompt$ dirname /this.is.another.bizarre.file
```

Some super handy utilities (1)

dirname path

- For the given path, write the directory to standard output
  - Deletes everything after the last "/" character
  - Based on the string; does not check that the directory exists
- Exit status 0 on success

```
prompt$ dirname foo/bar/file
foo/bar
prompt$ dirname this.is.a.bizarre.file
.
prompt$ dirname /this.is.another.bizarre.file
/
prompt$ █
```

Some super handy utilities (1)

dirname path

- For the given path, write the directory to standard output
  - Deletes everything after the last "/" character
  - Based on the string; does not check that the directory exists
- Exit status 0 on success

```
prompt$ dirname foo/bar/file
foo/bar
prompt$ dirname this.is.a.bizarre.file
.
prompt$ dirname /this.is.another.bizarre.file
/
prompt$ dirname /an/absolute/path/to/file
```

Some super handy utilities (1)

dirname path

- For the given path, write the directory to standard output
  - Deletes everything after the last "/" character
  - Based on the string; does not check that the directory exists
- Exit status 0 on success

```
prompt$ dirname foo/bar/file
foo/bar
prompt$ dirname this.is.a.bizarre.file
.
prompt$ dirname /this.is.another.bizarre.file
/
prompt$ dirname /an/absolute/path/to/file
/an/absolute/path/to
prompt$ █
```

Some super handy utilities (2)

`basename path`

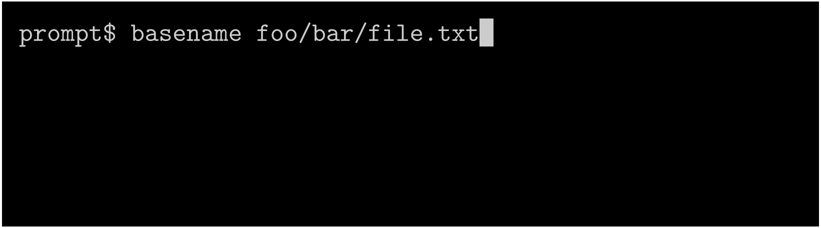
- Print `path` with outermost directories removed
- Will also remove a specified suffix, if present
- Exit status 0 on success
- Usage 1: `basename path suffix`
- Usage 2: `basename [-s suffix] path path ...`

```
prompt$ 
```

Some super handy utilities (2)

`basename path`

- Print `path` with outermost directories removed
- Will also remove a specified suffix, if present
- Exit status 0 on success
- Usage 1: `basename path suffix`
- Usage 2: `basename [-s suffix] path path ...`

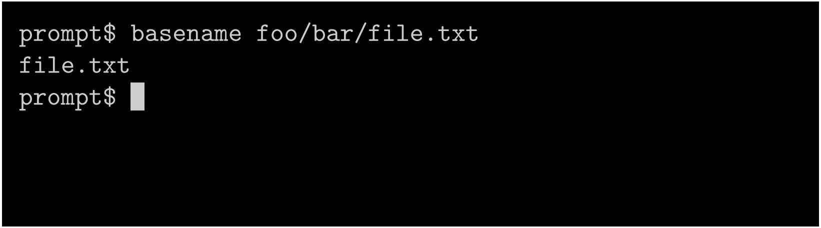


```
prompt$ basename foo/bar/file.txt
```

Some super handy utilities (2)

`basename path`

- Print `path` with outermost directories removed
- Will also remove a specified suffix, if present
- Exit status 0 on success
- Usage 1: `basename path suffix`
- Usage 2: `basename [-s suffix] path path ...`



```
prompt$ basename foo/bar/file.txt
file.txt
prompt$ █
```

Some super handy utilities (2)

basename path

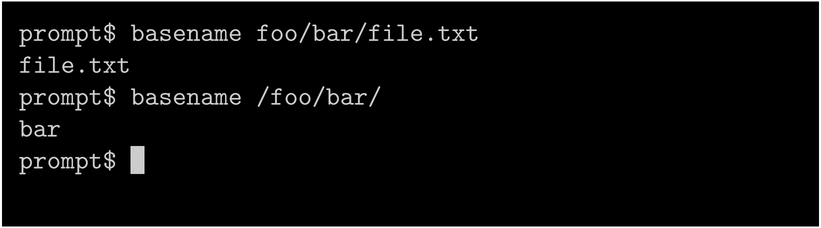
- Print path with outermost directories removed
- Will also remove a specified suffix, if present
- Exit status 0 on success
- Usage 1: `basename path suffix`
- Usage 2: `basename [-s suffix] path path ...`

```
prompt$ basename foo/bar/file.txt
file.txt
prompt$ basename /foo/bar/
```

Some super handy utilities (2)

basename path

- Print path with outermost directories removed
- Will also remove a specified suffix, if present
- Exit status 0 on success
- Usage 1: `basename path suffix`
- Usage 2: `basename [-s suffix] path path ...`

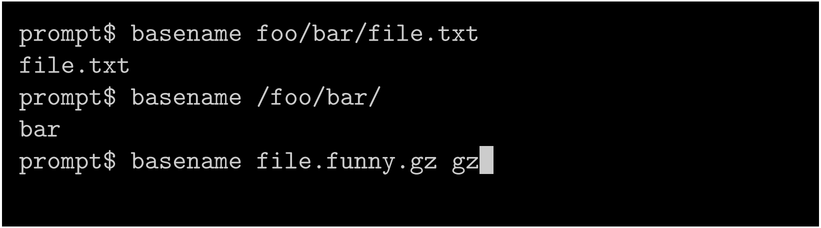


```
prompt$ basename foo/bar/file.txt
file.txt
prompt$ basename /foo/bar/
bar
prompt$ █
```

Some super handy utilities (2)

basename path

- Print path with outermost directories removed
- Will also remove a specified suffix, if present
- Exit status 0 on success
- Usage 1: `basename path suffix`
- Usage 2: `basename [-s suffix] path path ...`

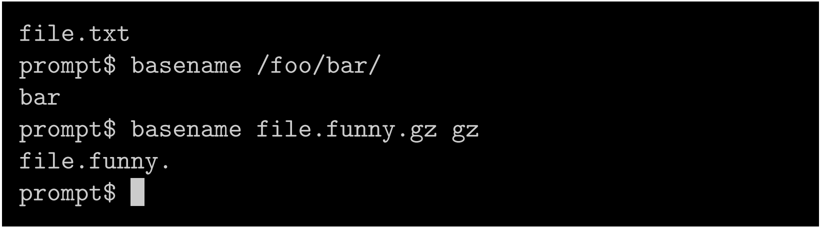


```
prompt$ basename foo/bar/file.txt
file.txt
prompt$ basename /foo/bar/
bar
prompt$ basename file.funny.gz gz
```

Some super handy utilities (2)

basename path

- Print `path` with outermost directories removed
- Will also remove a specified suffix, if present
- Exit status 0 on success
- Usage 1: `basename path suffix`
- Usage 2: `basename [-s suffix] path path ...`



```
file.txt
prompt$ basename /foo/bar/
bar
prompt$ basename file.funny.gz gz
file.funny.
prompt$
```

Some super handy utilities (2)

basename path

- Print `path` with outermost directories removed
- Will also remove a specified suffix, if present
- Exit status 0 on success
- Usage 1: `basename path suffix`
- Usage 2: `basename [-s suffix] path path ...`

```
file.txt
prompt$ basename /foo/bar/
bar
prompt$ basename file.funny.gz gz
file.funny.
prompt$ basename -s y.gz file.txt file.funny.gz
```

Some super handy utilities (2)

basename path

- Print path with outermost directories removed
- Will also remove a specified suffix, if present
- Exit status 0 on success
- Usage 1: `basename path suffix`
- Usage 2: `basename [-s suffix] path path ...`

```
prompt$ basename file.funny.gz gz
file.funny.
prompt$ basename -s y.gz file.txt file.funny.gz
file.txt
file.funn
prompt$ █
```

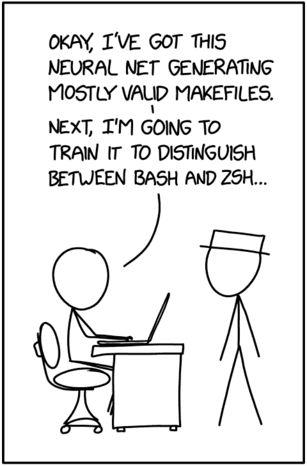
How to rename all “.cc” files as “.c”?

How to rename all ".cc" files as ".c"?

```
for f in *.cc; do
  bn=$(basename -s .cc $f)
  mv $bn.cc $bn.c
done
```

basename	:	print base file or directory name of a pathname
dirname	:	print directory containing a pathname
getopts	:	Check for switches
shift	:	Shift arguments
test	:	Set exit status based on an expression

An appropriate xkcd comic: http://xkcd.com/2510


OKAY, I'VE GOT THIS NEURAL NET GENERATING MOSTLY VALID MAKEFILES.
NEXT, I'M GOING TO TRAIN IT TO DISTINGUISH BETWEEN BASH AND ZSH...

PEOPLE OFTEN USE ANCIENT TOOLS AND UIs TO DEVELOP MODERN CUTTING-EDGE TECHNOLOGY, BUT I DO IT THE OTHER WAY AROUND.

End of lecture