CS 241: Systems Programming
Lecture 7. Shell Scripting 2

Fall 2025

Prof. Stephen Checkoway

Script positional parameters

```
$ ./script arg1 ... argn # or bash script arg1 ... argn
```

Special variables

- `$#` — Number of arguments
- `$0` — Name used to call the shell script (./script or script)
- `$1`, `$2`, …, `$9` — First nine arguments
- `${`*n*`}` — *n*th argument (braces needed for *n* > 9)
- `"$@"` — all arguments; expands to each argument, individually quoted
- `"$*"` — all arguments; expands to a single quoted string

Script positional parameters

```
$ ./script arg1 arg2 arg3
```

Special variables

- `"$@"` — all arguments; expands to each argument, individually quoted
  - ‘arg1’ ‘arg2’ ‘arg3’
- `"$*"` — all arguments; expands to a single quoted string
  - ‘arg1 arg2 arg3’

Two special builtin commands

`set --`

- Can set positional parameters (and `$#`)

  `set -- arg1 arg2 … argn`

`shift`
`shift n`

- Discard first *n* parameters and rename the remaining starting at `$1`
- If `n` is omitted, it's the same as `shift 1`
- Updates `$#`

Iterate over arguments

```
while [[ $# -gt 0 ]]; do
  arg="$1"
  # whatever you want to do with ${arg}
  shift
done
```

Another approach

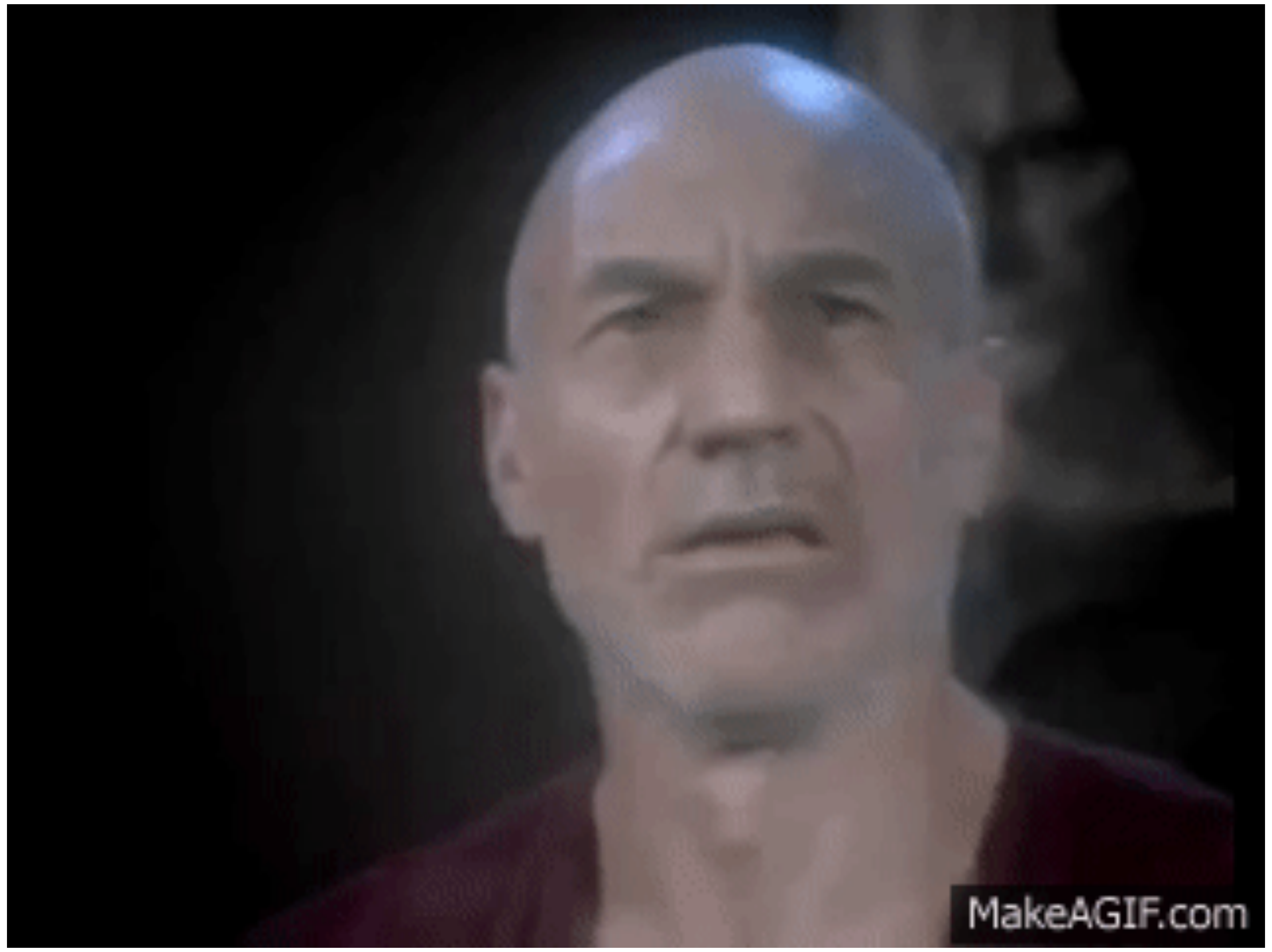
```
#!/bin/bash

echo "There are $# arguments: $*"
n=1
for arg in "$@"; do
    echo "$n: [$arg]"
    (( n ++ ))
done
```

```
$ ./printargs.sh AAA BBB 'CCC DDD' "EEE FFF"\ GGG\ 'HHH III’
```

How many arguments does this print out?

THERE ARE… FOUR ARGS

```
$ ./printargs.sh AAA BBB 'CCC DDD' "EEE FFF"\ GGG\ 'HHH III’
There are 4 arguments: AAA BBB CCC DDD EEE FFF GGG HHH III
1: [AAA]
2: [BBB]
3: [CCC DDD]
4: [EEE FFF GGG HHH III]
```



Functions

```
#!/bin/bash

num_args() {
  echo "foo called with $# arguments"
  if [[ $# -gt 0 ]]; then
    echo "  foo's first argument: $1"
  fi
}

echo "Script $0 invoked with $# arguments"
if [[ $# -gt 0 ]]; then
  echo "  $0's first argument: $1"
fi

num_args 'extra' "$@" 'args'
```

`local` creates a local variable.

What does this script print out?

A. `A`

B. `B`

C. `C`

D. The empty string

E. Nothing, it's a syntax error

```
#!/bin/bash

foo() {
  x="$1"
}
bar() {
  local x="$1"
}

x=A
foo B
bar C
echo "${x}"
```

`local` creates a local variable.

What does this script print out?

A. `A`

B. `B`

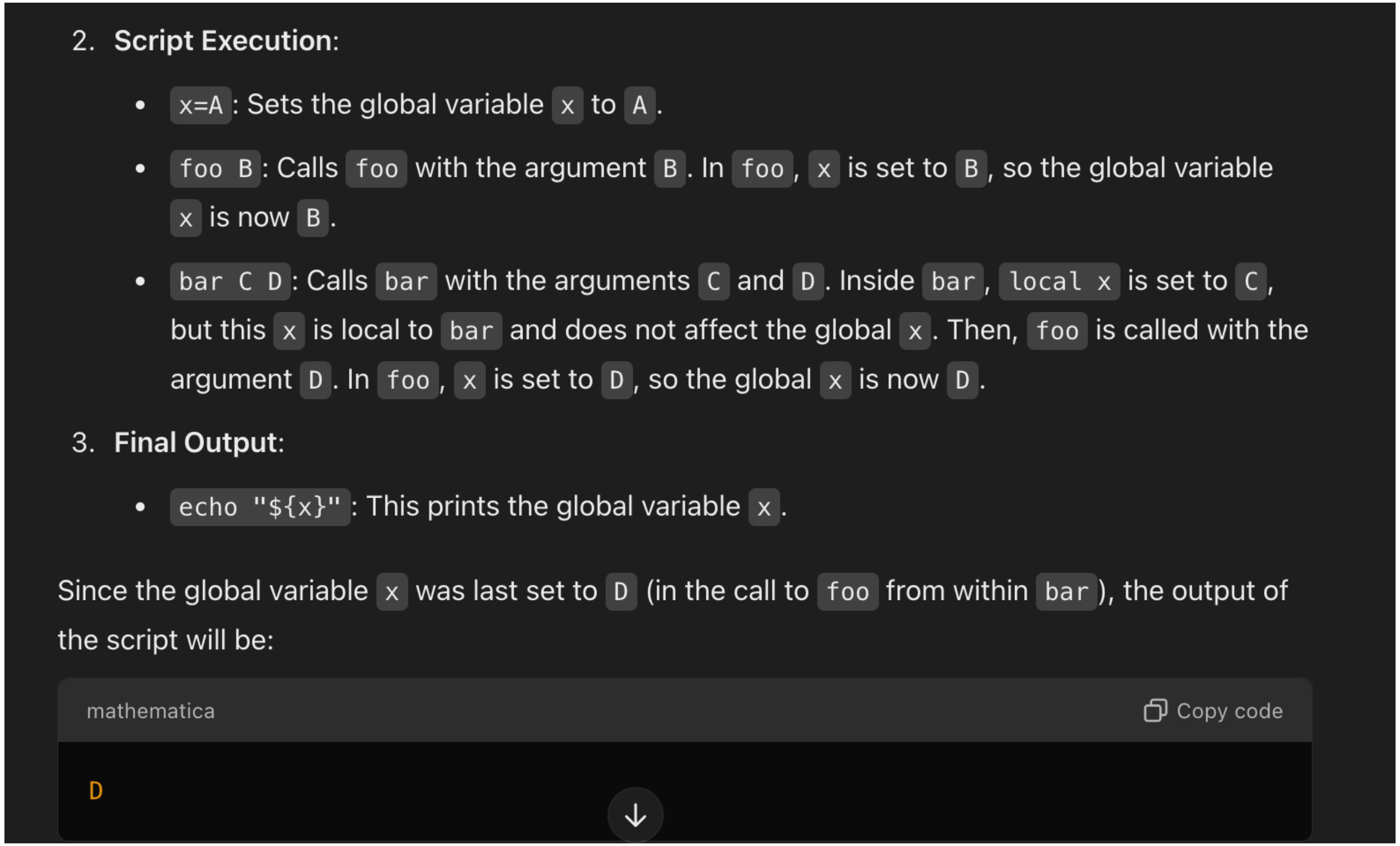
C. `C`

D. `D`

E. Nothing, it's a syntax error

```
#!/bin/bash

foo() {
  x="$1"
}
bar() {
  local x="$1"
  foo "$2"
}

x=A
foo B
bar C D
echo "${x}"
```

ChatGPT is very convincing, but wrong!



2. **Script Execution**:
   - `x=A`: Sets the global variable `x` to `A`.
   - `foo B`: Calls `foo` with the argument `B`. In `foo`, `x` is set to `B`, so the global variable `x` is now `B`.
   - `bar C D`: Calls `bar` with the arguments `C` and `D`. Inside `bar`, `local x` is set to `C`, but this `x` is local to `bar` and does not affect the global `x`. Then, `foo` is called with the argument `D`. In `foo`, `x` is set to `D`, so the global `x` is now `D`.
3. **Final Output**:
   - `echo "${x}"`: This prints the global variable `x`.

Since the global variable `x` was last set to `D` (in the call to `foo` from within `bar`), the output of the script will be:

```
D
```

Some Variable Expansion

```
$ "${parameter##word}"
```

Bash can expand a variable but only return the parts that match some `word`

The ‘##’ means to return whatever part of `parameter` matches `word`, but delete the longest matching case

-
  ```
  parameter = “This is a sentence. Hooray!”
  echo “${parameter##*.}”
  ```
  - This outputs everything after the ‘.’ - “ Hooray!”
  - The longest match of *. is “This is a sentence.”

Lists — sequence of commands

Pipeline: `cmd1 | cmd2 | … | cmdn`

- Exit value is exit value of last command in the pipeline
- Exit value can be negated by `! cmd1 | … | cmdn`

Lists

- `pipeline1 ; pipeline2 ; … ; pipelinen`
  can replace `;` with newline
- `pipeline1 && pipeline2`
  `pipeline2` runs if and only if `pipeline1` returns 0
- `pipeline1 || pipeline2`
  `pipeline2` runs if and only if `pipeline1` doesn't return 0
- `pipeline &`
  runs `pipeline` in the background

When writing a script, we often want to change directories with cd. If the directory doesn't exist, the script should exit with an error.

Which construct should we use?

```
A. cd "${dir}" && exit 0

B. cd "${dir}" || exit 0

C. cd "${dir}" && exit 1

D. cd "${dir}" || exit 1

E. cd "${dir}" && exit 2
```

Arrays

Assign values at numeric indices

- `arr[0]=foo`
- `arr[1]=bar`

Assign multiple values at once

- `arr=(foo bar)`
- `txt_files=(*.txt) # pathname expansion/globbing`

Append (multiple values) to an array

- `arr+=(qux asdf)`

Arrays

Access an element; **braces are required!**

- `${arr[0]}`
- `${arr[1]}`

-
  ```
  n=42
  ${arr[n]}
  ```

Access all elements

- `"${arr[@]}"` # expands to each element quoted by itself
- `"${arr[*]}"` # expands to one quoted word containing all elements

Array length

- `${#arr[@]}`

If `arr` is the two element array

`arr=('foo bar' baz)`

how should we print each element of `arr`?

```
A. for elem in ${arr}; do
     echo "${elem}"
   done

B. for elem in "${arr}"; do
     echo "${elem}"
   done

C. for elem in "${arr[*]}"; do
     echo "${elem}"
   done

D. for elem in "${arr[@]}"; do
     echo "${elem}"
   done

E. for (( n=0 ; n < ${#arr[@]}; n+=1 )); do
     echo "$arr[n]"
   done
```