Table of contents

1. Bash Arrays
2. Git Branching
Some Logistics

- Homework 2...
- Last time: "...I wanted to get your HW to you. That will happen tonight."
  - ...will send the fake release out via Piazza.
  - DO NOT UNDER ANY CIRCUMSTANCES ADD ANYTHING IN AN a2 FOLDER IN YOUR REPO!!!!!!
Bash Arrays
Bash Arrays

- Arrays in **bash** are extraordinarily flexible in some senses...
- ...and particularly finkicky in other senses.
- The short version:

  arr=( use parentheses and separate by spaces )

- Mixed types: **my_arr**=( "a string" 1 twelve "33" )
- Question: what are the types of **twelve** and "33"?
  - **twelve** would be interpreted as a **string**.
  - "33" can be either a **string** or a number!
- Types are not exactly a thing in **bash**.
- **echo $(( ${my_arr[3]} + 99 ))**
  - Woah that syntax is crazy.
  - Remember that (( double parens )) are arithmetic expressions.
  - The $ in front of them evaluated the expression.
  - The last part is indexing the array, which we'll get to.
• The majority of the remaining examples are either copied or modified from [2].
  • This is an excellent resource, and you should explore it on your own.
  • I do not have time to cover all of the cool and obscure things you can do with arrays.

• You should follow along either in a bash script, or in your shell.
Alternative Initialization

• Using (parentheses enumerations), and other initializations, give you indices between 0 up to but not including the length of the array.

• You can create your own indices instead!

```plaintext
arr[22]=22
arr[33]=33
arr[51]="a string value"
arr[52]="different string value"
```

• Of course, you can add on the indices to a (parenthetical declaration) after the fact if you want.

• You cannot have an array of arrays.
Array Functions

- You perform an **array** operation with `{$expr}`.
- You use the name of the variable followed by the operation:

```bash
echo "Index 11: ${arr[11]}" # prints: Index 11: 11
echo "Index 51: ${arr[51]}" # prints: Index 51: a string value
echo "Index 0: ${arr[0]}" # DOES NOT EXIST! (aka nothing)
```

- Recall that the `@` and `*` expand differently:

```bash
echo "Individual: ${arr[@]}"
# Individual: 11 22 33 a string value different string value
echo "Joined::::: ${arr[*]}"
# Joined: 11 22 33 a string value different string value
```

- Differently how?

```bash
echo "Length of Individual: ${#arr[@]}"
# Length of Individual: 5
echo "Length of Joined::::: ${#arr[*]}"
# Length of Joined::::: 5
```
Different HOW?!!!

- Easier to compare with loops, these will be in-line so you can copy-paste.
  - Remember that `;` allows you to continue on the same line.
- Individual expansion (@):

```bash
for x in "${arr[@]}"; do echo "$x"; done
# 11
# 22
# 33
# a string value
# different string value
```

- Joined expansion (*):

```bash
for x in "${arr[*]}"; do echo "$x"; done
# 11 22 33 a string value different string value
```

- The * loop only executes once.
- General rule: if you want them all, use @ to expand.
• Evaluate expressions and initialize at once:

```bash
echo "Index 44: \${arr[44]}"        # Index 44: 44
echo "Index 55: \${arr[55]}"        # Index 55: 55
```

• Alternative index specifications:

```bash
new_arr=([17]="seventeen" [24]="twenty-four")
new_arr[99]="ninety nine"      # may as well, not new
for x in "$\{new_arr[@]\}"; do echo "$x"; done
# seventeen
# twenty-four
# ninety nine
```

• Get the list of indices:

```bash
for idx in "$\{!new_arr[@]\}"; do echo "$idx"; done
# 17
# 24
# 99
```
You can just as easily splice your arrays.

Use @ to get the whole array, then specify the indices you wish to splice.

- ${var[@]:start:end}
- Don't need to specify end (will take until last index).

```bash
zed=( zero one two three four )
echo "From start: ${zed[@]:0}"
# From start: zero one two three four
echo "From 2: ${zed[@]:2}"
# From 2: two three four
echo "Indices [1-3]: ${zed[@]:1:3}"
# Indices [1-3]: one two three
for x in "${zed[@]:1:3}"; do echo "$x"; done
# one
# two
# three
for x in "${zed[*]:1:3}"; do echo "$x"; done
# one two three
```
• This is the core functionality of arrays that I believe you will profit from.
• This is actually not even close to what you can do with arrays in bash.
• I highly suggest you go through the examples listed in [2].
  • Search for Substring Removal for some insanely cool tricks!
Git Branching
The Lecture Slides Repository!
References
