Leiden University
Medical Center

Scripting for Life Science Researchers

Advanced scripting I

Jeroen Laros

Outline

Motivation

Suppose we want to keep record of addresses.

```
1  person1_name='John'
2  person1_town='Podunk'
3  person1_age=29
4
5  person2_name='Jane'
6  person2_town='Podunk'
7  person2_age=32
```

Listing 1: Keeping records of addresses.

Motivation

Suppose we want to keep record of addresses.

```
1  person1_name='John'
2  person1_town='Podunk'
3  person1_age=29
4
5  person2_name='Jane'
6  person2_town='Podunk'
7  person2_age=32
```

Listing 1: Keeping records of addresses.

This is not practical if we have many addresses to keep.
Impossible if we do not know how many people there are in advance.

Definition and terminology

An *array* is a systematic arrangement of objects.

Also known as:

- Record.
- List.
- Vector.

Arrays are *indexed*, we can ask for te *n*-th element.

Definition and terminology

0	1	2	3	4	5	6
John	Jane	Johnny	Jane			

Figure 1: An array containing names.

Definition and terminology

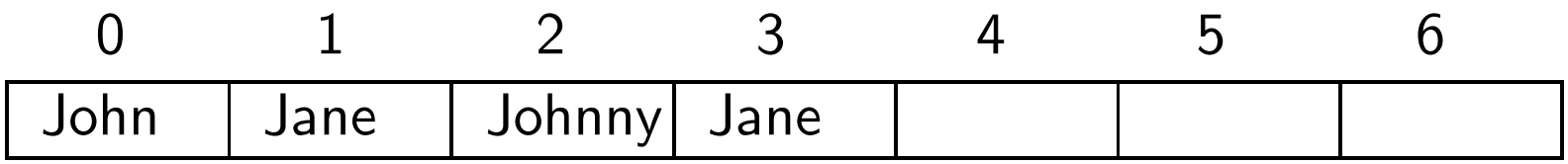

Figure 1: An array containing names.

Term	Description	Example
Element	Smallest unit of the array	1: Jane
Index	Position of the element	1
Value	Value of the element	Jane
Type	Type of the content	string

Table 1: Terminology.

Definition and terminology

Bash only knows two types:

- Integer (numbers).
- String.

Arrays may contain elements of different types.

Indexing

Usually, the first element in an array has index 0:

- The second element has index 1.
- The third element has index 2.
- ...

All indices must be non-negative:

- We can start with element 12.
- Element 2 and 4 may exist without element 3.

Initialisation

We first make clear we want to define an array with `declare -a`.

```
1   declare -a name
2
3   name=('Jonn' 'Jane' 'Johnny')
```

Listing 2: Initialise and fill an array.

The created array will start at 0 and the subsequent elements have consecutive indices.

Initialisation

We first make clear we want to define an array with `declare -a`.

```
1   declare -a name
2
3   name=('Jonn' 'Jane' 'Johnny')
```

Listing 2: Initialise and fill an array.

The created array will start at 0 and the subsequent elements have consecutive indices.

```
1   name[3]='Jane'
```

Listing 3: Add an element.

Retrieve and update elements

Retrieval is like any normal variable, i.e., use `${}`.

```
1   $ echo ${name[2]}
2   Johnny
```

Listing 4: Retrieve elements.

Updating is done in the same way as adding an element.

```
1   $ name[0]='John'
2   $ echo ${name[0]}
3   John
```

Listing 5: Update an element.

Special operations

```
1   $ echo ${name[*]}
2   John Jane Johnny Jane
```

Listing 6: Retrieve all values.

```
1   $ echo ${!name[*]}
2   0 1 2 3
```

Listing 7: Retrieve all indices.

```
1   $ echo ${#name[*]}
2   4
```

Listing 8: Number of elements.

Exercise

Initialisation and manipulation:

- Create an array named `counts` that contains the words 'one', 'two', 'drie' and 'four'.
- Add the word 'five' at the end of the array.
- Note that 'three' is not spelled correctly. Can you update only this element?

Adding an item:

- Add the word 'last' to the end of the array. This needs to work multiple times.

Hints:

- Use `${counts[*]}` after every operation.

Definition and terminology

An *associative array* is an *abstract data type* composed of a collection of `(key, value)` pairs, such that each possible key appears at most once in the collection.

Also known as:

- Dictionary.
- Hash.
- Map.
- Symbol table.

Definition and terminology

An *associative array* is an *abstract data type* composed of a collection of `(key, value)` pairs, such that each possible key appears at most once in the collection.

Also known as:

- Dictionary.
- Hash.
- Map.
- Symbol table.

Think of a real dictionary.

- Key: query word.
- Value: meaning of the query word.

Definition and terminology

`key`	`value`
'John'	29
'Jane'	32
'Johnny'	2

Table 2: Ages of family members.

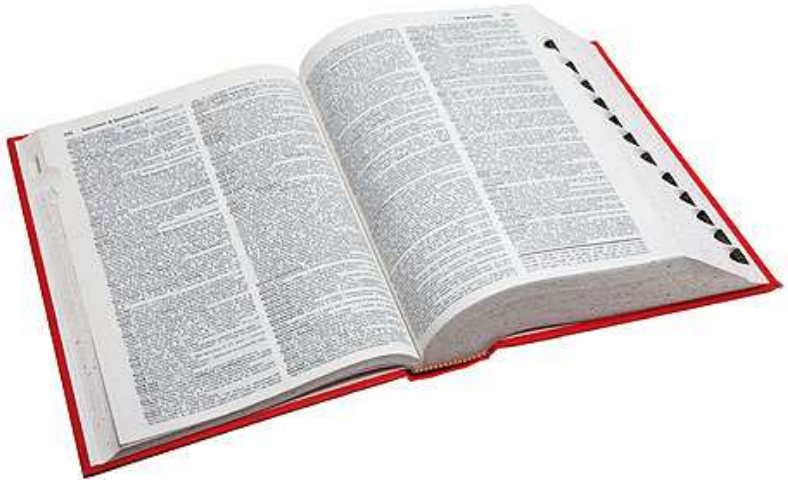

Figure 2: `(key, value)` pairs in a book.

Initialisation

```
1   declare -A age
2
3   age=(['John']=29 ['Jane']=31)
```

Listing 9: Initialise and fill an associative array.

```
1   age['Johnny']=2
```

Listing 10: Add an element.

Retrieve and update elements

```
1   $ echo ${age['John']}
2   29
```

Listing 11: Retrieve an element.

```
1   $ age['Jane']=32
2   $ echo ${age['Jane']}
3   32
```

Listing 12: Update an element.

Special operations

```
1   $ echo ${age[*]}
2   Jane John Johnny
3   $ echo ${!age[*]}
4   32 29 2
```

Listing 13: Retrieve all values and keys.

Note that the order is not preserved, but the order of the keys is the same as the order of the values.

```
1   $ echo ${#age[*]}
2   3
```

Listing 14: Number of elements.

Boolean operators

A *Boolean data type* is a data type with only two possible values: true or false.

x	y	x AND y	x OR y	NOT x
false	false	false	false	true
true	false	false	true	false
false	true	false	true	
true	true	true	true	

Table 3: Truth table.

Boolean operators

A *Boolean data type* is a data type with only two possible values: true or false.

x	y	x AND y	x OR y	NOT x
false	false	false	false	true
true	false	false	true	false
false	true	false	true	
true	true	true	true	

Table 3: Truth table.

Exercise:

$$x = \text{true}, y = \text{false}$$

$$\text{NOT } ((x \text{ AND } y) \text{ OR } (\text{NOT } y))$$

Status codes

A program may return a *status code*.

- Zero if everything went okay (true).
- Non-zero on failure (false).

Status code can be retrieved via the special variable `${?}`

```
1   $ grep -q 'root' '/etc/passwd'
2   $ echo ${?}
3   0
4
5   $ grep -q 'toor' '/etc/passwd'
6   $ echo ${?}
7   1
```

Listing 15: Testing for presence in a file.

Tests

Expressions are evaluated by placing them between brackets.

```
1   $ [ 'abc' = 'abc' ]
2   $ echo ${?}
3   0
4
5   $ [ 9 -gt 10 ]
6   $ echo ${?}
7   1
```

Listing 16: Comparison of strings and integers.

```
1   $ [ 'abc' = 'xyz' -o 9 -lt 10 ]
2   $ [ 'abc' = 'xyz' -a 9 -lt 10 ]
3   $ [ ! 'abc' = 'xyz' ]
```

Listing 17: Logical operators.

Arithmetic using variables.

Arithmetic expressions are evaluated as follows:

```
$(( expression ))
```

```
1  $ echo $((2 * 5))
2  10
3  $ echo $((4 % 2))
4  0
```

Listing 18: Basic integer arithmetic.

Arithmetic using variables.

Arithmetic expressions can be combined with tests.

```
1   $ value=7
2   $ [ $((${value} % 3)) -eq 0 ]
3   $ echo ${?}
4   1
```

Listing 19: Test whether a value is divisible by 3.

Exercise

First initialise a variable named `value` to 10.

Do the following tests:

- Is the value larger or equal than 10?
- Is the value divisible by 2?

Testing file properties:

- Does the file `/etc/passwd` exist?
- Is the file `/etc/passwd` newer than `/bin/bash`?

Hints:

- Use the manual: `man test`.

Structure of a conditional statement

Most simple form:

```
if expression; then
  body
fi
```

The body is executed if the expression evaluates to true.

Structure of a conditional statement

More general form:

```
if expression; then
  body 1
else
  body 2
fi
```

The first body is executed if the expression evaluates to true, otherwise, the second body is executed.

Structure of a conditional statement

Extended form:

```
if expression; then
  body 1
elif expression; then
  body 2
else
  body 3
fi
```

Structure of a conditional statement

```
1   if [ ${value} -eq 10 ]; then
2     echo 'equal to 10'
3   elif [ ${value} -lt 10 ]; then
4     echo 'larger than 10'
5   else
6     echo 'smaller than 10'
7   fi
```

Listing 20: Example.

FizzBuzz

FizzBuzz is a group word game for children to teach them about division.

- Start counting from 1.
- If the current number is divisible by 3, say 'Fizz'.
- If the current number is divisible by 5, say 'Buzz'.
- If the current number is divisible by 3 and 5, say 'FizzBuzz'.
- Simply say the number if none of the rules above apply.

Hints:

- You have seen the *modulo* operator before.
- See the slides on Booleans.

`https://en.wikipedia.org/wiki/Fizz_buzz`

Leiden University
Medical Center

Acknowledgements

Jeroen Laros
Magnus Palmblad
Jonathan Vis
Mihai Lefter
Yassene Mohammed