Week 4 - Topic 5: Arrays and Filehandling

Week 4: Topic 5: Arrays and Filehandling

Overview:

- Design note
- Arrays
- Multi-Dimensional Arrays
- File Handling
- The HD Maze Task

Design Note I

Remember we wish to reduce the amount of code we need to write and test.

Therefore we want to write code to be as reusable as possible.

Design Note II

Consider the code below

This code does work, but it is very specific – in name and function. Can we make the code more re-usable?

Run RUBY

```ruby
YEAR_TRUMP_ELECTED = 2016

def calculate_age_when_trump_elected(year_born)
    return (YEAR_TRUMP_ELECTED - year_born)
end

def main
  print("Age is: ")
  puts(calculate_age_when_trump_elected(2012).to_s())
end

main
```

Design Note III

How about a function that takes two numbers and returns the difference between them?

The function below allows the arguments to passed in any order and will return the difference (as a positive integer)

Run RUBY

```ruby
def difference(a, b)
  if (a > b)
    return (a - b)
  else
    return (b - a)
  end
end

def main()
  puts(difference(7, 9).to_s())
end

main()
```

Design Note IV

Run RUBY

```ruby
YEAR_TRUMP_ELECTED = 2016

def calculate_age_when_trump_elected(year_born)
    return (YEAR_TRUMP_ELECTED - year_born)
end

def difference(a, b)
  if (a > b)
    return (a - b)
  else
    return (b - a)
  end
end

```

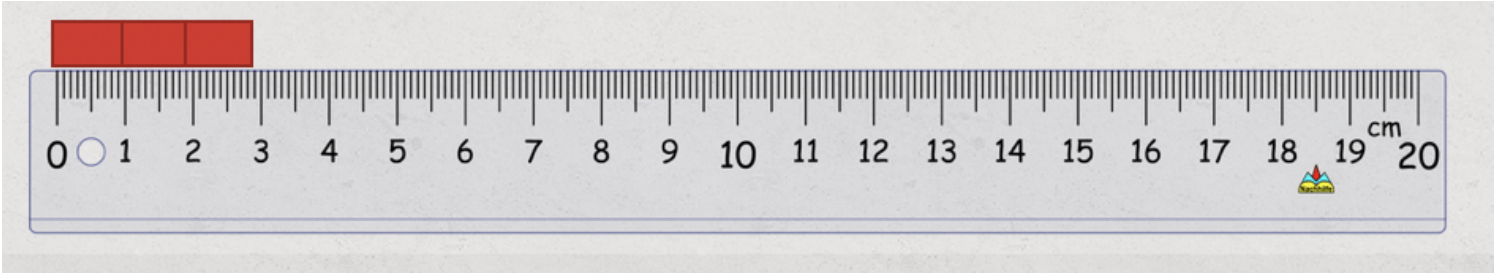
Arrays – The problem

- We have seen we can create variables to hold values.
- We might want to read in a list of names.
- But each variable holds just one value.
- What if we want to hold many names – 10, 100 or 1000? Do we create 1000 variables? How does the program move from one variable to the next to fill them up?

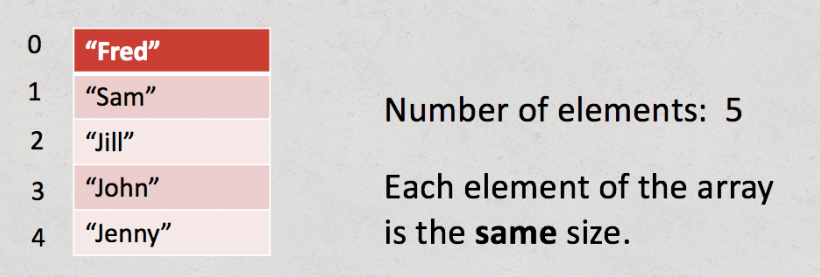
Arrays – The Solution I

- It holds multiple items (like a list)
- Eg: [‘Fred’, ‘Sam’, ‘Jenny’, ‘Jill’]

Consider our ‘bricks” example:



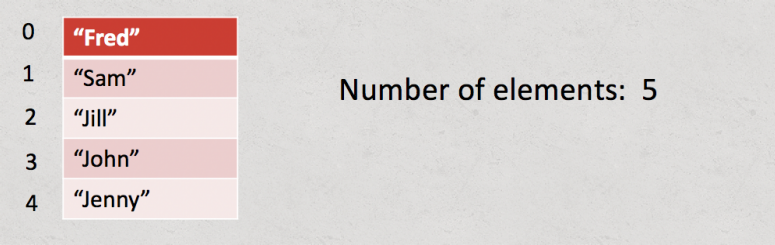
Arrays – The Solution II



Question: How long is a String?

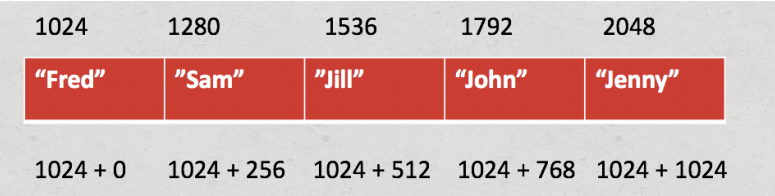
i.e How much memory is needed for each element?

Arrays – The Solution III



Let us assume (for now) that each element has 256 bytes available for each string.

Representation in Memory:



Creating Arrays I

This is one way of creating Arrays (i.e hard coding them):

```
genre_names = ['Pop', 'Classic', 'Jazz', 'Rock']
```

This creates an array of 4 elements.

Accessing Arrays I

`genre_names = ['Pop', 'Classic', 'Jazz', 'Rock']`

To access all the elements we could do the following.

This prints out each element of the array, starting with the first (zero) element:

Run RUBY

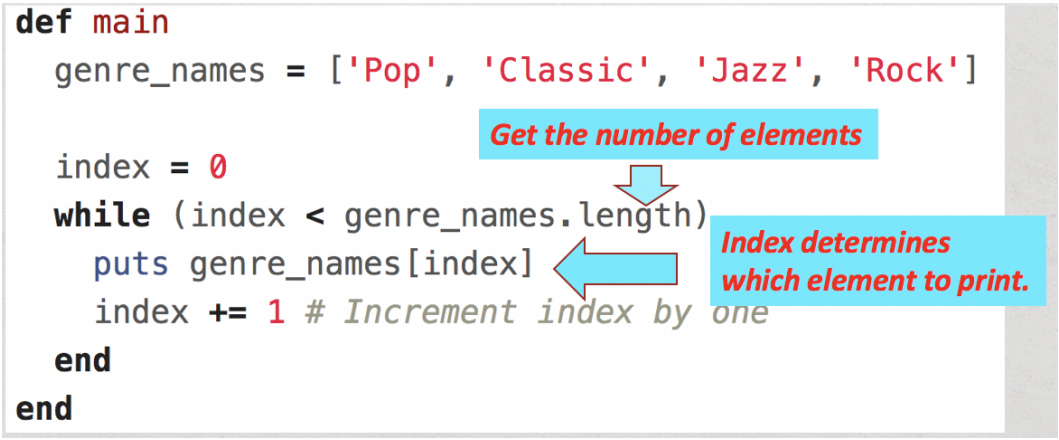
```ruby
genre_names = ['Pop', 'Classic', 'Jazz', 'Rock']

puts genre_names[0]
puts genre_names[1]
puts genre_names[2]
puts genre_names[3]

# what happens if we uncomment the following:
#puts "fourth: " + genre_names[4].to_s

```

Accessing Arrays II

A better way to access the elements might be with a loop:

Run RUBY

```ruby

def main()
  genre_names = ['Pop', 'Classic', 'Jazz', 'Rock']

  index = 0
  while (index < genre_names.length)
    puts(genre_names[index])
    index += 1 # Increment index by one
  end
end

main()
```



Output an array

Create an array (with hard-coded values) that contains: the following: ["apple", "pear", "orange", "plum"]

Use fruits.length to work out the length of the array.

Write a **while** loop to print out the position of the item in the array and the contents of the array as follows:

```
[user@sahara ~]$ ruby array_output.rb
0 apple
1 pear
2 orange
3 plum
[user@sahara ~]$
```

Putting values into arrays

- We created an Array with hard-coded values.
- But what if we don't know the values when we wrote the code? (e.g we want to read them in)
- We need a way to put a value in an Array.
- We can do that as follows:

If we are reading the values in, we do not need initial values in the Array.
We can create an "empty" array for 4 elements as follows:

Run RUBY

```ruby
my_array = Array.new(4)
puts "Enter a value for the array: "
my_array[0] = gets
puts "Value in array [0] is #{my_array[0]}"
```

Reading multiple values into an array

Now we can read multiple values, note we create a new empty array below, and in Ruby it grows dynamically if we add elements to it.

Run RUBY

```ruby
my_array = Array.new(4)

index = 0
while (index < my_array.length)
    puts "Enter a value for the array: "
    my_array[index] = gets
    index += 1 # Increment index by one
end

index = 0
puts "The array contains: "
while (index < my_array.length)
    puts my_array[index]
    index += 1 # Increment index by one
```

Dynamic Arrays

In Ruby if you do not know how many elements you need, the array will grow dynamically to fit more elements:

Run RUBY

```ruby
my_array = Array.new()

index = 0
while (index < 4)
    puts "Enter a value for the array: "
    my_array[index] = gets
    index += 1 # Increment index by one
end

index = 0
puts "The array contains: "
while (index < my_array.length)
    puts my_array[index]
    index += 1 # Increment index by one
```

Below is an alternative way of creating empty arrays and of adding items:

Run RUBY

```ruby
my_array = [] # Note the alternative to: Array.new()

index = 0
while (index < 4)   # cannot use my_array.length
    puts "Enter a value for the array: "
    # Note the change from my_array[index] = gets
    my_array << gets
    index += 1 # Increment index by one
end

index = 0
puts "The array contains: "
while (index < my_array.length)
    puts my_array[index]
```

Dangers of arrays

In any language, C, Ruby, Java etc there is a danger that you might try and access an element that has not been assigned a value. In C, this might crash your program, or - even more dangerously - allow you access 'junk' data, that might cause unpredictable, and unintended behaviour.

In Ruby, you will likely get a nil value, which you also need to guard against.

Note these are logical errors.

Here is C code accessing an array out-of-bounds:

Run C

```c
#include <stdio.h>

int main(){
  int array[] = {2, 3, 5, 7};
  int index = 0;

  while (index < 3) {
    printf("Array value at %d, is: %d\n", index, array[index]);
    index++;
  }

  printf("Array value at position 10: %d", array[9]);
}
```

Here is equivalent Ruby code:

Run RUBY

```ruby

my_array = [2, 3, 5, 7]

index = 0
while (index < 4)
  puts "Array at position #{index} is #{my_array[index]}"
  index += 1
end

puts "Array at position 10 is #{my_array[9]}"


```

Fill an array

Write Ruby code to read values into an array. The code should use a while loop to read each item and to increase the size of the array as the items are added.

The values to be read are (you will need to type each of these in the terminal on a separate line):

2, 4, 6, 8, 10

Use `gets.chomp.to_i` to read the data and `puts` to print out the array as follows: `puts(int_array)` .

When you run the program the input and output should look as follows:

```
Enter a number:
2
Enter a number:
4
Enter a number:
6
Enter a number:
8
Enter a number:
10
2
4
6
8
10
```

Multi-Dimensional Arrays I

[0] [0]	**[0] [1]**	**[0][2]**	**[0][3]**	**[0][4]**	**[0][5]**	**[0][6]**
[1][0]	[1] [1]	[1] [2]	[1] [3]	[1] [4]	[1] [5]	[1] [6]
[2] [0]	[2] [1]	[2] [2]	[2] [3]	[2] [4]	[2] [5]	[2] [6]
[3] [0]	[3] [1]	[3] [2]	[3] [3]	[3] [4]	[3] [5]	[3] [6]
[4] [0]	[4] [1]	[4] [2]	[4] [3]	[4] [4]	[4] [5]	[4] [6]
[5] [0]	[5] [1]	[5] [2]	[5] [3]	[5] [4]	[5] [5]	[5] [6]

Multi-Dimensional Arrays II

Run RUBY

```ruby

puts("How many columns: ")
col_count = gets.to_i
puts("How many rows: ")
row_count = gets.to_i

table_array = Array.new(row_count)

row_index = 0
while (row_index < row_count)
    table_array[row_index] = Array.new(row_count)
    col_index = 0
    while (col_index < col_count)
    puts("Enter row #{row_index} column #{col_index} item: ")
```

Questions on Arrays

- [ ] Arrays store multiple elements
- [ ] The size of an array can be obtained with `.length`
- [ ] Arrays elements need to be accessed using an index
- [ ] Arrays start the indexing at one
- [ ] Arrays can contain arrays
- [ ] A single array can store elements of different types

File Handling

Typically we want to either:

- read from an existing file; or
- write to a new one (or rewrite)

(NB: similar methods are used for reading and writing from a network)

Opening Files

Typically we want to either:

- read from an existing file; or
- write to a new one (or rewrite)

Either way we need to first open the file:

For reading:

```
a_file = File.new("mydata.txt", "r") # open for reading
```

For writing:

```
a_file = File.new("mydata.txt", "w") # open for writing
```

If this code opens the file successfully, then `aFile` will not be null.

File Reading/Writing

To write to a file we use a version of `puts`:

```
a_file.puts("This line is being written to a file)
```

To read from a file we use a version of `gets`:

```
line_read = a_file.gets
```

Finishing Reading

When reading we can tell if we reached the end of the file using the eof method. Eg:

```
if (a_file.eof?)
    puts "No more lines to read"
end
```

Once we have finished reading or writing to a file we need to close the file.

```
a_file.close()
```

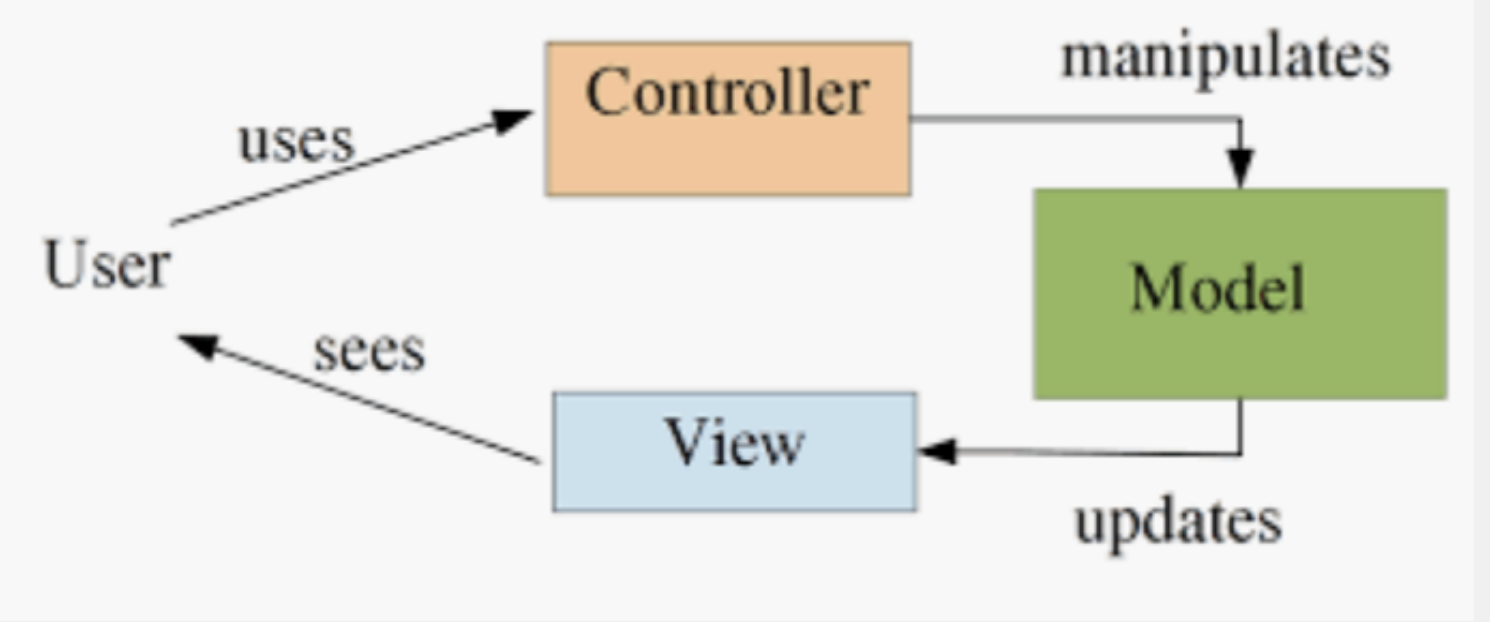
This not only frees the resource so that other programs can access it, but also flushes any output to the file that has been buffered.
Compare with USB ejection.

Writing and Reading from a file

This code slide does not have a description.

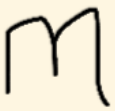
Graphical Programming - Model-View-Controller

The Gosu library allows us to write graphical user interface programs using the Model-View-Controller (MVC) pattern.



This maps to Gosu as follows:

```ruby
require 'gosu'

class GameWindow < Gosu::Window # gives your program the components for MVC

  def initialize
    # initial set up
  end

  def button_down(id)
    # allow user to interact with controller
  end

  #
  def update
    # update model
  end


  def draw
    # draw the view
  end
end

window = GameWindow.new
window.show # start the controller
```



The HD Maze Task

Create a cell:

```
cell = @columns[column_index][row_index]
```

Set up the cells on the left side:

```
if (column_index == 0)
  cell.west = nil
else
  cell.west = @columns[column_index - 1][row_index]
end
```

Set up the cells on the right side:

```
if (column_index == (x_cell_count - 1))
  cell.east = nil
else
  cell.east = @columns[column_index + 1][row_index]
end
```