SWINBURNE UNIVERSITY OF TECHNOLOGY

Introduction to Programming

Week 4 -Topic 1: Arrays and File Handling

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 you wrote for Task 2.2P:

```
YEAR_TRUMP_ELECTED = 2016

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

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

Design Note III:

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

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

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

Design Note IV:

Let us test the code and see if it works:

```ruby
def main
  puts difference(7, 10)
  puts difference(10, 7)
  puts calculate_age_when_trump_elected(1990)
  puts difference(YEAR_TRUMP_ELECTED, 1990)
end
```

```
MacBook-Pro:Code mmitchell$ ruby test.rb
3
3
26
26
MacBook-Pro:Code mmitchell$
```

KNOW ING

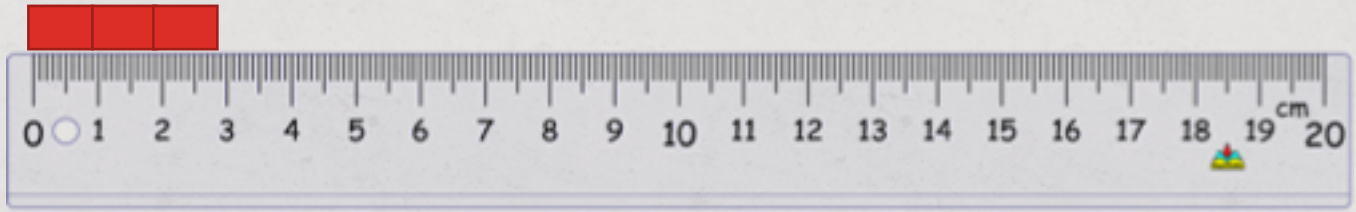
Arrays – The problem

- 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

0	**“Fred”**
1	“Sam”
2	“Jill”
3	“John”
4	“Jenny”

Number of elements: 5

Each element of the array is the **same** size.

Question: How long is a String?

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

Arrays – the solution III

0	**“Fred”**
1	“Sam”
2	“Jill”
3	“John”
4	“Jenny”

Number of elements: 5

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

Representation in Memory:

1024	1280	1536	1792	2048
“Fred”	**”Sam”**	**”Jill”**	**“John”**	**“Jenny”**
1024 + 0	1024 + 256	1024 + 512	1024 + 768	1024 + 1024

Creating Arrays I

- This is one way of creating Arrays:

```
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:

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

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

Accessing Arrays II

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

```
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
```

Get the number of elements

Index determines which element to print.

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

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:

```
genre_names[index] = read_string("Enter a value for the array: ")
```

Reading Multiple Values

- Just as we used a loop to print out multiple array values, now we can use a loop to read in multiple values:

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

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

Then we can print them as before:

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

Creating an Empty Array

- But 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:

```
genre_names = Array.new(4)
```

NB: *Ruby will* ***not*** *warn you if you try and access an element past the end of the array – it will simply return a nil value. C will also not warn you, it will give you whatever value is in that memory location.*

What happens if you try and change a value which is past the end of the array?

Dynamic Arrays

- So far we have worked with fixed sized arrays.
- These assume we know how many elements we will need when we write the program.
- If we don't know how many elements we need, then we probably want our array to "grow" to take as many elements as we need.
- These are called **dynamic** arrays, as the size is determined dynamically at run time.

Creating Dynamic Arrays

- We can create a dynamic array with no initial size:

```
name_array = Array.new()
```

- We use the "<<" operator to add elements, which automatically increases the size of the array by one:

```
name_array[index] << read_string("Enter next name: ")
```

- But we need to know how many elements to add. So we need something to determine the size. E.g:

```
count = read_integer("How many names: ")
```

Using Dynamic Arrays

- The final code looks like this:

```
def main
  name_array = Array.new()

  count = read_integer("How many names: ")
  index = 0
  while (index < count)
    name_array << read_string("Enter next name: ")
    index += 1 # Increment index by one
  end

  index = 0
  while (index < count)
    puts name_array[index]
    index += 1 # Increment index by one
  end
end
```

Multi-Dimensional Arrays

- Or we could even do:

```
col_count = read_integer("How many columns: ")
row_count = read_integer("How many rows: ")



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)
    table_array[row_index][col_index] = read_string(
      "Enter row #{row_index} column #{col_index} item: ")
    col_index += 1 # Increment index by one
  end
  row_index += 1
end
```

Multi-Dimensional Arrays

[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]

Summary:

- Arrays store multiple elements
- Arrays have a size (length)
- Arrays elements need to be accessed using an index.
- Arrays start the indexing at zero.
- Arrays can contain arrays.
- The contained arrays can be ‘ragged” (i.e of different lengths).

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 To Read

- 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:

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

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

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

Reading and Writing to Files

- To write to a file we use a version of **puts:**

```
a_file.puts('This is being written to the 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"
```

- 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.

Lets look at the Tasks

- Tasks for this week

The HD Maze Task

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

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

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