Week 3 - Topic 4: Selection and Iteration

Academic Integrity

Flow Control

This lecture we look at flow control. This makes up two of the three fundamental principles/elements of structured programming (coding aspects):

- Sequence (we looked at in previous weeks)
- Selection
- Iteration/repetition/looping

What we have done so far

So far our code has been based on sequences of statements:

Run RUBY

```ruby
def main()
  puts("Please enter your name: ")
  name = gets.chomp()
  puts("#{name} is a silly name")
end

main()
```

Making code more dynamic

Now we are going to look at how to make our code more dynamic:

Design for understanding

However, we need to carefully design our code so as to avoid creating ‘spaghetti code’:

Single entry/exit

We try and use blocks of code with one entry and one exit point:

Selection/Branching

Blocks can include decisions, or branches, where one of a variety of paths is taken: this is called selection :

Iteration/Looping/Repetition

The other kind of block is a loop, where instructions are repeated a number of times: this is called repetition - in these cases the code keeps looping around until a test condition is false:

Sequence, Selection, Repetition

Sequence, Selection, and Repetition: the key components of Structured Programming:

Flow for a Simple Program

Below is a control flow diagram for the Silly Name program

This branching is represented using an **if** statement:

Simple IF statements

Below is the simplest form of an *if* statement:

Run RUBY

```ruby
count = 5

if (count == 5)
  puts 'It matched!'
end
```

Notice the condition (double equals checks if two values are the same):

```ruby
(count == 5)
```

and the consequent:

```ruby
puts 'It matched!'
```

What happens in the above code if *count* is not equal to 5? (try it)

Common IF statements

Run RUBY

```ruby
# TRY CHANGING THE VALUE OF count BELOW then RUN:
count = 5

if (count == 5)  # Condition
  puts('It matched!') # Consequent
else
  puts('No match!') # Alternative
end
```

Conditional Comparisons

Conditional comparisons evaluate to be either true or false (i.e a Boolean value).

Different languages have different operators:

For Ruby's operators see :

See: https://www.tutorialspoint.com/ruby/ruby_operators.htm

Boolean conditions

Lets try some examples:

Single variable:

`(value > 10) or (value < 0)`

Multiple variables:

`(age == 25) and (value != 5)`

Try these out below:

Run RUBY

```ruby
value = 10

if (value > 10) or (value < 0)
  puts("The condition is true")
else
  puts("The condition is false")
end
```

Run RUBY

```ruby
age = 25
value = 5

if (age == 25) or (value != 5)
  puts("The condition is true")
else
  puts("The condition is false")
end

```

Boolean Truth tables

The following tables show when a condition is true based on the values of two variables (***A*** and ***B***):

Source: https://users.cs.jmu.edu/bernstdh/web/common/lectures/summary_logical-operators.php

Run RUBY

```ruby
# TRY CHANGING THE VALUES OF a AND b BELOW:

a = true
b = false

if (a and b)
  puts("AND: A and B are both True")
end

if (a or b)
  puts("OR: Either one of both of A or B is true")
end

if (a ^ b) # ^ is the XOR operator in Ruby
```

IF statements - variations

Remember the different types we saw:

IF-ELSE

Here are some IF-ELSE variations:

Run RUBY

```ruby
count = 5

if (count == 5)
  puts('It matched!')
else
  puts('No match!')
end

count = 3

if (count < 3)
  puts('Less than 3!')
elsif (count == 3)
  puts('Equal to 3!')
```

Write an IF statement

Write code that reads in a name, if the name is equal to "Jill" print "Awesome Name" otherwise print "That is a silly name".

UNLESS Statements

A variation on IF (available in Ruby but not C):

Run RUBY

```ruby
count = 5
unless (count == 5)
  puts('Not 5!')
else
  puts('It is equal to 5!')
end

puts('It is 5!') unless count != 5
puts('It is 5') if count == 5

```

CASE statements

Another type of conditional test is the *case* statement - this checks multiple possible conditions (in C the equivalent is a *switch* statement) .

Note: there are some problems with the code below - we need to fix them!

Run RUBY

```ruby
count = 10
case count
when 1..4
  puts('Not 5!')
when 5
  puts('It is 5!')
when 5..10
  puts('It is between 5 and 10')
else
  puts('It is less than zero or more than 10')
end

```

Case Statements – For Types

We can also have conditions based not on the **value** of a variable, but on its ***type***:

Run RUBY

```ruby
something = 10
case something
when Numeric
  puts("I'm a number.  My absolute value is #{something.abs}")
when Array
  puts("I'm an array.  My length is #{something.length}")
when String
  puts("I'm a string. In lowercase I am: #{something.downcase}")
else
  puts("I'm a #{something.class}")
end

```

Acknowledgement: http://rosettacode.org/wiki/Conditional_structures/Ruby

Iteration (loops)

There are two basic types of loops:

- Pre-test
- Post-test

We will see various forms of both.

Loops often require **control** variables – we will see these as we go.

When to use a post-test loop?

If you want the code in the body of the loop to execute at least once (i.e regardless of any conditional test), you use a post-test loop.

True

False

Examples for when to use loop types

Which of the following situations are likely to require a post-test loop?

- Reading from a file
- Reading from a network connection
- Reading a user's selection from a set of menu options
- Incrementing a counter from zero to a number ranging from zero to 10 (as entered by a user)

Pre-Test Loop - WHILE

Following is a while loop - which is a *pre-test* loop.

This is the most common loop structure.

Notice the use of *count* as a **control variable** to determine when to stop the loop:

Run RUBY

```
1 count = 5
2 while (count < 5)
3   puts('Count is ' + count.to_s())
4   count = count + 1
5 end
6
```

Pre-Test Loops – FOR

Notice the for loop auto-increments the counter
(i.e no need for `count = count + 1`):

Many languages (including C) have a type of FOR loop structure:

Run RUBY

```ruby
count = 5
for count in 0..4
  puts('Count is ' + count.to_s())
end
```

Write a WHILE loop

Write a **while** loop that counts to 6 producing output as follows:

```
[user@sahara ~]$ ruby loop.rb
1
2
3
4
5
6
[user@sahara ~]$
```

Post Test Loops – UNTIL

One option:

Run RUBY

```ruby
count = 0

begin
  puts('my line ' + count.to_s())
  count = count + 1
end until (count == 6)
```

Or Equivalently:

Run RUBY

```ruby
count = 0

begin
  puts('my line ' + count.to_s())
  count = count + 1
end while (count < 6)

```

Note: see: http://rosettacode.org/wiki/Loops/Do-while#Ruby

BREAK statements

It is possible to use a *break* statement to 'jump' out of a loop or *if* statement:

Two variations are below - but these are not recommended in Ruby (often used in C):

Run RUBY

```ruby
count = 0

loop do
  puts('my line ' + count.to_s())
  count += 1
  break if (count == 2)
end

count = 0

loop do
  puts('my line ' + count.to_s())
  count += 1
  break unless (count != 2)
```

TIMES loop

We can execute a block of code a certain number of times:

Run RUBY

```ruby
5.times do
  puts('Enter a name: ')
  name = gets.chomp
  puts('Name is ' + name)
end

```

REDO

Another option for post-test loops (in Ruby) is REDO:

Run RUBY

```ruby
1.times do
  puts('Enter a name: ')
  info = gets.chomp()
  unless (info == "exit")
    puts('You entered: ' + info)
    redo
  end
end

puts("second loop")

1.times do
  puts('Enter a name: ')
  info = gets.chomp()
```

Variable TIMES Loop

FredInstead of hard-coding the number of times to loop, we can use a variable:

Run RUBY

```ruby
count = 5
count.times do
  puts(count.to_s() + ' Enter a name: ')
  name = gets.chomp()
  puts('Name is ' + name)
end

```

Accessing the TIMES variable

If we want to know where we are up to in terms of how many times we have been through the loop already we can access the otherwise hidden control variable we give the variable whatever name we want, in this case *i*:

Run RUBY

```ruby
count = 5
count.times do |i|
  puts(i.to_s() + ' Enter a name: ')
  name = gets.chomp()
  puts('Name is ' + name)
end
```

UPTO loop

The *upto* loop also auto-increments the counter
(i.e no need for `count = count + 1`):

Run RUBY

```ruby
1.upto(5) do
  name = gets.chomp()
  print('Name is ' + name)
end
```

NESTED loops

You can have loops inside loops (called **nested** loops):

Run RUBY

```ruby
1.upto(5) do |i|
  puts(i.to_s() + ' Enter runner\'s names in order of completion: ')
  name = gets.chomp()
  print('Name is ' + name)
  points = 0
  i.upto(3) do |j|
    points += j
  end
  puts('. You get ' + points.to_s() + ' points')
end
```

Tasks for this week

- Silly name task
- Submenu task
- Shape Drawing Task

Loops and Conditions in structure charts

Representing Conditions and Loops in Structure Charts.

See the video for a structure chart for the Silly name Task:

https://echo360.org.au/media/b54387d1-4df6-47a1-a55a-e52c74514a15/public

Then the following shows the Simple Menu Program:

Graphical Programming (shape drawing)

We use a co-ordinate system to place things on the screen:

Graphical Programming: Rectangles

Drawing a rectangle:

```
Gosu.draw_rect(1, 1, 7, 3, Gosu::Color::BLUE, ZOrder::TOP, mode=:default)
```

Shape Drawing using GOSU

Follow the 3 steps below:

RUBY

```ruby
require 'rubygems'
require 'gosu'
require './circle'

# The screen has layers: Background, middle, top
module ZOrder
  BACKGROUND, MIDDLE, TOP = *0..2
end

class DemoWindow < Gosu::Window
  def initialize
    super(640, 400, false)
  end

```

Use the following site to select colours for the circle (which uses RGB values):
https://www.rapidtables.com/web/color/RGB_Color.html
Or
Use the Gosu colour constants:
https://www.rubydoc.info/github/gosu/gosu/master/Gosu/Color

1 Copy the code from the box above into your editor then save it is as *gosu_shapes.rb*.

RUBY

```ruby
require "rubygems"
require "gosu"

class Circle
  attr_reader :columns, :rows

  def initialize(radius)
    @columns = @rows = radius * 2

    clear, solid = 0x00.chr, 0xff.chr

    lower_half = (0...radius).map do |y|
      x = Math.sqrt(radius ** 2 - y ** 2).round
      right_half = "#{solid * x}#{clear * (radius - x)}"
```

2 Copy the code above into your editor and save it as *circle.rb*.

3 Run *ruby gosu_shapes.rb* It should look as follows: - then change it to make your own picture.

Use this link to help you: https://www.rubydoc.info/github/gosu/gosu/master/Gosu

Drawing Shapes using GOSU - Demonstration

NOTE: To run Graphical programs in this tool (ED) - you need to use the Run button for the task (not the terminal).

In this task you use Gosu to create a program that draws a picture.

GOSU is a development environment that makes it easy to create programs that use graphics, sounds, animations and other aspects relevant to creating small interactive games.

Follow these **3** steps:

1. Copy the code provided to your IDE (both `gosu_shapes.rb` and `circle.rb`) and use the demonstration shapes to create a picture of your own design. Your picture should include at least 3 different types of shapes (eg: a triangle, a rectangle and a circle)

> Use the following site to select colours for the circle (which uses RGB values):
> https://www.rapidtables.com/web/color/RGB_Color.html
> Or Use the Gosu colour constants:
> https://www.rubydoc.info/github/gosu/gosu/master/Gosu/Color

> eg: a Red circle with a radius of 50 pixels would be produced by the two statements:

```
img = Gosu::Image.new(Circle.new(50))
img.draw(200, 200, ZOrder::TOP, 0.5, 1.0, Gosu::Color::RED)
```

> Or you could use the HEX values:

```
img.draw(300, 50, ZOrder::TOP, 1.0, 1.0, 0xff_ff0000)
```

> See here to work out HEX values:
> https://www.binaryhexconverter.com/decimal-to-hex-converter

2. Run the code provided and study it to understand what is happening (you may need to run the command `gem install rubygems` to use the `circle.rb` code).

3. Using an IDE like Visual Studio Code, change the code to draw your own unique picture. You might want to draw it on paper first (perhaps graph paper or an electronic equivalent like this).

> The co-ordinate system works as follows

> Example:

Once your code is complete submit a copy and a screen shot to Doubtfire.

Resources

Sobkowicz, M 2015 *Learn game programming with Ruby : bring your ideas to life with Gosu*, The Pragmatic Bookshelf (See chapter 7 for help the grid aspect of the Maze Task)

Gosu Ruby Documentation

Gosu site

Gosu game video tutorial