UNIT - 2

The first thing we need to look at is how to represent and access Ruby datatypes from within C. Everything in Ruby is an object, and all variables are references to objects. In C, this means that the type of all Ruby variables is VALUE, which is either a pointer to a Ruby object or an immediate value (such as Fixnum).

This is how Ruby implements object-oriented code in C: a Ruby object is an allocated structure in memory that contains a table of instance variables and information about the class. The class itself is another object (an allocated structure in memory) that contains a table of the methods defined for that class. On this foundation hangs all of Ruby.

Ruby is an ideal object-oriented programming language. The features of an object-oriented programming language include data encapsulation, polymorphism, inheritance, data abstraction, operator overloading etc. In object-oriented programming classes and objects plays an important role. A **class** is a blueprint from which objects are created. The **object** is also called as an *instance of a class*. For Example, the animal is a class and mammals, birds, fish, reptiles, and amphibians are the instances of the class. Similarly, the sales department is the class and the objects of the class are sales data, sales manager, and secretary.

Defining a class in Ruby:

In Ruby, one can easily create classes and objects. Simply write class keyword followed by the name of the class. The first letter of the class name should be in capital letter.

Syntax:

```
class Class_name


end
```

A class is terminated by **end keyword** and all the data members are lies in between class definition and end keyword.
Example:

```
# class name is Animal
class Animal

# class variables
@@type_of_animal = 4
@@no_of_animal = 3

end
```

Creating Objects using the “new” method in Ruby:

Classes and objects are the most important part of Ruby. Like class objects are also easy to create, we can create a number of objects from a single class. In Ruby, objects are created by the **new method**.

Syntax:

```
object_name = Class_name.new
```

Example:

```
# class name is box
class Box

# class variable
@@No_of_color = 3

end

# Two Objects of Box class
sbox = Box.new
nbox = Box.new
```

Here Box is the name of the class and No_of_color is the variable of the class. *sbox* and *nbox* are the two objects of box class. You use (=) followed by the class name, dot operator, and new method.

Defining Method in Ruby:

In Ruby member functions are called as methods. Every method is defined by the **def keyword** followed by a method name. The name of the method is always in lowercase and the method ends with **end keyword**. In Ruby, *each class and methods end with end keyword*.
Syntax:

```
def method_name


# statements or code to be executed


end
```

Example:

```
# Ruby program to illustrate

# the defining of methods

 #!/usr/bin/ruby

 # defining class Vehicle

class GFG

 # defining method

def geeks

 # printing result
```

```
puts "Hello Geeks!"

 # end of method

end

 # end of class GFG

end

 # creating object

obj = GFG.new

 # calling method using object

obj.geeks
```

Output:

```
Hello Geeks!
```

Passing Parameters to new Method:

User can pass any numbers of parameters to “new method” which are used to initialize the class variables. While passing parameters to “new method” it is must to declare an **initialize** method at the time of class creation. The initialize method is a specific method, which executes when the new method is called with parameters.

Example:

```
 # Ruby program to illustrate the passing

# parameters to new method

 #!/usr/bin/ruby
 # defining class Vehicle

class Vehicle

# initialize method

def initialize(id, color, name)

 # variables

@veh_id = id

@veh_color = color

@veh_name = name


# displaying values

puts "ID is: #@veh_id"

puts "Color is: #@veh_color"

puts "Name is: #@veh_name"

puts "\n"

end

end


# Creating objects and passing parameters

# to new method
```

```
xveh = Vehicle. new("1", "Red", "ABC")

yveh = Vehicle. new("2", "Black", "XYZ")
```

Output:

```
ID is: 1
Color is: Red
Name is: ABC


ID is: 2
Color is: Black
Name is: XYZ
```

Explanation: Here Vehicle is the class name. **def** is a keyword which is used to define **“initialize”** method in Ruby. It is called whenever a new object is created. Whenever new class method called it always call *initialize* instance method. **initialize** method is like a constructor, whenever new objects are created *initialize method* called. *Id, color, name*, are the parameters in initialize method and *@veh_id @veh_color, @veh_name* are the local variables in initialize method with the help of these local variables we passed the value along the new method. The parameters in “new” method is always enclosed in double quotes.

The Jukebox extension:

Jukebox is a component to handle the playback of music and sound effects across multiple web browsers and operating systems.

```
#Jukebox.rb
require_relative
'./song_library.rb'
                    def jukebox(command)
                      if command.downcase == "list"
                        list_library
                      else
```

```
    parse_command(command)
  end
end
def list_artist(artist, album_hash)
   artist_list = "\n---------------\n"
   artist_list += "#{artist}:\n"
   artist_list += "---------------"
   album_hash[:albums].each do |album_name, songs_hash|
     artist_list += "\n#{album_name}:\n\t"
     artist_list += songs_hash[:songs].join("\n\t")
   end
   artist_list
end
def list_library
  lib = full_library
  lib.each do |artist, album_hash|
    puts list_artist(artist, album_hash)
  end
end
def parse_command(command)
  parse_artist(command, full_library) ||
play_song(command, full_library) || not_found(command)
end
def parse_artist(command, lib)
  cmd = command.downcase.to_sym
  parsed = false
  if lib.has_key?(cmd)
    puts list_artist(command, lib[cmd])
    parsed = true
  else
    lib.each do |artist, album_hash|
      if command.downcase == artist.to_s.gsub("_","
").downcase
        puts list_artist(artist, album_hash)
        parsed = true
        break
      end
    end
  end
  parsed
```

```
end
def play_song(command, lib)
  lib.each do |artist, hash|
    hash.each do |album_name, albums_hash|
      albums_hash.each do |album, songs_hash|
        songs_hash.each do |songs, song_array|
          song_array.each do |song|
            if song.downcase == command.downcase
            puts "Now Playing: #{artist.to_s.strip}: #{album} -
#{song}\n\n"
            return true
          end
        end
      end
    end
  end
  false
end
def not_found(command)
  puts "I did not understand '#{command}'!\n\n"
  true
end
end
```

#runner.rb

```
require_relative
'./jukebox.rb'
def run
  puts "Welcome to Ruby Console Jukebox!"
  command = ""
  while command.downcase != "exit" do
    puts "Enter a command to continue. Type 'help' for a list of
commands."
    command = get_command
    run_command(command) unless command.downcase ==
"exit"
  End
```

```
End
def get_command
  gets.strip
End
def run_command(command)
  case command
  when "help"
    show_help
  Else
    jukebox(command)
  End
End
def show_help
  help = "Never worked a jukebox, eh? Pretty standard.
Available commands are:\n"
  help += "'help' - shows this menu\n"
  help += "'list' - lists the whole song library\n"
  help += "or you can enter an artist's name to show that
artist's songs\n"
  puts help
End
Run
```

#song_library.rb

```
def
full_library
            {
              :"U2" => {
                :albums => {
                  :"The Joshua Tree" => {
                    :songs => ["With or Without You", "Still Haven't Found
          What I'm Looking For", "Bullet the Blue Sky"]
                  },
                  :"Zooropa" => {
                    :songs => ["Numb"]
                  }
                }
              },
              :"Talking Heads" => {
                :albums => {
```

```
        :"Fear of Music" => {
          :songs => ["Life During Wartime", "Heaven"]
        },
        :"Speaking in Tongues" => {
          :songs => ["This Must Be the Place (Naive Melody)",
"Burning Down the House"]
        }
      }
    },
    :"Huey Lewis and the News" => {
      :albums => {
        :"Sports" => {
          :songs => ["I Want a New Drug", "If This is It", "Heart of
Rock and Roll"]
        }
      }
    }
  }
End
```

Memory Allocation

You may sometimes need to allocate memory in an extension that won't be used for object storage---perhaps you've got a giant bitmap for a Bloom filter, or an image, or a whole bunch of little structures that Ruby doesn't use directly.
In order to work correctly with the garbage collector, you should use the following memory allocation routines. These routines do a little bit more work than the standard malloc. For instance, if ALLOC_N determines that it cannot allocate the desired amount of memory, it will invoke the garbage collector to try to reclaim some space. It will raise a NoMemError if it can't or if the requested amount of memory is invalid.

Memory Allocation

type **ALLOC_N**(*c-type*, n")

Allocates *n* *c-type* objects, where *c-type* is the literal name of the C type, not a variable of that type.

type **ALLOC**(*c-type*")

Allocates a *c-type* and casts the result to a pointer of that type.

REALLOC_N(*var*, *c-type*, n")

Reallocates *n* *c-type*s and assigns the result to *var*, a pointer to a *c-type*.

type **ALLOCA_N**(*c-type*, n")

Allocates memory for *n* objects of *c-type* on the stack---this memory will be automatically freed when the function that invokes ALLOCA_N returns.

Which method allocate memory for objects in Ruby?

Ruby releases a small amount of empty pages (a set of slots) at a time when there is too much memory allocated. The operating system call to **malloc** , which is currently used to allocate memory, may also release freed memory back to the operating system depending on the OS specific implementation of the malloc library.11-May-2015

How is memory allocated?

There are two basic types of memory allocation: **When you declare a variable or an instance of a structure or class. The memory for that object is allocated by the operating system**. The name you declare for the object can then be used to access that block of memory.

Steps to Find a Memory Leak

1. Check for any unused gems in the Gemfile and remove them. ...
2. Check the issue tracker of each gem still present in the Gemfile for reports of memory leaks. ...
3. Run Rubocop with the rubocop-performance extension. ...
4. Visually review the Ruby code for possible memory leaks.

The different storage allocation strategies are :

- Static allocation - lays out storage for all data objects at compile time.
- Stack allocation - manages the run-time storage as a stack.
- Heap allocation - allocates and deallocates storage as needed at run time from a data area known as heap.

Ruby type systems:

Ruby is a **dynamically typed language**, which means the interpreter tries to infer the data type of variables and object properties at runtime. This generally leads to programs being more dynamic and easier (faster) to code, and the interpreter/compiler loading ode faster.

Background

Static typing versus dynamic typing is an age old issue for programming languages. Statically typed languages are suitable for larger projects but are often less flexible. Dynamically typed languages allow for rapid development, but scaling teams and codebases with them can be difficult.
Dynamically typed languages implement type checking options (PHP, Python)

RBS:

We defined a new language called RBS for type signatures for Ruby 3. The signatures are written in .rbs files which is different from Ruby code. You can consider the .rbs files are similar to .d.ts files in TypeScript or .h files in C/C++/ObjC. The benefit of having different files is it doesn't require changing Ruby code to start type checking. You can opt-in type checking safely without changing any part of your workflow.

Key features in RBS

The development of a type system for a dynamically typed language like Ruby differs from ordinal statically typed languages. There's a lot of Ruby code in the world already, and a type system for Ruby should support as many of them as possible.

Duck typing

Duck typing is a popular programming style among Rubyists that assumes an object will respond to a certain set of methods. The benefit of duck typing is flexibility. It doesn't require inheritance, mixins, or *implement* declarations. If an object has a specific method, it works. The problem is that this assumption is hidden in the code, making the code difficult to read at a glance.

To accomodate duck typing we introduced *interface types*. An interface type represents a set of methods independent from concrete classes and modules.

If we want to define a method which requires a specific set of methods we can write it with *interface types*.

Interface_Appendable

```
# Requires `<<` operator which accepts `String` object.
Def<<:(String) -> void
end

# Passing `Array[String]` or `IO` works.
# Passing `TrueClass` or `Integer` doesn't work.
def append: (_Appendable) -> String
```

Non-uniformity

Non-uniformity is another code pattern of letting an expression have different types of values. It's also popular in Ruby and introduced:

- When you define a local variable which stores instances of two different classes
- When you write an heterogeneous collection
- When you return two different types of value from a method

Ruby programming with types

- **inding more bugs:** We can detect an undefined method call, an undefined constant reference, and more things a dynamic language might have missed.
- **Nil safety:** Type checkers based on RBS have a concept of *optional types*, a type which allows the value to be nil. Type checkers can check the possibility of an expression to be nil and uncovers undefined method(save!)' for nil:NilClass`.
- **Better IDE integration:** Parsing RBS files gives IDEs better understanding of the Ruby code. Method name completions run faster. On-the-fly error reporting detects more problems. Refactoring can be more reliable!
- **Guided duck typing:** Interface types can be used for duck typing. It helps the API users understand what they can do more precisely. This is a safer version of duck typing.

Ebedding Ruby to Other Languages:

When you decide to use an interpreted language such as Ruby, you e trading raw speed for ease of use. Its far easier to develop a program in a higher-level language, and you get a working program faster, but you sacrifice some of the speed you might get by writing the program in a lower-level language like C and C++.

Thats the simplified view. Anyone whos spent any serious amount of time working with higher-level languages knows that the truth is usually more complex. In many situations, the tradeoff doesn really matter: if the program is only going to be run once, who cares if it takes twice as long to do its job? If a program is complex enough, it might be prohibitively hard to implement in a low-level language: you might never actually get it working right without using a language like Ruby.

But even Ruby zealots must admit that there are still situations where its useful to be able to call code written in another language. Maybe you need a particular part of your program to run extremely fast, or maybe you want to use a particular library thats implemented in C or Java. When that happens youll be grateful for Rubys extension mechanism, which lets you call C code from a regular Ruby program; and the JRuby interpreter, which runs atop the Java Virtual Machine and uses Java classes as though they were Ruby classes.

Compared to other dynamic languages, its pretty easy to write C extensions in Ruby. The interfaces you need to understand are easy to use and clearly defined in just a few header files, there are numerous examples available in the Ruby standard library itself, and there are even tools that can help you access C libraries without writing any C code at all.

Embedding a Ruby Interpreter:

In addition to extending Ruby by adding C code, you can also turn the problem around and embed Ruby itself within your application. Here's an example.

```
#include "ruby.h"

main() {
  /* ... our own application stuff ... */
  ruby_init();
  ruby_script("embedded");
  rb_load_file("start.rb");
  while (1) {
    if (need_to_do_ruby) {
      ruby_run();
    }
    /* ... run our app stuff */
  }
}
```

To initialize the Ruby interpreter, you need to call ruby_init(). But on some platforms, you may need to take special steps before that:

```
#if defined(NT)
  NtInitialize(&argc, &argv);
#endif
#if defined(__MACOS__) && defined(__MWERKS__)
```

```
  argc = ccommand(&argv);
#endif
```

See main.c in the Ruby distribution for any other special defines or setup needed for your platform.

Embedded Ruby API

Void **ruby_init**(")

Sets up and initializes the interpreter. This function should be called before any other Ruby-related functions.

Void **ruby_options**(int argc, char **argv")

Gives the Ruby interpreter the command-line options.

Void **ruby_script**(char *name")

Sets the name of the Ruby script (and $0) to *name*.

Void **rb_load_file**(char *file")

Loads the given file into the interpreter.

Void **ruby_run**(")

Runs the interpreter.