Rogue Language Overview

Abe Pralle

February 1, 2017

v1.0

What Is Rogue?

Classification

- High-end mid-level language
- General-purpose
- Object-oriented
- Imperative
- Statically typed
- Garbage-collected
- Compiled

Philosophy

- Practical, Ergonomic, Efficient, Readable

Key Features

- Cross-compiles to fast, standalone C++
- Easily interoperates with C++
- Good mix of both original and existing language ideas
- Self-hosting
- Embeddable per-project - does not require monolithic install

Installing Rogue

Git Repo

- github.com/AbePralle/Rogue

Compatibility

- macOS, Linux (Ubuntu), Cygwin

Building

> make

Compiles bootstrap C++ and installs /usr/local/bin/roguec script

> roguec

Rogue Compiler

Editor Plug-ins in Syntax/ Folder

- Vim
- Sublime Text 3

Documentation

- GitHub Wiki
- This presentation (available on Wiki)
- Standard Library source code in Source/Libraries/Standard
- Questions welcome at abe.pralle@gmail.com

Hello World

Hello.rogue

```
println "Hello World!"
```

Output

```
Hello World!
```

Compiling

```
> roguec Hello.rogue --execute
```

- No main method or class
- Global scope commands are collected and executed when Rogue is launched
- Standard library is automatically included
- Monolithic compile - no partial compilation

General Conventions

Identifiers

- Lower camelCase for keywords
- Upper CamelCase for type names
- serpent_case for vars and methods
- Avoids abbreviations

Regulation

- Generally unregulated
- No private/public/final
- Not sandboxed
- Unchecked array access
- Language structure avoids common pitfalls

Statements

- Semicolons can be used to separate statements on a single line but are not otherwise required
- Incomplete expressions automatically continue on next line
- Ellipsis (...) explicit line continuation

Recurring Symbols

@	Literal or actual
$	Meta or substitution
->	Convert or produce

$include and $define

IncludeDemo.rogue

```
$include "Message.rogue"
$define WHOA true
println MESSAGE
```

Message.rogue

```
$define MESSAGE "Hello World!"
if ($defined(WHOA)) println "Whoa!"
```

Output

```
Hello World!
```

Preprocessor Directives

- Preprocessor directives start with $
- First file should directly or indirectly $include other source files
- Files are preemptively included
- Include order only matters for $define'd symbols and global-scope commands
- Standard library is automatically included

Comments

Single Line

```
# Single line comments start with '#'
# and extend to end of line.
```

Multi-line

```
#{ Multi-line comments can span
multiple lines.  Multi-line comments
can be #{nested}# }#
```

Notes

- Rogue's convention is to put comments *after* classes, properties, and methods but *before* other blocks of code

Notes

```
class Clock
  # Models 12-hour clock HH:MMxm
  PROPERTIES
    minutes : Int32  # H&M combined

  METHODS
    method init( h:Int32, m:Int32,
          is_pm:Logical )
      # Initializes this clock
      if (h == 12 and not is_pm) h = 0
      elseIf (h != 12 and is_pm) h += 12

      # Combine hours & minutes
      minutes = h * 60 + m
```

trace

TraceTest.rogue

```
trace 3+4
@trace 3+4
@trace "Easy as ", pi
```

Output

```
[Global.on_launch() TraceTest.rogue:1] 3+4:7
3+4:7
Easy as pi:3.141592653589793
```

Notes

- trace is an improvement to println debugging
- Prints source context by default, @trace suppresses
- Literal strings print as-is
- Other expressions log out the source expression followed by the expression value

Primitive Datatypes

Real64	1.0, 1e6, pi, NaN, infinity	
Real32	Real32(1.0), 1.0->Real32	
Int64	(Integer larger than 32-bit)	
Int32	1, 0xA0, 0c240, 0b1010_0000	_ ignored in literal numbers
Character	'A', '\xHH', '\uHHHH', '\[H+]'	Full 21-bit Unicode in 32 bit int
Byte	Byte(0), 255->Byte	Unsigned 0..255
Logical	true, false	

Operators

Arithmetic

+ - * /	Standard
%	-2%5==3, 6%2.5==1.0
^	Power, 2^3 == 8

Logical

and, or	Lazy evaluation
xor	
not	not true == false
?	Logicalize - if (x?)

Bitwise

&, |	Bitwise AND, OR
~	Bitwise XOR
!	Bitwise complement
:<<:	Left shift w/0 fill
:>>:	Right shift w/0 fill
:>>>:	Right shift w/sign extend

Notes

- Logical operators are words, bitwise operators are symbols

Operators

Relational

== !=	Overloaded on objects
< <= > >=	
<>	Compare - gives 1,0,-1
is, is not	Test reference equality
instanceOf	Also: 'not instanceOf'

Steppers

++, --	++n same as n++

- Steppers are standalone statements - cannot use in larger expressions
- Works on real numbers

Assignment

=	'a = b' assigns b to a
+=	Add and assign
-=	Other *modify and assign*
***=**	...
/=	
%=	
^=	
&=	
|=	
~=	

Operators

Ranges

..	Up to (1..10)
..<	Up to but not including
downTo	(10 downTo 1)
..>	Down to, not including
... step x	Optional for any range

- Optional 'step' can be specified with all ranges: '9 downTo 1 step -2'
- "Standalone" ranges produce 'Range' objects; "forEach" ranges are optimized out

Conversion

as	Reinterpret or null
->	Convert to type

- (x as String) results in either a String reference or else *null* if x is not a String
- 3->Real64 converts 3 to 3.0
- 3->String converts 3 to "3" (uses previously defined conversion method)

select

Standard select

- select{*<cond>*:*<expr>* || *<expr2>*}
- select{*<c1>*:*<e1>*||*<c2>*:*<e2>*||*<e3>*}
- select{...}

select-which

- select(*<expr>*){*<val>*:*<e1>*||*<e2>*}
- select(*<expr>*){...}

Notes

- Similar to C's *conditional operator* AKA *decision operator* or *ternary operator* (c ? x : y)
- Evaluates to one of the possible expressions based on the conditions checked
- Can include any number of *condition:result* pairs and then must end with a single default result

Local Variables

LocalTest.rogue

```
local a=3, b=4
local x, y : Real64
local z=true : Logical
@trace a, b, x, y, z
```

Output

```
a:3 b:4 x:0.0 y:0.0 z:true
```

Description

- Optional type name comes after list of variable names, not after each variable
- Type inferenced from initial value if not specified
- "=" assigns value, ":" declares type

Literal Strings

Standard Double-Quoted

```
"This is a string."
```

Single-quoted (must be > 1 char)

```
'My name is "Abe".'
```

Double-single-quoted

```
''I'm using "Rogue".''
```

Verbatim

```
@|This defines the exact
 |string typed, including
 |newlines and ignoring any
 |escapes like \n.
```

Escape Sequences

\n, \r, \t	Newline, CR, tab
\b, \f, \v	Bell, Form Feed, Vert tab
\0	Null terminator/Unicode 0
\e	Escape (Unicode 27)
****	Single \ (backslash)
\', \"	One single or dbl quote
\xHH	Char w/2-digit hex code
\uHHHH	Char w/4-digit hex code
\[H+]	Char w/1-6 hex digits `\[1F61C]` -> 😜

Formatted Strings

StringFormat.rogue

```
local a=3, b=4
println "$ + $ = $" (a,b,a+b)
```

Output

```
3 + 4 = 7
```

Description

- General syntax: "format" (args)
- Uses "fill in the blank" style rather than embedding expressions in string
- Each '$' *marker* in *format* string replaced with corresponding arg
- Converted to series of string builder concatenations at compile time
- Format must be literal string, cannot be a variable

Optional Values

OptionalValues.rogue

```
local num : Int32?
println num  # null
println num.exists # false
println num?  # false

num = 5
println num  # 5
println num.exists # true
println num?  # true
println num.value # 5

num = null
println num # null
```

Notes

- Any value can be specified as an *optional value* by declaring it as an *optional type* (add '?' after the type name)
- Optional values can be assigned 'null' or any valid value
- Use num.exists or num? to check existence
- Use num.value to access value after checking for existence
- Compiler does not enforce null-checking

Control Structures

Conditionals

- if / elseIf / else / endIf
- which / case / others / endWhich
- contingent / necessary / sufficient / satsified / unsatisfied / endContingent

Miscellaneous

- try / catch / endTry
- block / endBlock

Loops

- loop / endLoop
- while / endWhile
- forEach / endForEach

Dynamic Flow Modifiers

- nextIteration (AKA continue)
- escapeIf (AKA break)
- escapeWhich
- escapeContingent
- escapeTry, escapeBlock
- escapeLoop, escapeWhile
- escapeForEach

if, which

General Multi-line if

if (*condition*)
  statements
elseIf (*condition*)
  statements
else
  statements
endIf

Single Line (AKA Same Line) if

if (condition) statements
elseIf (condition) statements
else statements

which

which (*expression*)
  case *expression*
    statements
  case *expression*: *statements*
  others
    statements
endWhich

contingent

contingent

```
contingent
  # Any number & mix of the following
  statements

  # Skip to unsatisfied if condition false
  necessary (condition)

  # Skip to satisfied if condition true
  sufficient (condition)
satisfied
  statements
unsatisfied
 statements
endContingent
```

FindPrimes.rogue

```
forEach (n in 1..20)
  contingent
    necessary (n > 1)
    sufficient (n == 2)
    necessary (n % 2)?
    forEach (d in 3..(n.sqrt) step 2)
      necessary (n % d)?
    endForEach
  satisfied
    println "$ is prime" (n)
  endContingent
endForEach
```

loop, while

Infinite loop

loop
  statements
endLoop

Fixed loop

loop (*count-expression*)
  statements
endLoop

loop (*count-expression*) *statements*

while

while (*condition*)
  statements
endWhile

while (*condition*) *statements*

forEach

forEach

forEach (*control*)
  statements
endForEach

forEach (*control*) *statements*

Iteration Control

forEach (*range*)
forEach (*item* in *collection* [step n])
forEach (*index* of *collection*)
forEach (*item* at *index* in *collection*)
forEach (... in/of *local_var*=*collection*)

Implicit forEach

- Any term of an expression may be one of the following:
  (forEach in *collection*)
  (forEach of *collection*)
- Statements containing such *implicit forEach* loops are wrapped in an appropriate forEach and the original term replaced by the control variable
- Examples:
  println (forEach in 1..5)
  total += (forEach in list).count

try, block

try

```
try
  statements
catch (var:MoreSpecificType)
  statements
catch (var:MoreGeneralType)
  ...
endTry
```

throw

```
throw ExceptionType(message)
```

block

```
block
  statements
endBlock
```

- Similar to a {...} code block in C
- Can be used to limit local var scope
- Can be used as an escape point with *escapeBlock*

Basic Classes

Hourglass.rogue

```
class Hourglass
  PROPERTIES
    top = 5
    bottom : Int32

  METHODS
    method tick
      if (not times_up)
        --top
        ++bottom
      endIf
```

Hourglass.rogue (continued)

```
    method times_up->Logical
      return (top == 0)

    method to->String
      return "$/$" (top,bottom)

    method turn_over
      local temp = top
      top = bottom
      bottom = temp

endClass
```

Basic Classes

Hourglass.rogue (continued)

```
class Hourglass ... endClass

local hourglass = Hourglass()
println hourglass
while (not hourglass.times_up)
  hourglass.tick
  println hourglass
endWhile
hourglass.turn_over
println hourglass
```

Compile & Execute

```
> roguec Hourglass.rogue --execute
    5/0
    4/1, 3/2, 2/3, 1/4, 0/5
    5/0
```

- TypeName([args]) instantiates object - parens req'd but no "new"
- Parens not required on no-args method calls
- Class definitions are split into sections such as PROPERTIES and METHODS
- *Properties* are AKA *instance variables*

Auto-store Parameters

HourglassV2.rogue

```
class Hourglass
  PROPERTIES
    top, bottom : Int32

  METHODS
    method init( top=5, bottom=0 )

    method tick->this
      if (not times_up) --top; ++bottom
      return this
```

HourglassV2.rogue (continued)

```
    method times_up->Logical
      return (top == 0)

    method to->String
      return "$/$" (top,bottom)

    method turn_over->this
      swapValues top, bottom
      return this

endClass
```

Routines

HourglassV2.rogue (continued)

```
class Hourglass ... endClass

test( Hourglass(6) )

routine test( hourglass:Hourglass )
  println hourglass
  while (not hourglass.times_up)
    println hourglass.tick
  endWhile
  println hourglass.turn_over
endRoutine
```

Compile & Execute

- Class constructors are named *init*
- Classes have *methods*
- *routines* are global methods defined outside of a class scope
- A method parameter without a type is an *auto-store parameter* that automatically stores the passed value in the corresponding property
- Method parameters can have default values

Property Parameters

HourglassV3.rogue

```
class Hourglass( top=5:Int32,
    bottom=0:Int32 )
  METHODS
    method tick->this
      ...
```

Notes

- A class can define *property parameters* immediately after the class name
- Property parameters implicitly define properties and an *init()* constructor that auto-stores those properties

Getters and Setters

Distance.rogue

```
class Distance
  PROPERTIES
    cm : Real64

  METHODS
    method inches->Real64
      return cm / 2.54

    method set_inches( n:Real64 )
      cm = n * 2.54
endClass
```

DistanceTest.rogue

```
$include "Distance.rogue"

local ruler = Distance()
ruler.inches = 12
@trace ruler.cm
# ruler.cm:30.48
```

Notes

- Can use getters and setters (access methods) to create faux properties
- Rogue downplays difference between properties and access methods

Direct Property Access

Percent.rogue

```
class Percent( value=0:Int32 )
  METHODS
    method set_value( n:Int32 )
      if (n < 0) n = 0
      elseIf (n > 100) n = 100
      @value = n
endClass
```

Notes

- "@x" is a "direct access" that bypasses getters and setters while reading or writing a property
- percent.@value = 110

Direct Auto-store Parameter

```
method set_value( @value )
  if (value < 0) @value = 0
  elseIf (value > 100) @value = 100
```

Using clamped() from API

```
method set_value( n:Int32 )
  @value = n.clamped( 0, 100 )
```

Global Methods

DistanceV2.rogue

```
class Distance
  GLOBAL METHODS
    method cm( n:Real64 )->Distance
      return Distance().set_cm( n )

    method inches( n:Real64 )->Distance
      return Distance().set_inches( n )

  PROPERTIES
    cm : Real64
```

DistanceV2.rogue

```
  METHODS
    method inches->Real64
      return cm / 2.54

    method set_cm( @cm )->this
      return this

    method set_inches( n:Real64 )->this
      cm = n * 2.54; return this

    method to->String
      return "$ cm / $ inches" (cm,inches)
endClass
```

Global Methods

DistanceTestV2.rogue

```
@trace Distance.inches(12)
@trace Distance.cm(100)
```

Output

```
Distance.inches(12):30.48 cm /
  12.0 inches
Distance.cm(100):100.0 cm /
  39.37007874015748 inches
```

Notes

- Global methods are AKA static methods in C++/Java/C#
- It is useful to have setters return *this* to allow call chaining during construction
- GLOBAL PROPERTIES can be defined (AKA static variables or class variables in Java/etc.)

Singletons

Singletons.rogue

```
class Settings [singleton]
  PROPERTIES
    gravity = 9.8
    music_enabled = true
endClass
...
println Settings.gravity
Settings.music_enabled = false
...
class GameState [singleton];
move_history.add( GameState )
GameState = GameState()
```

Notes

- The [singleton] attribute allows the singleton pattern to be automatically applied to and used with any class
- Singletons are created on first access rather than at program launch, sidestepping order-of-initialization issues often found with global properties and methods
- [singleton] classes can still be instantiated as standard objects
- Singletons can be reassigned to reset state or switch context

Compounds

Point.rogue

```
class Point( x:Int32, y:Int32 ) [compound]
  METHODS
    method operator+( other:Point )->Point
      return Point( x+other.x, y+other.y )

    method to->String
      return "($,$)" (x,y)
endClass

local a = Point(3,4)
println a + Point(2,3)  # (5,7)
local pt = null : Point  # ERROR
local pt = null : Point?  # OK
```

Notes

- Compound instances are pass-by-value rather than pass-by-reference
- Must use property parameter syntax to define base constructor
- Additional constructors must be via global create() methods rather than instance init() methods
- Cannot extend; not polymorphic
- Additional PROPERTIES okay
- Cannot modify compound properties within methods - must create new
- Cannot assign "null" to compound var - use optional value
- Ideal for manip. many small structs

DEFINITIONS AND ENUMERATE

Defs.rogue

```
class Defs
  DEFINITIONS
    TWO_PI = (2*pi)
    PRINT_PI = println(pi)

endClass

println Defs.TWO_PI  # 6.28
Defs.PRINT_PI  # 3.14
```

Enums.rogue

```
class Enums
  ENUMERATE
    ALPHA
    BETA
    GAMMA = 5
    DELTA
endClass

println Enums.ALPHA  # 0
println Enums.BETA  # 1
println Enums.GAMMA  # 5
println Enums.DELTA  # 6
```

Inheritance

Pet.rogue

```
class Pet( name:String ) [abstract]
  METHODS
    method speak->String [abstract]
endClass

class Cat : Pet
  METHODS
    method speak->String
      return "$ meows" (name)
endClass

Cat( "Fluffy" ).speak  # Fluffy meows
```

Notes

- Single inheritance
- Initializers are inherited
- Any extended initializers hide all inherited initializers (still accessible via 'prior' superclass call)

LoudCat.rogue

```
class LoudCat : Cat
  METHODS
    method speak->String
      return prior.speak + " loudly!"
endClass
```

Global create() Methods

PetV2.rogue

```
class Pet( name:String )
  GLOBAL METHODS
    method create( type:String,
        name:String )->Pet
      which (type)
        case "cat": return Cat(name)
    ...
endClass

println Pet("cat","Frisky").speak
# Frisky meows
```

Notes

- Global create() methods are essentially factory methods
- create() methods and init() methods (initializers) are two forms of constructors

Inline Class Extension

ClassInstance.rogue

```
class Pet( name:String )
  METHODS
    method to->String
      return "A $ named $" (type,name)

    method type->String
      return "pet"
endClass

local cat = Pet( "Fluffy" ) instance
  METHODS
    method type->String
      return "cat"
endInstance
```

ClassInstance.rogue (continued)

```
println cat  # A cat named Fluffy
```

Notes

- AKA *anonymous classes*
- General syntax:
  ExistingClass(...) instance
    # New PROPERTIES & METHODS
  endInstance
- Extends a class and creates a single instance of it

Flag Args, Named Args, Default Args

Flag Args

- Allows true or false to be passed for named parameters in any order
- method m(a:Logical,b:Logical,c:Logical) can be called in these equivalent ways:
- m(true, true, false)
- m(&a, &b, &!c)
- m(&!c, &a, &b)
- m(&b, &!c, true)
- Same syntax can be used to define Logical parameters (order does matter):
- method m(&a, &b, &c)
- Inspired by URL args (?a=x&b=y&c=z)

Named Args

- Extension of same flag arg system can be used for named args of any type
- method m(n:Int32, st:String) can be called in any of the following ways:
- m(5, "Rogue")
- m(&n=5, &st="Rogue")
- m(&st="Rogue", &n=5)

Default Args

- method show(setting=true:Logical)
- show(true) == show()
- Equivalent to show(&setting=true)

Templates

Pair.rogue

```
class Pair<<$FirstType,$SecondType>>
  PROPERTIES
    first : $FirstType
    second : $SecondType

  METHODS
    method init( first, second )
    method to->String
       return "Pair($,$)" (first,second)
endClass

println Pair<<Int32,String>>(1,"one")
# prints: Pair(1,one)
```

Simplifying Template Names

```
class NumPair : Pair<<Int32,String>>;
```

or

```
$define NumPair Pair<<Int32,String>>
...
println NumPair(1,"one")  # Pair(1,one)
```

Template With Arbitrary Tokens

```
println eval<<+>>(3,5)  # 8
println eval<<*>>(3,5)  # 15

routine eval<<$op>>(a:Int32,b:Int32)->Int32
  return a $op b
endRoutine
```

Functions

Function Type

Function(*Type*,...)->(*ReturnType*)

Function

function(*param*:*Type*,...)->(*ReturnType*)
  [with(*p*,*p*:*Type*,*p*=*value*:*Type*)]
  statements
endFunction

- AKA *lambdas*
- 'with' is capture list, can also create new persistent properties
- Explicit 'return' required if return type specified

Generic Function

(*param*,...) [with(...)] => *expression*

- No param or return types specified
- Only single expression allowed
- Expression is implicitly return value
- Must be defined inline during call

Callback Function

object=>*method_name*

- Returns function that calls selected method on given object when invoked

Functions

FunctionTest.rogue

```
FnTest()

class FnTest
  METHODS
    method init
      local sq = function(n:Int32)->(Int32)
        return n^2
      endFunction

      test( sq )
      test( (n)=>n^3 )
      test( this=>negate )
```

FunctionTest.rogue (continued)

```
    method test( fn:Function(Int32)->(Int32) )
      println "f(2) = $" (fn(2))

    method negate( n:Int32 )->Int32
      return -n
endClass
```

Output

```
f(2) = 4
f(2) = 8
f(2) = -2
```

Lists

ListTest.rogue

```
local nums = Int32[]

local integers = [3,4,5]
println integers  # [3,4,5]

local reals = Real64[][3,4,5]
println reals  # [3.0,4.0,5.0]

println integers.count  # 3
forEach (n in integers)
  println n  # 3 | 4 | 5
endForEach
```

ListTest.rogue

```
println (forEach of integers) # 0 | 1 | 2

integers.add( 6 )
println integers # [3,4,5,6]
println integers[1]  # 4
println integers.first  # 3
println integers.last  # 6
println integers.remove_first # 3
integers.sort( (a,b)=>(a>b) )
println integers # [6,5,4]

integers.clear
```

Tables

TableTest.rogue

```
local nums = Table<<String,Int32>>()
nums[ "one" ] = 1
nums[ "two" ] = 2
println nums  # {one:1,two:2}
println nums.contains("one")  # true
println nums["one"]  # 1
println nums//one  # 1
println (forEach in nums) # 1 | 2
println nums.keys  # [one,two]
local entry = nums.find( "two" )
if (entry)
  println "$->$" (entry.key,entry.value)
endIf  # above prints: two->2
```

Notes

- AKA *dictionaries/maps*
- No convenience syntax for creating tables - standard template syntax
- nums//x is shorthand for nums["x"]
- // is "string access" operator; styled after web page breadcrumb trail

Operator Overloads

Operator Overload Syntax

```
class TypeX
  GLOBAL METHODS
    method operator+(a:TypeX,b:TypeX)
        ->TypeX
      if (a is null) return b
      return a.operator+(b)

  METHODS
    method operator+( other:TypeX )
        ->TypeX
      return TypeX( ... )
endClass
```

Notes

- Can implement operator method, global method, or both
- Global method prioritized but not polymorphic, good for LHS null check
- All standard ops except logical ops (and,or,not,xor) & bit shifts supported
- If you implement one of the following sets then Rogue derives remaining relational ops: [<>] [==,<] [==,>]
- Can override logicalize with global method operator?()
- Cannot define new operators

Conversion Methods

to->Type Conversion Methods

- method to->Type shorthand for to_Type->Type
- Called for conversion operator obj->Type
- Ex: to->String == to_String->String
- Parameters allowed:
  method to->String(&hex, &binary)
  ...
  println 127->String(&hex)

Notes

- If existing a:Alpha, then conversion a->Beta uses Alpha.to->Beta if it exists or else Beta.init(Alpha)
- Example:

```
class Percent(n:Int32)
  METHODS
    method to->String
      return "$%" (n)
endClass

println 110->Percent
println 110->Percent->String # equiv
```

Language API - forEach

forEach API

forEach (value in obj) can be used on any object that implements either of two possible reader mechanisms:

Random Access Reader API

- An Int32 *count* method or property
- get(Int32)->X or at(Int32)->X
- *at()* has precedence but usually *get()*

Sequential Access Reader API

- A Logical *has_another* method/prop.
- A *read()->X* method

Numbers.rogue

```
forEach (n in Numbers(1,5)) println n
# 1 | 2 | 3 | 4 | 5

class Numbers( cur:Int32, last:Int32 )
  METHODS
    method has_another->Logical
      return (cur <= last)

    method read->Int32
      ++cur
      return (cur - 1)
endClass
```

Language API - on_cleanup()

on_cleanup()

- If an object defines the method *on_cleanup()* then that method will automatically be called when the object is ready to be garbage-collected
- Once called, the object will not be collected until the *next* GC assuming the object is still unreferenced
- *on_cleanup()* is only ever automatically called once on an object even if it reinserts itself into the program's data structures for a time

Log.rogue

```
class Log
  PROPERTIES
    writer : FileWriter

  METHODS
    method init( filepath:String )
      writer = File( filepath ).writer

    method on_cleanup
      writer.close
    ...
endClass
```

Language API - call()

Call API

- Any object that defines the method *call(params)* can be called using parens and compatible args

Fibonacci.rogue

```
local fib = Fibonacci()
println fib()  # 0
println fib()  # 1
println fib()  # 1
println fib()  # 2
println fib()  # 3
println fib()  # 5
```

Fibonacci.rogue (continued)

```
class Fibonacci
  PROPERTIES
    prev=1, cur=0 : Int64

  METHODS
    method call->Int64
      local next = prev + cur
      prev = cur
      cur = next
      return prev
endClass
```

Task System

FibonacciV2.rogue

```
routine fibonacci->Int64 [task]
  local prev=1, cur=0 : Int64
  loop
    yield cur
    local next = prev + cur
    prev = cur
    cur = next
  endLoop
endRoutine

local fib = fibonacci()
loop (10) println fib()
```

Notes

- A method or routine with the [task] attribute can execute asynchronously
- The initial call returns a Task object capable of being executed
- Call the Task object to start or continue execution; each call returns on a 'yield <value>' or 'return <value>'
- A task can yield any number of values
- A task object is also instanceOf Function() with the appropriate return type
- A task method has full access to context object properties

Task System

IndependentTasks.rogue

```
nums(100).start
nums(200).start
println "Tasks started"

routine nums( n:Int32 ) [task]
  loop (3)
    println n
    ++n
    yield
  endLoop
endRoutine
```

Output

```
100
200
Tasks started
101
201
102
202
```

Notes

- Call start() on a task object to begin its execution as an independent task
- 'yield' is necessary to allow cooperative concurrency

Task System

Await.rogue

```
handle_download.start
println "Download queued"

routine handle_download [task]
  println "Starting download"
  local result = await async_download
  println "Finished: " + result
endRoutine

routine async_download->String [task]
  local timer = Timer(1.0)
  while (not timer.is_expired) yield
  return "data"
endRoutine
```

Output

```
Starting download
Download queued
Finished: data
```

Notes

- One task can 'await' another
- This blocks the waiting task until the awaited task yields/returns a value
- If the awaited task yields or returns a value then it is available as a result of 'await'

Value System

Description

- The Value System is a way to manage JSON-style data
- Collection of classes with compiler integration
- JS-style syntax
- Dynamically typed feel
- Base class Value with wrapper types NullValue, Real64Value, etc.
- Core classes ValueList & ValueTable
- @[] creates ValueList or can be used as shorthand for type ValueList
- @{} shorthand for ValueTable

Value

- Value(x) turns any primitive or object into a Value
- Value->Int32 etc. turns values back into primitives and strings
- Can check type with Value.is_int32, Value.is_list, etc.
- Can turn Value into JSON with Value.to_json(&formatted)->String
- Can turn JSON into Value with JSON.parse(String)->Value

Value System

ValueTable.rogue

```
local person : Value
person = @{ name:
  {first:"Abe",last:"Pralle"}, zip:98004 }
println person//name//first  # Abe
println person//name.count # 2
println person//town? # false
println person//town.count  # 0
person//town = "Bellevue"
println person//zip->Int32  # 98004
println person//zip->String # 98004
```

ValueList.rogue

```
local x = 3
local list = @[ 1, "two", {x:x,y:x+1}, true ]
println list  # [1,two,{x:3,y:4},true]
println list.first  # 1
println list.last  # true
println list.count  # 4
println list[2]//x  # 3
println list.add( list.remove_at(1) )
  # [1,{x:3,y:4},true,"two"]
list.clear
```

Aspects

StringGen.rogue

```
class StringGen [aspect]
  METHODS
    method string->String [abstract]
    method strings(n:Int32)->String[]
      local list=Int32[]( n )
      loop(n) list.add( string )
      return list
endClass

class D6 : Die, StringGen
  METHODS
    method init: prior.init(1,6)
    method string->String: return ""+roll
endClass
```

StringGen.rogue (cont'd)

```
local obj = D6()
println (obj instanceOf D6)  # true
println (obj instanceOf StringGen) # true
println ((obj as D6).strings(4)) # [6,3,4,1]
```

Notes

- Aspects are like Java interfaces with properties and default method defs
- Aspects are *incorporated* into a class definition via base type list
- Approximates multiple inheritance

Aspects and Inheritance

AspectInheritance.rogue

```
class TestAspect [aspect]
  METHODS
    method x: println "AX"; prior.x
    method y: println "AY"; prior.y
endClass

class TestBase
  METHODS
    method x: println "BX"
    method y: println "BY"
endClass
```

AspectInheritance.rogue (cont'd)

```
class TestClass : TestBase, TestAspect
  METHODS
    method y: println "CY"; prior.y
endClass

TestClass().x  # AX | BX
TestClass().y  # CY | BY
```

Notes

- Aspect methods are default defs of methods not defined by incorp. class
- Aspect methods are not necessarily part of the inheritance chain

Augments

AugmentTest.rogue

```
class AugmentTest
  METHODS
    method greek
      println "Beta"

    method color
      println "Orange"
      <<other_colors>>
      println "Green"
endClass
```

AugmentTest.rogue

```
augment AugmentTest
  METHODS
    method greek
      println "Alpha"
    method color
      <<insert>>
      println "Red"
      <<append>>
      println "Blue"
      println "Purple"
      <<other_colors>>
      println "Yellow"
endAugment
```

AugmentTest.rogue

```
local a = AugmentTest()
a.greek
# Alpha
# Beta

a.color
# Red
# Orange
# Yellow
# Green
# Blue
# Purple
```

Augments

General Syntax

augment *TypeName* [: *BaseTypes*]
  # New or mix-in properties and
  # methods
endAugment

Notes

- Can augment regular classes, class templates ("Table"), and specialized templates ("Table<<Int32,String>>")
- Can define base type or multiple aspects to incorporate

Notes

- New augment properties and methods are added to target class
- Existing augment properties and methods are merged in
- Can use <<labels>> to define insertion points in class & corresponding code in augment
- Built-in labels <<insert>> (the default) and <<append>>

Conditional Compilation

Compile Targets

roguec --target=C++,XYZ

- C++ is default target
- Can specify multiple targets
- Currently C++ should always be specified w/any other targets

Single Line $if

$if ("XYZ") *code*
$elseIf ("ABC" or "DEF") *code*
$else *code*

Multi-line $if

$if ("XYZ")
  code
$elseIf ("ABC" or "DEF")
  code
$else
  compileError "X is unsupported"
$endIf

Notes

- Code only parsed for *true* conditions
- Use *compileError* to alert dev when no targets supported

Compiling Rogue-generated C++

Linking

- Compiling XYZ.rogue produces XYZ.h and XYZ.cpp
- roguec with --main or --execute to generate main() method or use custom approach
- #include "XYZ.h"
- Compile and link with XYZ.cpp
- roguec with --gc=manual will check for GC after launch and every time Rogue_update_tasks() is called

Default main()

```
try
{
  Rogue_configure( argc, argv );
  Rogue_launch();
  while (Rogue_update_tasks()) {}
  Rogue_quit();
}
catch (RogueException* err)
{
  printf( "Uncaught exception\n" );
  RogueException__display( err );
}
return 0;
```

Calling Rogue From C++

CallInto.rogue

```
class CallInto [api]
  GLOBAL METHODS
    method hello( n:Int32 )
      loop (n) println "Hello World!"
endClass
```

Main.cpp

```
#include "CallInto.h"
...
Rogue_configure(argc,argv);
Rogue_launch();
RogueCallInto__hello__Int32(5);
```

Notes

- Define a GLOBAL METHOD
- Make class and method [essential] so they aren't culled out when the compiler thinks they're unused
- Or make class [api]
- A global method CName.m(Int32,Real64)->Logical implicitly has the following C++ signature:
  RogueLogical RogueCName__m__Int32_Real64 (RogueInt32 p0, RogueReal64 p1)

Calling C++ From Rogue

StdIO.rogue

```
class StdIO
  # Wraps <stdio.h> class
  DEPENDENCIES
    nativeHeader #include <stdio.h>

  GLOBAL METHODS
    method FOPEN_MAX->Int32
      return native("FOPEN_MAX")->Int32
endClass

println StdIO.FOPEN_MAX
# 20 [system-dependent]
```

Notes

- nativeHeader/endNativeHeader defines code to inject into .h
- native("*<C++>*")->*ReturnType* for inline C++
- Only literal strings can be passed to native(...)
- Use $<var> or $this to insert correct code for local, property, or context

Calling C++ From Rogue

PassRogueParam.rogue

```
class ABC
  GLOBAL METHODS
    method xyz( value:Int32 )
      native @|xyz($value);

      nativeHeader void xyz( int n );
      nativeCode
        void xyz( int n)
          { printf( "XYZ %d!\n", n ); }
      endNativeCode
endClass

ABC.xyz(3)  # XYZ 3!
```

Notes

- nativeCode/endNativeCode defines code to inject into .cpp
- Use $*varname* or $this to insert correct code for parameter, local, property, or context
- nativeCode and nativeHeader can be in different places:
  - In a method - will only be injected if method is ever called
  - In DEPENDENCIES section - will be injected if class is ever used
  - At global scope - will always be injected

Native Properties and Cleanup

Vector.rogue

```
class Vector
  DEPENDENCIES
    nativeHeader #include <vector>

  PROPERTIES
    native "std::vector<int>* list;"

  METHODS
    method init
      native @|$this->list = new
              |std::vector<int>();

    method on_cleanup
      native @|delete $this->list;
```

Vector.rogue (continued)

```
    method add( n:Int32 )
      native @|$this->list->push_back($n);

    method count->Int32
      return native( "(RogueInt32)
        $this->list->size()" )->Int32
endClass
```

Notes

- Constructors and destructors not called on direct (non-pointer) C++ objects
- Consequently native properties should be pointers, primitives, or simple structs

Validation Operators

assert

- assert *<expr>*
- assert *<expr>* || "*Error message*"
- Can be used as expression
- Throws Error if <expr> is false
- return (assert result || "Null result")
- Enabled if compiled with --debug

require

- require <expr>
- require <expr> || "Error message"
- Works like assert but always enabled

ensure

- ensure *<expr>*
- ensure *<expr>*(*args*)
- If <expr> is null, instantiates new object of expression type with optional args
- Equivalent:

```
local list : Int32[]
...
if (list is null) list = Int32[](5)
collect_nums( list )
...
collect_nums( ensure list(5) )
```

Miscellaneous

Meta Info

$methodSignature	Ex: fn(Arg,Arg)
$souceFilepath	".../File.rogue"
$sourceLine	Ex: 65
<expr> isReference	*true* if ref type

Directives

$essential Type	No culling
$essential Type.method	of named

noAction

- A statement that does nothing (no-op)

swapValues a,b

- Generates code equivalent to:
  local temp = a; a = b; b = temp

Modules

module ABC	Id -> ABC::id
using ABC	Allow 'xyz' vs ABC::xyz

Unit Tests

unitTest <expr>	unitTest assert ...
unitTest/endUnitTest	Multi-line

Introspection (Reflection)

type_name and type_info

```
local hg = Hourglass()
println hg.type_name # Hourglass
local info = hg.type_info : TypeInfo
println (info is @Hourglass) # true
println info.properties
# [top:Int32,bottom:Int32]
local p = info.properties[0]
println p.name  # top
println p.type.name  # Int32
println hg.get_property("top")  # 5
hg.set_property("bottom",3)
println hg  # 5/3
```

TypeInfo.create_object()->Object

```
local type = @Hourglass
local obj = type.create_object()
println obj  # 5/0
# ref is type Object, object is type Hourglass

hg = type.create_object() as Hourglass
# OR
hg = type.create_object<<Hourglass>>()
```

Notes

- Currently no support for method invocation via introspection

Native Python Extensions

RogueLib.rogue

```
class Hourglass( top=5:Int32 ) [api]
  PROPERTIES
    bottom : Int32

  METHODS
    method tick
      if (not times_up) --top; ++bottom

    method times_up->Logical
      return (top == 0)

    method to->String
      return "$/$" (top,bottom)

    method turn_over
      swapValues top,bottom
endClass
```

setup.py

```
# python setup.py build_ext --inplace
from distutils.core import setup,Extension
from Cython.Build import cythonize

setup(
  ext_modules = cythonize(
    Extension(
      "roguelib",

    sources=["roguelib.pyx","roguelib_module.cpp"],
       language="c++"
    )
  )
)
```

Native Python Extensions

Build roguelib

```
> roguec RogueLib.rogue --target=Cython
  Writing roguelib_module.h...
  Writing roguelib_module.cpp...
  Writing roguelib.pyx...

> pip install cython

> python setup.py build_ext --inplace
  (Produces roguelib.so and roguelib.cpp)
```

python

```
>>> import roguelib
>>> hg = roguelib.Hourglass()
>>> print hg
5/0
>>> print hg.top, hg.botom
5 0
>>> hg.tick()
>>> print hg, hg.times_up()
4/1 False
>>> hg.turn_over()
>>> print hg, hg.times_up()
1/4 False
>>> hg.tick()
>>> print hg, hg.times_up()
0/5 True
```

The End