Jinx Tutorial

Introduction

Jinx is a lightweight embeddable scripting language, intended to be compiled and used from within a host application. The library is written in modern C++, and the API is designed to be simple and easy to use. The language syntax is highly readable, looking like a cross between pseudo-code and natural language phrases.

Why Yet Another Scripting Language?

Lua is a highly successfully embeddable scripting language and was a major inspiration for Jinx, along with others like C/C++, Python, AppleScript and more. As successful as Lua has been, there are a few ways in which it is less than ideal. The language has a number of syntax quirks and can be challenging for non-programmers to use. Lua's original design reflects its intended use as a *data description language*, and as such, many language features are aligned for that sort of use. And while the C-based API is powerful, it can be obtuse and tricky to use.

In contrast, Jinx is specifically designed for the sort of highly *procedural, asynchronous* scripting that is often required of real-time applications like videogames. The language itself is simple and highly readable, and has a small but robust set of features. Variables are dynamically and strongly typed, and values are easily converted between types by explicit casts. Properties and functions are organized by libraries that also act as namespaces, and various levels of visibility can be assigned to these elements, rather than sharing everything in a global namespace by default.

Jinx is written in modern C++. An important design goal was to make both the language *and* the API safe and simple to use. Special attention has also been paid to memory management. Jinx has a built-in block allocator that reduces the number of small, individual allocations typically required by a scripting language. Methods to replace the underlying allocations and error logging functions are also available.

Jinx also has a fundamentally different approach to script execution. Every script has its own private execution stack by default, meaning each script runs as a co-routine. This makes it incredibly simple to write a script that executes over a specified period of time, an important feature for real-time applications. Naturally, you can use Lua this way too, but it requires a significant amount of non-trivial boiler-plate code.

Jinx Features

- Written in modern C++.
- Robust, easy-to-use API for interaction between scripts and native code.
- Available as traditional library or header-only version.
- Intuitive syntax that looks very similar to pseudocode.
- Language and API features specifically designed for real-time applications.
- Every script executes asynchronously as a co-routine.
- Scripts and strings are fully UTF-8 compatible.
- Customizable memory allocator callbacks.
- Built-in block allocator for improved small object allocation performance.

- Ref-counted memory management.
- Load and compile on the fly, pre-compile bytecode, or use a mixture of the two.
- Generated bytecode is 32/64-bit platform independent.
- Built-in profiling and memory use APIs.

Getting Started With Jinx

Here we list everything you need to get the Jinx library compiling and integrated into your own project.

Jinx Prerequisites

Jinx is written in C++ 17, and as such, requires a compiler that conforms to this ANSI standard. The library compiles cleanly with Visual Studio 2017 on Windows, the latest version of Xcode on Mac using Clang 6.0, and with g++ 8.0 or later on Linux.

In this tutorial, we'll also assume the reader is familiar with C++, so Jinx syntax will occasionally be explained in terms of functionality relative to C++, rather than an emphasis on teaching someone how to program.

Compiling Jinx

Jinx supports the CMake build system. If you generate Jinx as a standalone project, CMake will also generate all related tests and tools. If you include CMakeLists.txt from another project, Jinx will only generate a library.

If you wish to compile Jinx using a different build system, it should be reasonably straightforward to do so, as all required files are contained in the `Source/` folder.

Alternatively, Jinx is also available as a single-header-file amalgamation. You can find this file at `Include/Jinx.hpp`.

Either version provides identical functionality, so feel free to use whichever style you prefer.

Running Your First Script

Let's go over everything you need to do to compile and execute your first script. For now, we'll keep things simple and avoid complicated error handling, asynchronous behavior, customization, and so on. To start with, you'll create a runtime object using the `CreateRuntime()` function. Each runtime object represents a distinct execution

environment. Next, you'll compile and execute the script using the runtime's `ExecuteScript()` member function.

```
#include "Jinx.h"

using namespace Jinx;

// Create the Jinx runtime object
auto runtime = CreateRuntime();

// Text containing our Jinx script
const char * scriptText =
u8R"(

-- Use the core library
import core

-- Write to the debug output
write line "Hello, world!"

)";

// Create and execute a script object
auto script = runtime->ExecuteScript(scriptText);
```

Congratulations! You've compiled, created, and executed your first Jinx script. `Runtime::ExecuteScript()` is a convenience function that compiles a text to bytecode, creates a script, and executes that script in a single call. There are functions to perform each of these steps separately, of course, which you would likely use in more complex scenarios. We'll look into the nuances of the native API later, but this should at least get you far enough to start experimenting with the language itself a bit.

Hello, World!

Let's examine the classic "Hello, world!" program written in Jinx, which will help demonstrate some basic language features:

```
-- Use the core library
import core

-- Write to the default output
write line "Hello, world!"
```

The first thing you may notice is that Jinx supports single line comments using a pair of dashes. The next line demonstrates how to use code modules or packages, which are known in Jinx as *libraries*. There is only one built-in library called `core`, which we're using here. Using Jinx, it's simple to create and use your own libraries as well. The `import`

keyword acts similar to both an `#include <filename>` and a `using namespace` in C++. We'll examine libraries in more detail later.

Finally, we use the `core` library's `write line` function to output the string `"Hello, world!"` to the debug output function. By default, this generates console output using the standard C `printf()` function, but an application can intercept this and route the output to its own logging functions.

The Jinx Language

Now that you've gotten the Jinx library compiled, and created and executed a simple script, let's take a quick tour of the Jinx language itself.

Statements and Whitespace

You may have noticed in our earlier example that there are no statement termination symbols, such as the semicolon in C/C++. In Jinx, the end of each line typically marks the termination of a statement. There are two exceptions to this rule, which we'll demonstrate later in more detail:

- You may add an ellipse (...) to the end of a line and continue the statement on the next line.
- You may break up initialization lists into muliple lines.

Combining multiple statements onto a single line is not allowed. No other whitespace is significant, except for its role in separating tokens.

Case Insensitivity

Jinx is a case insensitive language. This compliments the general design principles of acting like a natural language. An identifier `foobar` is equivalent to `Foobar` or `FOOBAR`. The runtime adheres to standardized language-independent Unicode full folding rules using a mapping table, converting uppercase values into lowercase values prior to token comparison. The relatively rare instances of case folding that requires language-specific lexical analysis are not implemented.

Comments

Jinx supports both single-line and block style comments. Single line comments begin with two dashes. Anything following those dashes on the same line is a comment.

```
-- This is a single-line comment
```

Block comments use dashes as well. Three consecutive dashes starts a block comment. Three more consecutive dashes also ends the comment block. This form of comment ignores newline markers, continuing until it sees a terminating symbol. We see here several forms that block comments can take.

```
---Block comment---

--- Block
comment ---

---
Block
comment
---
```

One interesting aspect of block comments is that three dashes *minimum* are required to begin and end them – more dashes on both ends are perfectly legal. This makes it possible to create block comments like this:

```
-------------
Block
comment
-------------
```

Variables, Types, and Assignment

Variables in Jinx are dynamically typed, meaning they can be easily easily converted between compatible types, either explicitly via casting, or implicitly with function parameter coersion. We'll see examples of both of these later. Unlike with some dynamically typed languages, variables can only be declared as part of an assignment operation. This helps to prevent errors due to a simple mistyping of a variable or identifier name.

Common Variable Types

There are six commonly-used types used in the Jinx language. We see here a few examples showing how to assign values to some variables. The comments list the value type each variable represents as the assignment is made. Variable assignment follows the form: `set {variable} to {expression}`. Here's what that looks like using a few built-in types.

```
set var1 to null                    -- null type
set var2 to 123.456                 -- number type
set var3 to 42                      -- integer type
set var4 to false                   -- boolean type
set var5 to "a string value"        -- string type
set var6 to "red", "green", "blue" -- collection type
```

Aside from a few special exceptions (loops with index variables and function parameters), all variables are declared and assigned values using this form. If you try to use a variable in

an expression that hasn't been declared yet, it will result in a syntax error when compiling the script.

Null

The `null` type is represented internally by a C++ `nullptr_t`. It's primary purpose is to differentiate itself from every other type, or to serve as an invalid type for the purpose of error checking.

Number

The `number` type is represented internally by a C++ `double`, which on supported platforms is a 64-bit floating-point value. Any numeric constant with a decimal point is assumed to be a `number`.

Integer

The `integer` type is represented internally by a C++ `int64_t` type, a 64-bit signed integer. Any numeric constant without a decimal part is assumed to be an `integer`.

Boolean

The `boolean` type is represented internally by a C++ `bool`. Possible values are `true` and `false`.

String

The `string` type is represented internally by a C++ `std::string`. Any valid UTF-8 codepoints are supported.

Collection

The `collection` type is represented internally by a C++ `std::map`. Several types (numbers, integers, strings, and GUIDs) may be used as a key, but ordering between different types is undefined. This collection type is unlike the other basic types in that the internal representation is a *pointer* to a collection. As such, copying the collection from one variable to another only copies the pointer, not all the elements it contains. The built-in comparison operators also operate on the pointer value, not the collection values.

Less Common Types

In additon to the more common data types, there are a few less commonly used types. Some of these are only available via the native API, or are derived from other script operations.

CollectionItr

A collection iterator is used when looping over a collection, and can be used to access a node's key or value.

UserObject

A ref-counted shared pointer contains a user-defined object, primarily useful for passing custom objects to other user-defined functions.

Buffer

A memory buffer object is useful for storing and retrieving custom data in scripts. Like collections and user objects, this variable stores a *pointer* to the buffer itself, so copying the variable does not actually copy the buffer contents.

Guid

A 128-bit Globally Unique ID can be used as collection keys or used by a host application for a variety of purposes.

ValueType

The `valuetype` type represents the type of a variable. Variables can be queried at runtime for their type by using the `type` keyword after the variable name, and this value allows comparison or other operations to be performed. It's typically used implicitly, not stored in variables, but there is no prohibition against doing so.

Variable Names

Variables must start with a non-numeric and non-symbolic character, and it cannot be the same as a reserved keyword. Aside from those restrictions, symbols can use any valid Unicode character. Unlike most languages, Jinx allows multi word variables. The parser always favors the longest possible variable name match, which you'll see is important when you start using functions. You may also surround a variable name in single quotes in order to specify a multi-word variable more explicitly.

```
-- Both legal variable names
set some variable to 123
set 'some variable' to 123 -- Same result as previous line
set resumé to "my resumé text"
```

Casting and Type

Variables can be explicitly cast to other types. The runtime system will cast between many types without issues, but will log warnings if incompatible types are cast, such as attempting to cast a `collection` into a `integer` value. The `type` keyword is used to retrieve the type of a variable or property at runtime.

```
-- Cast variables between integer and string types
set a to 123
set b to a as string      -- b = "123"
set c to b as integer     -- c = 123
```

```
-- The 'type' keyword retrieves a variable's type
set d to a type            -- d = integer
```

Variable Scope

Jinx, like many languages, has a concept of scope. The keywords `begin` and `end` can be used to explicitly create a scope block. Variables created outside are visible to any inner scope, but the reverse is not true.

```
set a to 32          -- Declare a variable at the outermost scope
begin
    set b to a       -- Legal
end
set c to b           -- Error: b = undefined
```

Other language constructs like the `if` or `loop` keywords automatically create a new scope block. The `end` keyword is typically (but not always) used to mark the end of the scope.

External Variables

Variables can be set in the script by the external program before the script ever executes. We'll learn about how to do this later when we discuss the C++ API. Use the `external` keyword to designate a variable that's been set outside the script itself without assigning a value to it. You can think of it as a placeholder that allows the parser to know about a variable that doesn't yet exist.

```
external var1        -- Reserve a variable
set var2 to var1     -- Assign to a new variable
```

Here we see an example of a variable called `var1` being reserved, and then assigned to a new variable named `var2`. Unlike normal variables, external variables can only be declared at the root level. That is, you can't declare an external variable inside a scope block of any sort.

```
begin
    external var1    -- Error!
end
```

Mathematical Operators

Six basic math operators are supported: addition, subtraction, multiplication, division, modulus, and negation. Mathematical expressions work like in C in that multiplication, division, and modulus operations are executed before addition and subtraction. You can change the operation order by explicitly grouping operations with parentheses.

```
-- Evaluate expressions
set a to 2 + 3 - 1      -- 4
set b to a * 4 / 2      -- 8
```

```
set c to 2 + 2 * 3      -- 8
set d to (2 + 2) * 3    -- 12
```

If any of the operands are numbers, the expression will always result a number as well.

```
set a to 1 + 2.5        -- 3.5
```

If you divide two integers and the result can't be stored in an integer, the result will be a number. If you wish to preserve the result as an integer no matter the result, you must explicitly cast the result of the operation back to an integer. If either value is a number, than the result will be a number.

```
set a to 3 / 2                -- 1.5
set b to (3 / 2) as integer   -- 1
```

Dividing by zero, whether using an integer or number, will result in a runtime error.

Modulus operators find the remainder of a division operation. If both values are integers, the remainder value is also an integer. If either value is a number instead of an integer, then the result will be a number.

```
set a to 5 % 3          -- 2
set b to 5 % 3.5        -- 1.5
```

It should also be noted that the mod operator works similarly to Python instead of C++, in that the resulting sign matches the divisor, rather than the dividend.

```
set a to -5 % 3          -- 1 (in C++, -2)
```

As with the division operator, a mod by zero will result in a runtime error.

Negation of both constants and variables is also permitted. In the case of constants, such as in the following statement, if the negative value prepends the numeric value with no whitespace, both the negative sign and number are parsed together as a negative constant value.

```
set a to -1
```

However, if a minus sign prepends a variable or the result of an expression, then the negation operator is applied. The following code demonstrates various ways the negation operator can be used.

```
set a to 1
set b to -a          -- b = -1
set c to -(1 + 2)    -- c = -3
set d to -a - -a     -- d = 0
```

Comparison Operators

Comparison operations always evaluate to a true or false Boolean value. Comparing two values for equality or not-equality uses the `=` and `!=` operators. In the example below, a value of `false` is assigned to `a`, and `true` is assigned to `b`.

```
set a to 1 = 2     -- false
set b to 1 != 2    -- true
```

Jinx also supports 'less than' or 'greater than' operators, as well as 'less than or equal to' and 'greater than or equal to' operators.

```
set a to 1 < 2    -- true
set b to 1 <= 2   -- true
set c to 1 > 2    -- false
set d to 1 >= 2   -- false
```

Comparison operators have a lower precedence than mathematical operators, and so will be evaluated after other math operations unless otherwise indicated by parentheses.

```
set a to 1 + 5 < 1 + 6        -- true
set a to (1 + 5) < (1 + 6)    -- true (equivalent)
```

For equality (`=`) tests, comparing fundamentally different types will result in a return value of `false`, and for inequality (`!=`) tests, `true`. The exception to this rule is when comparing an `integer` and `number`, in which case the values are logically compared as floating-point values.

For value comparison operators (`<`, `<=`, `>`, `>=`), comparison of invalid or non-matching types will result in a run-time error. Only types `integer`, `number`, `string`, and `guid` can use these operators. Again, direct comparisons between types `integer` and `number` are permitted.

Logic Operators

Logical operators `and` and `or` are supported.

```
set a to true and false   -- false
set b to true or false    -- true
```

These operators have a lower precedence than math or comparison operations, so there is no need to use parentheses if your intention is to evaluate math expressions first. In the following example, both the left and right expressions surrounding `and` are evaluated before the `and` operator is invoked.

```
set a to 1 != 2 and 3 != 4       -- true
```

The `not` operator has an equivalent precedence to the `and` and `or` operators, and so will negate the expression that follows it until it reaches one of these operators. If you wish to negate part of an expression which includes other logic operators, then you can use parentheses to do so.

```
set b to not 1 != 2 and 3 != 4      -- false
set c to (not 1 != 2) and 3 != 4    -- false (equivalent)
set d to not (1 != 2 and 3 != 4)    -- true (not equivalent)
```

Short Circuit Evaluation

As logic operators are evaluated, they are subject to "short-circuit" evaluation, which dictates that the interpreter will bypass the runtime evaluation of the second half of an logic operator if the outcome can be determined via the first half of the expression. Both the `and` and `or` operators are subject to this short circuit evaluation rule.

In the following cases, the second half of the expression will be skipped, because the outcome is already guaranteed:

```
set a to 1 < 2 or 3 = 4        -- 3 = 4 is not evaluated
set b to 2 < 1 and 3 = 3       -- 3 = 3 is not evaluated
```

Increment and Decrement Statements

For numeric variable types, you can increment or decrement variables using the `increment` and `decrement` statements.

```
set a to 4
increment a     -- a = 5
decrement a     -- a = 4
```

You can also increment or decrement by a specified amount using the `by` keyword and a subsequent value or expression.

```
set a to 4
increment a by 3     -- a = 7
decrement a by 4     -- a = 3
```

You cannot use the `increment` or `decrement` keywords in expressions because they don't return a value. They are considered to be statements, not operators.

Incrementing a non-numeric variable or incrementing by a non-numeric variable or value will result in a runtime error.

Conditional Branching

Jinx supports `if` and `else` keywords to perform conditional branching. If the expression following the `if` statement is `true`, the branch will be executed, and not if `false`.

```
import core

-- Simple if branch
if 1 < 2
    write line "1 < 2"
end
```

Notice how the `end` keyword is used to mark the end of the branch block. Unlike C/C++, you must always terminate blocks explicitly. In the case of an `if`/`else` branch, the `else` statement acts as the block terminator.

```
-- If/else
if 1 > 2
    write line "1 > 2"
else
    write line "1 <= 2"
end

-- If/else-if/else
if 1 > 2
    write line "1 > 2"
else if 1 = 2
    write line "1 = 2"
else
    write line "1 < 2"
end

-- Nested if branching
if 1 < 2
    if 3 < 4
        if 5 = 6
            write line "compound check is true"
        else
            write line "compound check isn't true"
        end
    end
end
```

Collections

Collections are keyed associative arrays with arbitrary key-value pairs. When creating a collection without explicitly specifying keys, incrementing integers are used automatically. This allows scripts to simulate arrays using collections.

Integers, numbers, strings, and GUIDs may all be used as key values, and any value that can be stored in a variable may be stored as the associated value component, with the exception of null, as this value is used as a shorthand for removing elements.

Creating an Empty Collection

Creating an empty collection in Jinx is accomplished by using an empty set of index operators.

```
set coll to []
```

Collections and Initialization Lists

Any list of values separated by commas is translated into a collection with sequential keys. This is known as an *initialization list*. When elements are added without specifying a key, new elements start with a key of one and always add elements sequentially.

```
set coll to 5, 4, 3, 2, 1
```

Accessing Collection Elements

Accessing an individual element is done with square brackets, in which the index value is specified. Again, note that when a collection is simulating an array, the indices are one-based, not zero-based.

```
set x to coll [2] -- x = 4
```

Collection Size

The core library contains a size function that returns the number of elements in a collection. Remember that you must import the core library before using this function.

```
set c to coll size -- c = 5
```

Explicit Key-Value Assignment

You can also explicitly assign both keys and values together. When two values are contained in a set of square brackets, it is assumed to be a key-value pair. These can also be contained in an initialization list.

```
-- This...
set keyed coll to [1, "one"], [2, "two"], [3, "three"], [4, "four"], [5, "five"]

-- is equivalent to this...
set unkeyed coll to "one", "two", "three", "four", "five"
```

Just like before, individual values are accessed via the key using the index operator. There is no difference whether keys are assigned implicitly or explicitly.

```
set x to unkeyed coll [3] -- x = “three”
set x to keyed coll [3]   -- x = “three”
```

New values can be assigned or existing values modified using this operator as well.

```
set coll [6] to "six"
```

As mentioned earlier, keys can be other types than integers. You can also use numbers, GUIDs, or strings as keys. There is no prohibition against mixing key types, but the elements in the collection will be sorted by type.

```
set coll to ["apple", 345], [111, "orange"], [12.34, false], [-99, true]
```

Collections in Collections

Collections may also be stored inside collections, as in this example. However, keep in mind that collections cannot be stored as keys. Collections may be nested arbitrarily deep.

```
set coll to ["fruit", ["apple", 345]]
```

Collections stored inside collections may be accessed via multiple index operators.

```
set a to coll ["fruit"]["apple"] -- a = 345
```

If an element does not exist, it can be created in this manner as well. If the keys already exist, the value is modified.

```
set coll ["fruit"]["pear"] to 789
```

Accessing an element that does not exist will not cause a runtime error. Instead, the returned value will be null.

```
set coll to []
set a to coll ["doesn't exist"] -- a = null
```

Erasing Elements in a Collection

The `erase` keyword is used to remove elements from a collection by key.

```
-- Remove element from container by key
erase coll ["apple"]
```

Empty Collection Check

The `is empty` function returns `true` if the collection is empty and `false` if not empty.

```
if coll is empty
    -- do something
```

Syntactic Sugar

We mentioned earlier that, for convenience, it is permitted to break up items in an initialization list into multiple lines, as follows:

```
-- A multi-line initialization list
set unkeyed coll to...
    "one",
    "two",
    "three",
```

```
    "four",
    "five"

-- A multi-line initialization list with key-pair values
set keyed coll to...
    [1, "one"],
    [2, "two"],
    [3, "three"],
    [4, "four"],
    [5, "five"]
```

Breaking up individual elements within a key-value pair onto multiple lines, however, is considered a syntax error.

In these examples, the general-purpose ellipse is required to allow the first item in the initialization list to be placed on a new line. Subsequent items are allowed on a new line because of the comma, which unambiguously indicates that another element in the list is expected.

Loops

Jinx has a number of ways to create loops. Every loop begins with the `loop` keyword, and in most cases is terminated with the `end` keyword.

Counting Loops

Integer-based counting is a common operation, and so has built-in language support. In the simplest case, you can count from a low to a high integer. The starting and ending values are inclusive, meaning the index value must exceed the specified range before the loop exits. Any expression can be used to set the loop parameters, but this expression is evaluated only once as the loop starts. If you require a loop which executes an expression repeatedly, then you must use a *conditional loop*, explained in a later section.

Here's how you count from one to ten.

```
import core

loop from 1 to 10
    write line "looping"
end
```

If you wish to access the index variable, you can specify a variable name to use for this.

```
loop x from 1 to 10
    write line "x = ", x
end
```

Counting down in reverse from ten to one is just as easy. Jinx will compare the two initial starting and ending expressions and automatically count in an appropriate direction.

```
loop x from 10 to 1
    write line "x = ", x
end
```

You can also explicitly assign the value to count by, meaning you can count in either direction and by any value. If you are specifying a "count by" value and counting in reverse, you must be sure to specify a negative number. When using multiples, the loop will continue as long as the index value is less than or equal to the ending value when counting up, or greater than or equal to the ending value when counting down. In the following example, the values of 1, 3, 5, 7, and 9 will be printed over five loops.

```
loop x from 1 to 10 by 2
    write line "x = ", x
end
```

Collection Loops

Looping over a collection uses a slightly different syntax, using the `over` keyword.

```
set 'my list' to 5, 4, 3, 2, 1
loop over 'my list'
    write line "element"
end
```

An optional variable can hold an iterator, which can be used to access either the key or value at each element using core library functions of the same name.

```
loop x over 'my list'
    write line "key = ", x key, ", value = ", x value
end
```

It is safe to remove a collection element in the middle of the loop using the `erase` keyword, as the iterator will implicitly advance to the next element after the current element is erased.

```
loop x over 'my list'
    if x value = 1
        erase x
    end
end
```

Conditional Loops

Jinx supports the `while` keyword to check for arbitrary looping conditions. The loop continues while the expression after the while keyword evaluates to `true`. You may position the statement either at the beginning or end of the loop.

```
-- While loop with condition at front
set x to 1
loop while x < 10
```

```
    increment x by 2
    write line "x = ", x
end

-- While loop with condition at end
set x to 0
loop
    increment x
    write line "x = ", x
while x < 10
```

Jinx also supports the `until` keyword, which reverses the conditional logic, looping while the expression behind evaluates to `false`. Or, describing it differently, the loop continues until the expression evaluates to `true`.

```
-- Until loop with condition at front
set x to 1
loop until x >= 10
    increment x by 2
    write line "x = ", x
end

-- Until loop with condition at end
set x to 0
loop
    increment x
    write line "x = ", x
until x >= 10
```

This provides a small syntactic convenience, allowing the loop to read more naturally depending on the functions you have available. For instance, if you wish to loop until a function returns a true value, you could use this while loop:

```
loop while not condition is met
end
```

However, it's slightly easier to read using:

```
loop until condition is met
end
```

Breaking Out of Loops

You may break out of a loop by using the `break` keyword.

```
-- Use break to exit a loop
loop while true
    break
end
```

Functions

Functions in Jinx are perhaps its most interesting language feature. In Jinx, functions are identified with a list of names and parameters that forms a unique *function signature*. Any reserved keywords may be used as part of the function signature, so long as at least one part of the signature is a non-keyword token. The end result can be remarkably natural-looking prose when calling functions. For instance, a single function call may look like this:

```
wait between 3.5 and 10 seconds
```

Both `wait` and `and` are reserved keywords, which doesn't matter at all, since `between` and `seconds` (non-keywords) are also part of the signature. The parameter values are also very clearly labeled, so there is no ambiguity about what `3.5` or `10` really mean.

Another interesting feature allows you to define alternate name parts that can be used interchangeably. In the case of this particular function, when passing a value of `1`, it sounds more natural like this:

```
wait between 0.1 and 1 second
```

The last word in the function has been changed from `seconds` to `second`. This is perfectly legal, so long as the name was defined with that particular spelling variation as well. We'll examine how we can do that without having to create a new overloaded function a bit later in this section.

What happens if you declare a variable that happens to match one of the function signature name parts? Here we've defined a local variable that does just this.

```
set seconds to 42
wait between 5 and 10 seconds
```

The parser will always give precedence to matching function names first, and always favors the longest possible match. So, the preceding code will compile and execute without issues.

Simple Function Declarations

Let's take a look at a simple function declaration and how to call that function.

```
import core

-- Declares a function
function say hello
    write line "Hello!"
end

-- Calls the fuction
```

```
say hello
```

The function signature first requires a declaration using the `function` keyword. The rest of the tokens on the line make up the name of the function. As described earlier, notice how we aren't limited to a single word. Calling the function is as simple as invoking its name.

Functions are registered with the runtime when the script is first executed, so the function definition must appear in the script before any code calls it. Since the function is registered immediately when the signature is parsed, functional recursion is supported, meaning a function can call itself.

Functions must be declared at the root script level. They may not be declared inside scope blocks, including conditional statements or loops, or within other functions.

Return Values

Part of the fundamental utility of functions is the ability to return values to the original code that made the call. Jinx supports the return keyword, which allows your function to return a value. Here's what this looks like:

```
function meaning of life
    return 42
end

set a to meaning of life
write line "a's value = ", a -- a = 42
```

In Jinx, function return values are always optional. Every function always returns a value, whether or not you specify one. In the case where a return value is unspecified, the null value is used.

An interesting side effect of Jinx's initializer list syntax is that it allows a user to return multiple values from a function.

```
function some values
    return "wolf", "goat", "cabbage"
end

set wolf to some values [1]
set goat to some values [2]
set cabbage to some values [3]
```

In essence, the values are returned as a collection, which you can then access by the index operator.

Function Parameters

Let's look at an example of a function that takes a few parameters and returns a value.

```
function {x} minus {y}
    return x - y
end

set a to 3 minus 2
write line "a's value = ", a -- a = 1
```

Parameters are indicated by a pair of curly brackets, with a variable name between them. In this case, two parameters `x` and `y` are treated as local variables inside the function body. You may notice that it's perfectly legal for a function definition to begin with a parameter name.

As demonstrated earlier in the collections section, multiple variables or values separated by commas are automatically turned into a collection. For this reason, parameters must always be separated by a name part in the signature. The recommendation is to use this name to identify the parameter in some way. Alternatively, you can choose to explicitly support collection parameters, like the `write line` function. This makes it simple enough to support arbitrary numbers of parameters to functions, since each parameter's type can be queried and handled appropriately at runtime.

Compound Expressions as Parameters

When passing arguments to expression, you may use any legal expression as a parameter. Mathematical and logical expressions have a lower precedence than functions, and so are always evaluated before the function call is executed. Here is an example:

```
set a to 3 * 2 minus 2 + 1        -- a = 1
set a to (3 * 2) minus (2 + 1)    -- Equivalent to previous line
```

This works just as well with expressions passed as a middle parameter – that is, a parameter surrounded by function names on either side. Here is another example:

```
function write {x} value
    write line "Value is: " + x
end
write 1 + 1 value     -- OK!
write (1 + 1) value   -- Equivalent to previous line
```

Chained Function Calls

When passing the results of other function calls as parameters, precedence works as you might expect, working from left to right. Let's take a look at an example of this:

```
set a to 5 minus 3 minus 1   -- a = 1
set b to (5 minus 3) minus 1 -- b = 1 (Equivalent to previous line)
set c to 5 minus (3 minus 1) -- c = 3
```

If you have a function that takes no parameters and returns a value, this can also be used to pass a parameter value to another function call.

```
function meaning of life
    return 42
end

function write {x} value
    write line "Value is: " + x
end

write meaning of life value     -- OK!
write (meaning of life) value   -- Equivalent to previous line
```

Casting Parameters to Explicit Types

The variable may be preceded with an optional value type keyword which indicates the value type to which parameters will be implicitly cast. Jinx is a dynamically typed language and will not generate compile-time errors due to type-related issues, but will instead generate runtime errors if invalid casts occur. For instance, a collection can't be cast to an integer without generating an error.

Let's look at our previous example function and see how we can ensure that any parameters are automatically converted to numbers if possible before we subtract them.

```
function {number x} minus {number y}
    return x - y
end

set a to "3" minus "2"   -- a = 1
```

Alternative Name Parts

Any names in the function signature separated by a forward slash are considered legal alternatives, and can be substituted by the script calling the function.

```
function log value/values/collection {param}
    if param type = collection
        loop p over param
            log value p
        end
    else
        write "log: value = ", param
        write newline
    end
end

-- These are all legal calls to the same function
log value 321.123
log values 1, 2, 3, 4, 5
log collection ["a", "Alpha"], ["b", "Beta"], ["c", "Gamma"]
```

Optional Name Parts

Function signatures can contain optional name parts. In the example below, one of the three name parts is optional, designated by parentheses. Only a single word or a group of alternative words can be designated as optional per set of parentheses. Multiple words may be designated as optional, but they are independently optional from each other. In each function definition, at least one name part must be non-optional between any parameters, or in the function definition as a whole.

In this example, we demonstrate a function with one optional name part.

```
function something (is) enabled
    return true
end

-- These are both legal calls to the same function
set var to something enabled
if something is enabled
    -- This code will execute
end
```

In code, creating optional parts can help for code to read more naturally. For instance, the name without the optional component is useful in an assignment statement. However, when used in an `if` statement, we can then use the optional name part.

As indicated, it's also possible to combine the use of alternative name parts and optional name parts in the same function signature, as follows:

```
function (this/those) (is/are) enabled
    return true
end

set a to enabled -- a is set to true

if this is enabled
    -- This code will execute
end

if those are enabled
    -- This code will execute
end
```

Function Argument Passing Styles

As you can see, Jinx functions are quite flexible in format. Let's examine a few strategies for those situations in which you may have a function with many arguments to pass. Remember that Jinx requires a descriptive word in-between each parameter you pass to a function. Let's first show an example using alternating names and arguments.

```
function somefunc width {w} height {h} description {d}
    -- Do something useful here
end

somefunc width 3 height 15 description "a string"
```

One option available, if you simply wish to give a description for each parameter, is to decorate each descriptive name part with a colon at the end. Because the colon is not a reserved operator in Jinx, it simply becomes part of the name. It would then look like this:

```
function somefunc width: {w} height: {h} description: {d}
    -- Do something useful here
end

somefunc width: 3 height: 15 description: "a string"
```

For modest numbers of parameters, this seems like an excellent way to describe each parameter. Its value becomes more apparent if you're passing variables instead of literals.

```
set w to 3
set h to 15
set d to "a string"

somefunc width: w height: h description: d
```

The addition of the colon helps to indicate the purpose of each parameter quite clearly.

How about if you'd just like to pass a list of comma-separated values, like you can do in many languages? In fact, you can simulate C-style functions with a simple list of arguments, if this is a style you prefer:

```
somefunc(3, 15, "a string")
```

The parentheses are actually optional. So, this works just as well:

```
somefunc 3, 15, "a string"
```

How does this work? What's actually happening here is that Jinx is creating and passing a collection with automatically assigned index values. We've previously referred to this as an *initialization list*, which is created any time Jinx sees a simple comma-separated list. So, your function declaration would look something like this:

```
function somefunc {collection c}
    set width to c[1]
    set height to c[2]
    set description to c[3]
    -- Do something useful here
end
```

You can also choose to create a collection more explicitly, assigning both keys and values in pairs. If we're doing this, we may wish to name the parameters using strings instead of simply using index values.

```
function somefunc {collection c}
    set width to c["width"]
    set height to c["height"]
    set description to c["description"]
    -- Do something useful here
end

set params to []
set params["width"] to 3
set params["height"] to 15
set params["description"] to "a string"

somefunc params
```

An alternate method to call this function might be to create the collection inline, like this:

```
somefunc ["width", 3], ["height", 15], ["description", "a string"]
```

If you are concerned about stuffing a large number of parameters names and values onto a single line, keep in mind that Jinx allows you to break up initialization lists automatically, or you can use the ellipse (`...`) operator to signal an explicit line break. As such, your function call could look like this:

```
 somefunc ...
    ["width", 3],
    ["height", 15],
    ["description", "a string"]
```

For parameter validation inside the function body, you'll have to check the assigned variables against `null`, and take appropriate action, which may be logging an error or supplying a default value. It's perfectly safe to first attempt to assign the parameters to variables and then check those variables, because attempting to access a non-existing key-value pair in a collection just returns `null`. Here's what this might look like:

```
function somefunc {collection c}
    set width to c["width"]
    set height to c["height"]
    set description to c["description"]
    if width = null or height = null or description = null
        -- Handle error
    end
    -- Do something useful here
end
```

While passing collection of name-value pairs is a bit more verbose, it does have the virtue of allowing the caller to dispense with any required ordering, and even permits completely optional parameters, assigning default values if the specified key doesn't exist in the collection.

As you can see, you have a fair amount of flexibility in the style you choose for your function calls.

Parsing Ambiguity

Due to the flexible rules which allow multi-part function names with overloaded variations and the use of reserved keywords as name parts, it is possible to accidentally create parsing ambiguities that result in syntax errors when chaining function calls. If this occurs, and using a less ambiguous function name is not an option, the most reliable method of resolving those ambiguities is to use parenthesis.

The Return Statement Outside Functions

While the `return` statement's utility is primarily focused on returning a value from a function, it can also be used to exit from the main body of a script. This can be useful if you wish to check for early out conditions, or if you simply wish to temporarily disable a script by placing a `return` statement before anything else can execute.

```
return

-- This code will never be reached
set x to 123
```

In this example, the code below the `return` statement will not be executed. Any attempt to access the variable `x` will return a `null` type.

In this context, returning a value is meaningless, and if attempted, will generate a warning. This is because you can instead assign a value to any local variable and access that variable by name after the script is finished executing.

Concurrency

Jinx supports concurrency through the use of cooperative multi-tasking. Each script contains its own execution state, and so can be run indefinitely without any interference from other scripts. A script will pause execution and return from the `Execute()` function when it encounters the `wait` keyword.

```
import core

-- Assume " current time" function exists and retrieves time in seconds
set t to current time + 5
```

```
-- Loop for five second
loop while t < current time
    wait -- Exits execution function, continuing when Execute() is called again
end
```

Jinx also provides a syntactic shortcut for combining this empty loop into a single statement, as follows:

```
-- Wait for five second
wait while t < current time
```

You can use the `while` keyword after `wait`, followed by any expression. The script will continue to defer continued execution as long as the expression evaluates `true`.

As with loops, you can invert the logic of the expression by using the `until` keyword in place of `while`.

```
-- Wait for five second
wait until t >= current time
```

Libraries

Multiple scripts can be packaged together into reusable modules called *libraries*.

Jinx makes it easy to extend the language with libraries of functions to perform any sort of high-level task required. These libraries may take the form of Jinx scripts, native code, or a combination of both.

It is the responsibility of the host application to manage dependencies between libraries and any scripts which use those libraries by controlling the order of both script compilation and execution. In order to successfully compile a script that uses a library, the scripts that make up that library must either be compiled or executed before any script that uses it within the same runtime object.

The Import Keyword

You've seen some examples using the `import` keyword already. This tells Jinx to allow a specific library to be used in the script. For example, the following line tells the parser to recognize any functions from the built-in core library.

```
import core
```

Import statements must be the first statements in a script.

The Library Keyword

After any import statements, the `library` keyword is optionally used to declare which library this script belongs to. If this statement isn't found, the library is added to a library identified by an empty string by default. Library names follow the same naming rules as variables. You cannot use reserved keywords as names, and multiple words must be surrounded by single quotes.

```
library 'custom stuff'
```

Library Names as Identifiers

The `import` keyword also acts like the C++ `using namespace` keywords. If there are no name collisions, you can use any library functions without the name of the library preceding it. In the following example, both function calls are equivalent.

```
import core

write line "test"
core write line "test"
```

If a library function or property (a public variable) conflicts with another library name, you will be required to specify the library name to resolve the conflict.

Library Visibility

Libraries have a notion of visibility relative to the current library in which the definition is made. There are three levels of visibility in libraries: *local*, *private*, and *public*.

Local Visibility

Local visibility is implicit unless specified otherwise, and indicates visibility only within the current script.

Private Visibility

Private visibility is declared with the `private` keyword, and indicates visibility within the entire library.

Public Visibility

Public visibility is declared with the `public` keyword, and indicates visibility from any script that imports the library with the `import` statement.

Library Functions

You can declare functions to be public or private. Simply use the `public` or `private` keyword before the function keyword.

```
public function do something public
    write line "do some public task"
```

```
end

private function do something private
    write line "do some private task"
end
```

Library Properties

Public or private variables are called properties, and they have some additional restrictions than normal variables. Properties cannot be declared within any scope blocks or within functions. They must be declared so that they are visible to the entire script. As with functions and normal variables, properties can't be accessed until after they've been declared.

```
set public 'public prop' to "I'm a public property"
set private 'private prop' to "I'm a private property"
```

Read-only Properties

Properties can also be read-only. This is useful for creating library-wide or global constant values. The `readonly` keyword can be used with public or private properties.

```
set public readonly 'public readonly prop' to "I'm a public readonly property"
```

The Jinx API

The Jinx C++ API allows you to integrate the Jinx scripting language into your own application. While an important aspect of this is compiling, executing, and managing Jinx scripts, perhaps an even more critical aspect of an embeddable scripting language is how it shares data with its host application. After all, the entire point of a scripting language is to actually perform some useful task for the host application, and for that, we need a method of sending data to and retrieving data from the scripting system.

Initialization and Shutdown

There are a few parameters in the Jinx library that are considered global, such as memory management, hooking up debugging callbacks, and so forth. These parameters must be initialized before you allocate the Jinx runtime object, from which all other objects are allocated. After all, you need to define how your allocation works before you allocate any objects. This function, however, is optional if you wish to simply use the default values.

Here is a sample of how you might use the various initialization parameters:

```
Jinx::GlobalParams globalParams;
globalParams.enableLogging = true;
globalParams.logBytecode = true;
globalParams.logSymbols = true;
```

```
globalParams.enableDebugInfo = true;
globalParams.logFn = [](LogLevel level, const char * msg) { printf(msg); };
globalParams.allocBlockSize = 1024 * 16;
globalParams.allocFn = [](size_t size) { return malloc(size); };
globalParams.reallocFn = [](void * p, size_t size) { return realloc(p, size); };
globalParams.freeFn = [](void * p) { free(p); };
Jinx::Initialize(globalParams);
```

We won't describe these all in detail here, as there is extensive API documentation available. The general idea is that you may wish to replace the standard `malloc`/`free` or logging functions with your own application-specific functions.

Note that from here on, we'll assume you understand that all interface elements are part of the `Jinx` namespace, so you must either prepend everything with `Jinx::` or add a `using namespace Jinx` statement preceding these calls.

If you have a specific point in your shutdown procedure where you check for memory leaks, you'll likely wish to call the shutdown function before then. If you don't care about this, then the shutdown function may be omitted. Jinx objects will all clean up after themselves at program exit as their smart pointers are destroyed.

```
ShutDown()
```

Remember that it's your responsibility to release any objects you may be holding onto before this function is called.

Jinx uses an internal block allocator that makes small, frequent allocations and deallocations more efficient. If your own allocator has such functionality built-in, and you wish to disable this internal block allocator, you can uncomment the `#define JINX_DISABLE_POOL_ALLOCATOR` near the top of the `JxMemory.cpp` source file to disable that functionality.

The Runtime Object

The first thing your program will do after global initialization is to create a runtime object. Each runtime object represents a discrete runtime environment for Jinx scripts and libraries. If wish to keep specific types of scripts completely isolated from each other, then you may wish to create multiple runtime objects.

A runtime object is created as follows:

```
auto runtime = CreateRuntime();
```

All Jinx interfaces use `std::shared_ptr`, so the allocated objects will be deleted when they go out of scope and have no more references.

The runtime object is the interface through which you can compile scripts to bytecode, create scripts using that compiled bytecode, create or retrieve libraries, and other such functionality.

Let's look at how we might compile and execute a script:

```
// Compile the text to bytecode
auto bytecode = runtime->Compile(scriptText);
if (!bytecode)
    return; // Compile error!

// Create a runtime script with the given bytecode
auto script = runtime->CreateScript(bytecode);
```

Once we have a script object, we can then execute it at any time. Most of the other functions are variations on these core functions. One variation called `ExecuteScript()` compiles, executes, and returns a script object (if all steps were successful) in a single function call for convenience or testing.

It may be desirable to compile source just once and cache the bytecode if you plan on creating many separate script instances that share common source code. This would eliminate the runtime cost of compiling the same script multiple times. The bytecode is retrieved as a shared pointer to a `Buffer` object, so will only be deleted when all references to the object have been released.

Note that depending on the value of the `enableDebugInfo` in the global initialization parameters, the compiled bytecode may have some additional debug information appended in the form of an opcode to line number lookup table. This table can be stripped from the bytecode later using the function `IRuntime::StripDebugInfo()`. Leaving debug info in the bytecode doesn't affect runtime performance, so it is recommended to leave it in place unless you really need to minimize the memory or disk footprint of the bytecode.

Several of the functions used to compile the script also have a few other optional parameters. As an example, let's examine `IRuntime::CreateScript()` in detail:

```
virtual ScriptPtr CreateScript(
    const char * scriptText,
    void * userContext = nullptr,
    String name = String(),
    std::initializer_list<String> libraries = {}
) = 0;
```

In this instance, obviously, the first parameter is the script text to compile.

The second parameter of script creation is always a `void * userContext` parameter. This is useful to pass script-specific application data or objects to a script. The script is passed when calling library functions, and the context pointer can be retrieved via the script function `IScript::GetUserContext()`. In this way, you can enable library functions to operate on objects, calling member functions or accessing data. Naturally, being a void pointer, it falls on the programmer to ensure the data is valid and cast correctly before use.

Another parameter, typically the third after the user context parameter, is a string that acts as a unique identifier for compile-time or runtime identification and debugging. This is intended to be something like a script filename, an ID number, or some other way of identifying this particular script in log messages.

The last optional parameter is an initialization list of libraries to automatically import. If you specify one or more library names in this parameter, users will not have to bother typing an import statement at the beginning of each script. This can be helpful when scripts will always be calling functions within specific libraries, helping to avoid unnecessary typing at the start of each script.

Libraries are also created or retrieved by name using the `GetLibrary()` member function.

```
// Get or create a library by name
auto library = runtime->GetLibrary("custom");
```

We'll learn more about how to use library objects a bit later.

The Script Object

Once you have a script object, you can execute the bytecode in the script at any time using the `Execute()` function. The function returns true if execution was successful or false if a runtime error was detected.

```
// Execute script and update runtime until script is finished
do
{
    if (!script->Execute())
        return; // Execution runtime error!
}
while (!script->IsFinished());
```

While a real program would probably not spin in a loop indefinitely like this code does, this simple example demonstrates that a script may not be finished executing after the first `Execute()` call. It is expected that host applications may execute a script once per frame or simulation tick until execution has been successfully completed. Any notion of elapsed time or work performed would be handled by user-created library functions.

The `IScript` interface has an `IsFinished()` member function that returns `true` when the script is finished executing. A return value of `false` indicates that the program has suspended execution using the `wait` keyword.

`IScript` also has member functions `GetVariable()` and `SetVariable()` which can set or get variables by name. No local variables will exist before the script executes by default, but you can explicitly set new variables before you call the `Execute()` function. To allow the script to work with these externally set variables, you can declare "placeholder" variables using the `external` keyword, allowing you to use those variables inside the script just like any other variable.

In C++, you would use the following line of code before you call the script's `Execute()` function:

```
script->SetVariable("p", 123);
```

In Jinx script, getting and setting that variable would look like this:

```
-- Reserve the variable previously set via the API (p = 123)
external p

-- Assign the variable a different value
set p to 456
```

It's also possible to read normal variables after the script is finished executing as well. Let's assume we have a simple script like this:

```
set var to 789
```

As long as `var` has been declared at the *root level* (meaning it wasn't declared inside any scope blocks), once the script is finished executing, you can access the local variable using the `IScript::GetVariable()` function:

```
auto var = script->GetVariable("var"); // Variant containing value 789
```

This is an appropriate time for us to discuss how we interact with Jinx values. For that, we need to examine the `Variant` class.

The Variant Class

The `Variant` class in Jinx is the internal representation of stored variables in scripts and the runtime. When you access a variable via the `IScript::GetVariable()` function, you receive a copy of one of these objects. The class looks a bit unwieldy with a large number of functions, but it's actually fairly simple.

A C++ enum called `ValueType` lists all the types that may be represented by a `Variant` object. But more likely, however, is that you'll simply use the many helper functions to query types, or functions to retrieve those specific types.

`Variant` constructors will take any supported value type as a parameter, so that means you can generally pass any raw value to a function parameter that take a `Variant`, and it will be converted into the appropriate object type thanks to those overloaded constructors.

```
script->SetVariable("a", 123.45);
script->SetVariable("b", false);
script->SetVariable("c", "some string");
```

One thing to note is that while Jinx does use the `std::basic_string` class to represent strings, it uses a specialized allocator, making it incompatible with the standard library `std::string` typedef that uses the default allocator. As such, when passing or retrieving strings, you may have to use raw C-style string using the `std::basic_string::c_str()` function.

Let's assume we've retrieved a `Variant` object from some Jinx function, and need to extract the raw data from it.

```
script->Execute();
auto var = script->GetVariable("a");
```

In many cases, we know the specific type we're expecting. If this is the case, we can check with one of the type query functions.

```
if (var.IsNumber())
    // do something
```

Each type has a corresponding function for convenience, or, you can use the generic functions to get or check the type explicitly.

```
if (var.IsType(ValueType::Number))
    // do something

if (var.GetType() == ValueType::Number)
    // do something
```

There may be times that you're certain of the type, or else you can assume that a cast is acceptable. In this case, you can retrieve the value using the Get functions.

```
auto num = var.GetNumber();
```

It's important to remember that these `Get` functions will perform automatic value conversions when they can conceivably do so. If the cast is deemed inappropriate – for

instance, if the original example was a string that couldn't be parsed into an integer, the function would return a value of zero and log a warning message. It's your responsibility to check types if needed.

If you need to check against a specific value, you can use the implicit construction of appropriate types to streamline your code. For example, you can check for a variable like this:

```
if (var.GetVariable("a") == "some string")
    // do something
if (var.GetVariable("b") == 456)
    // do something
```

In both these cases, appropriate `Variant` objects are implicitly created based on the types you're comparing against, and the overloaded `Variant` equality operator is then called to check equivalence.

There is a `Variant` member function called `CanConvertTo()` to check for conversion viability to a given type. For instance, if a `Variant` object contains a string with the value `"42"`, the string can be converted to an equivalent number, and the function would return `true`. If the value is instead `"banana"`, then no logical numeric conversion could occur, and the function would return `false`;

```
if (var.CanConvertTo(ValueType::Number))
    // do something
```

Additionally, there is a `Variant` member function called `ConvertTo()` which will not just check the viability of a conversion, but will apply that conversion to the variant object if possible. The function still returns `true` or `false` indicating success, and so can be used in mostly the same way as the previously described function.

```
if (var.ConvertTo(ValueType::Number))
    // do something
```

The Library Class

Libraries are the means by which you organize multiple scripts or native methods into reusable modules that can be consumed by other scripts. As we saw earlier, we can either create a new library or retrieve an existing library with the runtime's `GetLibrary()` function. In either case, it means a library by the name given is now registered with the runtime. Any new scripts compiled with that runtime will now be able to access this library.

In order to compile scripts that call library script functions or properties, those library scripts must either be compiled with the runtime, or the bytecode for the libraries must be

executed, or those elements must have been registered via the native API. This is required because the runtime parser must be able to check a script's content against a library's function signatures and property names.

Jinx has no dependency system to automate this process, because it has no way of retrieving scripts or bytecode on its own. Your application is completely responsible for ensuring library scripts are parsed and/or executed before any dependencies to ensure those scripts can be compiled.

You should understand that Jinx registers library functions and properties for use by executing the scripts in which those are defined. From then on, the functions and properties are defined in the libraries and the runtime contains the data required for execution.

This has ramifications for how scripts should be organized. In general, while it is possible to execute scripts that contain library function and property definitions multiple times, it's wasteful and unnecessary to do so. Instead, those library function and property definitions should be separated into scripts that can be compiled and executed only once at program startup.

Native Property Registration

Library properties can be set from native code. This is done with a single function call. The following code creates a public property "someprop" with a default value of 42.

```
auto library = runtime->GetLibrary("test");
library->RegisterProperty(Visibility::Public, Access::ReadWrite, "someprop", 42);
```

The `ILibrary::RegisterProperty()` function takes four parameters. The first parameter sets visibility to `Visibility::Public` or `Visibility::Private`. The second parameter is access type, which can be `Access::ReadWrite` or `Access::ReadOnly`. The third `String` parameter is the name of the property. The fourth `Variant` parameter is the default property value.

Native Function Registration

The library interface can also register functions. The member function `ILibrary::RegisterFuntion()` is used to do this. The native functions to be used as a callback should look something like the following:

```
static Variant ConvertValue(ScriptPtr script, Parameters params)
{
    return params[0] * 2;
}
```

The script that executed the function is passed as the parameter `script`, and any function parameters are passed as `params`, which is actually a `std::vector` of `Variant` objects. The number of parameters is guaranteed by the Jinx parser, so you don't have to worry about checking to see if parameters exist before using them. Remember that a list of parameters separated by commas is automatically converted into a collection, even though it may look like a variable number of arguments.

The function must return a `Variant` object. If the function signature does not require a return value, then you can simply return `nullptr`, which will be turned into a null `Variant` and subsequently ignored.

Registering the function looks something like the following:

```
auto library = runtime->GetLibrary("test");
library->RegisterFunction(Visibility::Public, "convert {} value", ConvertValue);
```

The first parameter indicates public or private visibility. The second parameter is a string containing a function signature similar to what you'd write in a Jinx script, with the difference being you don't need to provide parameter names. Parameters are indicated with curly braces in the string (e.g. `"{}"`), with an optional value type to cast the parameter to (e.g. `"{integer}"`). The third parameter is the native function callback.

Native Function Registration Using Lambda Syntax

There are some alternate ways to provide Jinx with a callback function, such as by using C++ lambda syntax to provide the function call inline. Here's what the previous example would look like when using lambda syntax:

```
library->RegisterFunction(Visibility::Public, "convert {} value",
    [](ScriptPtr script, Parameters params)->Variant
{
    return params[0] * 2;
});
```

Advanced Script Topics

Although Jinx is largely a procedural language, there are a few ways Jinx is specifically designed to work with C++ objects. It is expected that a common pattern will be to attach scripts to objects with a finite lifespan, and to execute the script with the intention of calling a member function specific to that object. Jinx provides two ways to do this.

Using the Script-Specific User Context Data

The Jinx script object can hold a single piece of "user context" data in the `std::any` container, which can be passed at script creation time. `std::any` is a more type safe alternative to a `void *` pointer. When native callback functions are executed, the calling

script is passed to the function. The user context data can then be cast back to its original type inside the callback function using `std::any_cast`. If it was a pointer to an object, the function is then free to call a member function. Here's an example of how this might look. Let's assume we have a class with member functions as follows:

```
void SomeObject::Initialize()
{
    // Create and execute script
    m_script = m_runtime->CreateScript(m_scriptBytecode, this);
    m_script->Execute();
}

void SomeObject::MemberFunction()
{
    // Do something via a script call
}
```

Now let's also assume we've registered script function like the following:

```
Jinx::Variant CallFunc(Jinx::ScriptPtr script, Jinx::Parameters params)
{
    auto someObject = std::any_cast<SomeObject *>(script->GetUserContext());
    someObject ->MemberFunction();
    return nullptr;
}

void RegisterScriptFunctions(Jinx::RuntimePtr runtime)
{
    auto library = runtime->GetLibrary("mylib");
    library->RegisterFunction(
        Jinx::Visibility::Public,
        "call func",
        CallFunc);
}
```

You can see that we've passed the object's `this` pointer to the script creation function. We can then cast it back in the callback function and use that pointer to call a member function. Obviously, care must be taken not to let any such script exceed the lifetime of the object holding it, as otherwise you will get a dangling pointer.

std::any and macOS 10.14 Limitations

You should note that the use of `any_cast` is restricted to macOS 10.14 (Mohave) and above. If you wish to target macOS versions lower than this, Jinx provides aliases that can substitute `void *` and `reinterpret_cast` in place of the more modern `std::any` container and `std::any_cast`. If you wish to switch between these, you can comment out or conditionally modify `JINX_USE_ANY` in Jinx.h, and instead use the `Any` and `JinxAnyCast` aliases.

Calling Script Functions From Native Code

It's also possible to call Jinx library functions from the script object interface, whether those are script functions or even registered native callbacks. Let's assume we have a script object that has already executed this script, which means the function is now registered as part of the runtime and is available to call:

```
public function {number x} minus {number y}
    return x - y
end
```

Note that in order to call a script function from C++, it must be declared as either a `public` or `private` library function. Remember that when creating public or private functions in a script, if no library is declared, the function is added to the *default library*, which can be referenced with an empty string.

To call this function from a script object requires two steps. First, we have to acquire the runtime ID of the signature, which is a 64-bit hash generated from the function signature. This is how Jinx represents functions and properties internally. Here's how we retrieve that ID value.

```
auto id = script->FindFunction(nullptr, "{} minus {}");
```

If the `id` value is not equal to `InvalidID`, then it means the function was successfully located in the runtime.

The first parameter is the library to search in. A null pointer indicates that the function should search in the script's assigned library. Next, we have a string representing the function definition. Remember, if the match is not exact, the hash ID will not match either, and the function won't be located.

Now, let's call the function using that `id` value, as well as passing it the parameters the function requires and finally, retrieving an optional return value:

```
auto val = script->CallFunction(id, { 5, 2 });
```

We've now called the script function `"{} minus {}"`, and passed it two parameters, a `5` and a `2`. Naturally, we expect an answer of `3` in `val`.

There are a few things to remember about calling functions in Jinx. As we mentioned earlier, you must remember that you must execute the scripts that contain these functions before they can be accessed by any other scripts. Interestingly, it doesn't really matter which script you call them from, because the execution is isolated from the normal script

execution. You can even create a stand-alone no-op script specifically for the purpose of creating an execution environment, like this:

```
auto script = runtime->ExecuteScript("");
```

Odd as it may sound, an empty string is a legitimate Jinx script which can be compiled or even executed. Naturally, it just no-ops and exits. This little trick may be useful if you wish to execute a Jinx library function but don't have a script handy to call it from.

String-based Table to Collection Conversion

Jinx has a somewhat specialized string to collection conversion. You can automatically convert a comma-delimited or tab-delimited table from a string to a collection. Detection is performed automatically based on the number of tabs vs commas in the first row. The first column is assumed to be the key for rows, while the top row is assumed to be keys for columns. When the collection is finished, a collection consisting of the content of the first column is created. Inside each value of that initial collection, a collection consisting of the corresponding row is created.

Let's look at a simple example to see how this might work. Assume we have a table that looks like the following:

Name	Age	Gender	Profession
Mark	32	male	baker
Judy	29	female	accountant
William	48	male	carpenter

If this file is saved as a comma or tab delimited text file and loaded as a string, it can be stored in a variable, then converted from a string to a collection. This can be done either in native code or in a script. In native code, it would look like this:

```
Variant table = tableText; // Assume tableText contains the table as a string
table.ConvertTo(ValueType::Collection);
```

In a Jinx script, a variable can be converted from a string to a collection as follows:

```
set table to table as collection
```

Now, this variable, when accessed in code, can use `[row][column]` format to access any element in the array. For instance, you can access Judy's gender as follows:

```
set a to ["Judy"]["Gender"]
```

If you don't know the values in the first column, you can iterate through the collection and either obtain the key name directly using the get key function, or access the first column by its key (in this case, `"Name"`), since it's stored just like every other column value.

Thread Safety and Concurrency

In addition to each script executing as a co-routine, all library and runtime member functions are thread-safe, allowing both scripts and external code to safely access shared data and functions from independent threads. However, script member functions are *not* thread-safe, so you must take care to only access each individual script from a single thread or protect the script from simultaneous access from multiple threads with your own code.

This should allow considerable freedom for using Jinx in any number of application-specific multi-threaded scenarios.

Exceptions

The Jinx library does not use C++ exceptions. This reflects the reality that many game developers and development platforms do not use or support exceptions, and Jinx was specifically designed with this particular audience in mind.

Strings and Unicode

Jinx uses UTF-8 internally to represent all string data, and uses a custom typedef of `std::basic_string<char>` called `String`, so it can use the Jinx library allocators. However, many programs use `char16_t` or `wchar_t` strings.

If you wish to pass `char16_t` or `wchar_t` strings to the Jinx API, you can use the conversion function `Str()` to do this. Additionally, when retrieving string data from the `Variant` class, you can explicitly get wide-character or U16 strings as well. Here's an example of how to do this:

```
const wchar_t * wstr = script->GetVariable(Str(L"x")).GetWString().c_str();
const char16_t * str16 = script->GetVariable(Str(u"x")).GetStringU16().c_str();
```

Conclusion

I hope this tutorial has been helpful in giving you an overview of the Jinx language and API. Feel free to e-mail me at james.boer@gmail.com or contact me via the GitHub project at https://github.com/JamesBoer/Jinx. I'm curious to hear if anyone makes use of this scripting library besides me, and how it works out for you.