Jinx Tutorial

Introduction

Jinx is a lightweight embedded 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 embedded 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 typed, and are easily converted between types by casts or assignments. 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 clean 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 by 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.
- 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

You're probably anxious to get started using Jinx, so let's go over what needs to happen to get the library working in your own project.

Jinx Prerequisites

Jinx is written in C++ 11/14, and as such, requires a compiler that conforms to the latest ANSI standards. The Jinx library has at least one language feature (unrestricted unions used in the `Variant` class) that makes it incompatible with Visual Studio 2013 and earlier. The library compiles cleanly with Visual Studio 2015 on Windows, the latest version of Xcode on Mac using LLVM, and with G++ on Linux.

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

Compiling Jinx

While a few sample projects are offered for selected environments, it should also be straightforward to include the code in your own project. The entire library is contained in a single folder. Add all Jinx source files to your project using your native IDE or make system, include `Jinx.h` as appropriate in your source, and you should be ready to start using the library.

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'll 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 a 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 marks the termination of a statement. Combining multiple statements onto a single line is not allowed. No other whitespace is significant, except for its role in separating tokens.

Case Sensitivity

Jinx is case sensitive, primarily because case insensitivity is a somewhat thorny problem when you support most Unicode codepoints as identifiers, and can produce different results depending on the current locale. Since functions and identifiers in Jinx aren't limited by whitespace restrictions (meaning there is no reason for CamelCase words), the recommended convention is to simply avoid any capitalization altogether, but there is no requirement to do so.

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 implicitly converted between types without runtime warnings when the conversion would not result in a loss of data. There is no keyword to declare variables, and they can only be declared as part of an assignment operation.

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.

```
var1 is null                -- null type
var2 is 123.456             -- number type
var3 is 42                  -- integer type
var4 is false               -- boolean type
var5 is "a string value"    -- string type
var6 is 1, 2, 3             -- collection type
```

Jinx uses the `is` keyword for assignment (the `=` operator is reserved for equality tests). Aside from a few special exceptions (loops with index variables and function parameters), all variables are initially declared on the left side of an `is` (assignment) operator, which names the variable and assigns it an initial type and value. If you try to use a variable that hasn't been assigned, 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 purposes 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`. Any type 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 reference 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 internally by the script when looping over a collection. It is not directly accessible by scripts.

UserObject

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

Buffer

A memory buffer object is useful for storing and retrieving custom data in scripts. Like collections, 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 as part of application-written functions.

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
some variable is 123
'some variable' is 123 -- Same result as previous line
resumé is "my resumé text"
```

Casting and Type

Variable 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
a is 123
b is a as string    -- b = "123"
c is b as integer   -- c is 123

-- The 'type' keyword retrieves a variable's type
d is a type         -- d is 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.

```
a is 32          -- Declare a variable at the outermost scope
begin
    b is a       -- Legal
end
c is b           -- Error: b is 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.

Mathematical Operators

Five basic math operators are supported: addition, subtraction, multiplication, division, and modulus.Mathematical expressions work in a fairly typical fashion, but unlike in C, there is no operator precedence. Operations are evaluated from left to right, except when precedence is explicitly defined via parenthesis.

```
-- Evaluate expressions
a is 2 + 3 - 1      -- 4
b is a * 4 / 2      -- 8
d is 2 + 2 * 3      -- 12
e is 2 + (2 * 3)    -- 8
```

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.

```
a is 3 / 2              -- 1.5
b is 3 / 2 as integer   -- 1
```

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.

```
a is 5 % 3          -- 2
b is 5 % 3.5        -- 1.5
```

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 our example below, a value of `false` is assigned to `a`, and `true` is assigned to `b`.

```
a is 1 = 2     -- false
b is 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.

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

Comparison operators have the same precedence as mathematical operators, and so will be evaluated strictly from left to right unless otherwise indicated by parentheses.

Logic Operators

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

```
a is true and false   -- false
b is 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 the `and` keyword are evaluated first

```
a is 1 < 2 or 4 != 5       -- true
```

The `not` operator has a lower precedence than all other operators, and so will negate the *entire* expression that follows it. If you wish to negate only a portion of an expression, then you can use parentheses to do so.

```
b is not 1 < 2 or 4 != 5   -- false
c is not (1 < 2 or 4 != 5) -- false (equivalent)
d is (not 1 < 2) or 4 != 5 -- true (not equivalent)
```

Increment and Decrement Statements

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

```
a is 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.

```
a is 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 as statements, not operators.

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`.

Jinx will automatically convert numbers, integers and strings to Boolean values when required. Any non-zero or non-empty value is generally converted to a `true` value, while zero values and null are equivalent to `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.

Creating an empty collection can be done by using empty index operators.

```
coll is []
```

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 with a key equal to the current count + 1, or the next highest free key value available. This remains true even if elements are removed or non-numeric keys are added.

```
coll is 5, 4, 3, 2, 1
```

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.

```
x is coll [2] -- x = 4
```

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.

```
c is coll size -- c = 5
```

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...
keyed coll is [1, "one"], [2, "two"], [3, "three"], [4, "four"], [5, "five"]
```

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

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

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

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

```
coll [6] is "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 order will be sorted by type.

```
coll is ["apple", 345], [111, "orange"], [12.34, false], [-99, null]
```

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

```
coll is ["fruit", ["apple", 345]]
```

Adding or removing elements to an existing collection can be done using core library functions. These functions can add or remove one or many elements, depending whether a single key value or a collection of keys are passed as the first argument.

```
-- Add key-value elements to container
add ["peach", 999] to coll

-- Remove element from container by index
remove "peach" from coll
```

The `remove value/values {} from {}` function must iterate over the entire list searching for values to remove, so be aware of the performance implications.

```
-- Remove value/values from collection in linear time operation
remove value 999 from coll
remove values 999, 345 from coll
```

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

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

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.

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 these expressions are evaluated only once as the loop starts.

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. The script will compare values 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 any 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
```

Looping over a collection uses a slightly different syntax, using the `over` keyword. An optional variable can be specified to hold a copy of the value in each element.

```
'my list' is 5, 4, 3, 2, 1
loop over 'my list'
```

```
    write line "element"
end

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

Jinx also supports the `while` keyword to check for arbitrary looping conditions. You may position the statement either at the beginning or end of the loop.

```
-- While loop with condition at front
x is 1
loop while x < 10
    increment x by 2
    write line "x = ", x
end

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

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
```

You'll notice that `and` is a reserved keyword, which doesn't matter at all, since `wait`, `between`, and `seconds` (all 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
```

In fact, this is also perfectly legal, so long as the name was defined with that particular spelling variation as well.

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.

```
wait is 123
wait between 0.1 and 1 second
```

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, followed by an optional `return` keyword, indicating whether a return value is expected. 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.

Function Parameters

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

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

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

A return value is indicated by the return keyword following the function declaration. 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.

Complex Expressions as Parameters

When passing arguments to expression, you may use any legal expression as a parameter. There are a few caveats to be aware of though, due to the use of whitespace available to functions. This can cause problems for the parser when complex expressions are passed as parameters. By complex expressions, we are referring to expressions involving multiple parameters in a list, the use of binary operators, other function calls, and so on – essentially, anything more complex than a simple constant or variable. Here is such an example:

```
a is 1 + 2 minus 2   -- error!
```

Jinx can't resolve expressions such as this, so it needs a bit of assistance to indicate which expressions are to be passed as a parameter, like this:

```
a is (1 + 2) minus 2   -- a = 1
```

This is not a problem for arguments passed at the *end* of the function, however, because there is no ambiguity there, as the expression is naturally delimited by the newline indicating the end of the statement. So, the following statement is perfectly legal.

```
a is 3 minus 1 + 1   -- a = 1
```

To demonstrate another example, this allows us to pass lists of parameters to the write line function without having to surround the arguments in parentheses.

```
write line "three ", "two ", "one ", "go!"
```

Functions as Parameters

When passing the results of other function calls as parameters, you must be cautious that precedence works as intended. Since function signatures take precedence over other types, the parser may see additional legal function calls as it moves from left to right over an expression. Let's again use the previous function as an example. The two following statements are equivalent:

```
a is 3 minus 2 minus 1   -- a = 2
b is 3 minus (2 minus 1) -- b = 2
```

Because "`2 minus 1`" is a legal function expression, it will be parsed as an expression to be passed instead of the constant `2` in the initial expression "`3 minus 2`". Since this could be a potential source of confusion, it is recommended to use parentheses in order to explicitly specify the results you want:

```
c is (3 minus 2) minus 1 -- a = 0
```

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 explicitly cast. Jinx is a dynamically typed language and will not generate compile-time errors due to type-related issues, but will provide runtime warnings if invalid types are cast. For instance, a collection can't be cast to an integer without generating a runtime warning.

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

a is "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"]
```

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 `yield` keyword.

```
import core

-- Assume "get current time" function exists and retrieves time in seconds
t is get current time + 5

-- Loop for five second
loop while t < get current time
    yield -- 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:

```
-- Yield for five second
yield while t < get current time
```

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

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.

```
public 'public prop' is "I'm a public property"
public 'private prop' is "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.

```
readonly public 'public readonly prop' is "I'm a readonly public 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 embedded 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 consider 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.logFn = [](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.

Several of the functions used to compile the script also have a few other optional parameters.

One of these, typically the second parameter after the script text itself, 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 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 `yield` 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, of course, but you can explicitly set new variables before you call the `Execute()` function. However, the script can't directly access variables by name that it doesn't know about, determining that to be a syntax error. The core library has a function called `variable` that takes a string value indicating the name of the variable and returns its value.

In C++, you would call this as follows:

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

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

```
import core

-- get the variable set in code (somevar = 123)
somevar is variable "p"

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

This function will retrieve the value of a local variable, but obviously this is only useful for undeclared variables set by external code as parameters, since otherwise you could simply access the variable by name as is typically done. There is a corresponding function `set variable {} to {}` as well, if you wish to set those undeclared variables to some value as well.

However, because it is possible to read *any* variable after execution is finished, the `set variable {} to {}` function is a bit less useful than its cohort which reads variables. For instance, take the following script:

```
var is 123
```

So 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 like this:

```
auto var = script->GetVariable("var")->GetInteger(); // var = 123
```

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::string` class to represent strings, it uses a specialized allocator, making it incompatible with the standard library `std::string` that uses the default allocator. As such, when passing or retrieving strings, you may have to use raw C-style string using the `std::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.

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(false, true, "someprop", 42);
```

The `ILibrary::RegisterProperty()` function takes four parameters. The first `bool` indicates whether the property is read-only (`true`) or writeable (`false`). The second `bool` set the scope to public (`true`) or private (`false`). 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` type `Variant` and subsequently ignored.

Registering the function looks something like the following:

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

The first `bool` parameter indicates whether the function is public (`true`) or private (`false`). The second `bool` parameter indicates whether a return value is expected. The third

parameter is an initialization list of `String` objects which is used to generate the name signature. 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 fourth parameter is the native function callback.

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, however, 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 simply 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.

UTF-8 vs UTF-16

Jinx uses UTF-8 internally to represent all string data. However, many programs use `char16_t` strings or, more likely among Windows programmers, `wchar_t` strings. Many programmers are surprised to learn that `char` strings can just as easily (one might even argue *better*) represent all Unicode codepoints. Moreover, `wchar_t` is not a platform-independent type, as it is represented as 16-bits on some platforms (like Windows) and as 32-bits on others (like Linux). As such, it's not a suitable internal representation for a library making efforts to be as platform-neutral as possible. C++ 11's new `char16_t` type is a more suitable type for representing UTF-16 data. However, UTF-8 strings avoid endian issues when serializing data, and are more compact when mostly representing ASCII text, as is expected of a scripting language.

Jinx provides functionality in its `Variant` class to convert back and forth between UTF-8 and UTF-16 encodings. A `StringU16` typedef is provided, as well as some specific functions and overloads to handle conversions. You can pass `char16_t *` literals to the `Variant` class overloaded constructor, and the string will be converted to UTF-8 internally. If you wish to retrieve a UTF-16 string, you can use the `Variant::GetStringU16()` function, which will perform the conversion from UTF-8 to UTF-16.

Because of the non-portability of the `wchar_t` type, the `StringU16` type uses `char16_t` rather than `wchar_t`. If your code uses `wchar_t` everywhere and you can assume that it will be 16-bits based on your supported platforms, you can cast the C-style strings yourself when using the conversion functions.

Conclusion

I hope this tutorial has been helpful in giving you a quick running start with 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.