The Ferite Programming Manual

Chris boris Ross

chris@darkrock.co.uk

Blake Watters

blakewatters@nc.rr.com

The Ferite Programming Manual
by Chris boris Ross and Blake Watters

Table of Contents

Chapter 1. Introduction

What is ferite?

Ferite is a small robust scripting engine providing straight forward application integration, with the ability for the API to be extended very easily. The design goals for ferite are lightweight - small memory and CPU footprint, fast, threadsafe, and straight forward both for the programmer of the parent application and the programmer programming ferite scripts to learn the system.

What does this documentation provide?

This document is the official commentary on ferite including language information such as constructs and known issues. An API guide for the standard objects provided with every ferite distribution and the means in which to embed ferite are proivided seperately.

Why should I choose ferite?

Ferite is designed to be added into other applications. With a constant API your application will be able to stay binary compatible with the latest ferite engine. This is very good because it allows you, the application programmer, to add powerful scripting to your application without having to worry about the actual internals. Ferite provides type checking, and does a lot of work for the programmer to keeps things as simple as possible.

Ferite provides a language very similar to that of C and Java with additional features from other languages (e.g. Regular expressions in the style of Perl). This means that the skill set acquired through learning these main stream languages can be instantly applied to the ferite scripts. Ferite is by no means a heavy language, it has kept the small language size of C which allows it to remain fast and lightweight. There is also the ability to push the language further with native classes, objects, namespaces, variables and methods.

Ferite also has a very small system memory and disk foot print making it ideal for it's use.

If you are looking for a scripting engine that is threadsafe, and also allows for multi-threading within an application - ferite is the way to go. It relies on operating system threads which allows ferite to scale to multi-processor systems very easily.

Chapter 2. Language Reference

Scripts

Scripts are made up of two main parts and are written as follows:

```
Class, Namespace, Function, Global and Use definitions
Main program code
```

The **Main program code** is what is called when the script is run. All parts of a program have to be declared before they are used. This is to keep the code clean rather than a name resolution reason (all names are resolved at runtime within ferite). The **Main program code** is equivalent to the main() method within a C or Java program.

An example script (the famous Hello World program):

```
uses "console";
Console.println( "Hello World from ferite" );
```

The 'uses' statement is used to import API either from and external module or from another script and is described in greater depth later on. This is highly useful as the default operation of ferite has no API.

Comments

These are possibly the most important feature of ferite. Seriously. Ferite supports two methods of commenting code either the C style (/* */) or the C++ style (//). These can be placed anywhere within the scripts. All I can say is use them - comments make peoples life so much easier.

```
// This is a comment
/* This is another comment */
```

Types

Ferite is a semi-strongly typed language. This means, unlike in Perl or php, you have declare variables that have to be used and what type they are. Ferite has a number of types within it's system. The simple types are **number** (automatically switches between integer or real number system), **string**, **array**, **object** and **void**, and are described below:

number

This type encapsulates all natural and real numbers within the 64bit IEEE specification. Ferite will automatically handle issues regarding overflow and conversion. Several things have to be said about the number type:

- All numbers start out as C longs (64bit integers). When the value goes over the maximum value allowed for long the type will switch over to a C double.

- Comparisons can be made between numbers but it should be noted that once a number has internally become a double, equality comparisons are likely to give unexpected results. To add this and make things slightly more reasonble, when doubles are being compared there is a slight amount of tolerance involved which means that they do not need to be identical but very very close in vaule.

```
Example:
number someValue = 10;
number someOtherValue = 1.21;
number newValue = someValue + someOtherValue;
```

string

Strings are specified using double quote ("") and can contain control characters. The control characters are defined as in C by use of \'s. Anything that ferite recognises as an evaluable construct within the string will be evaulated.

It is possible to access individual characters within the strings using square bracket notation (described later on).

You can reference other variables or even place small expressions within the strings such that they get evaulated at runtime. This allows for the complicated construction of strings to be less painful. To reference a variable you simply prefix it's name with a dollar symbol '$' (It should be noted that the string representation for the variable will be used. For objects that contain a `toString()` method, the method will be called and it's return value used.). To reference an expresion you use a dollar symbol followed by a set of curly braces '{ }' with the expression placed between them. For example, the following code will print out "Hello World" and then print out 2.

```
Example:
string test = "Hello";
Console.println( "$test World" );
Console.println( "${(1 + 1)}" );
```

Strings can also be defined using single quotes (''), these differ from the above notation because everything within the single quotes is quoted. This means that the above variable substitution and escape sequences will not work.

array

Array's provide a method of storing a collection of information and accessing it randomly. The contents of the array either be accessed by means of a hash key or indexed by means of a number. A variable of any type can be placed within an array - this means that you can mix numbers, strings, objects, and even other arrays. This is a very useful feature as it allows you to group together likewise data on the fly. Arrays are accessed using the "[ ]" notation as in other languages, this is discussed later under the Index Operator section.

The standard way of creating arrays is to declare a variable and then modify it using various different operations. Most of the time this is fine albeit a bit overkill. To make array creation slighly easier there is a notation that can be used. You simply create the array by using a pair of '[' ']' brackets and placing a comma seperated list of values between them.

```
Example:
array a = [ 1, 2, 3 ]; // Declare an array 'a' and initialise it to have 3 elements
```

object

Objects are instances of classes. When first declared they point to the null object (this allows the user to check to see whether the object has been instantiated). To create an instance please see the new operator. The null object is always within a script and can be referenced using the keyword **null**.

```
Example:
object o = null;
object o2 = new SomeObject();
```

void

The void type is similar to Perl's and php's type when first delcared. Anything can be assigned to them, but after having something assigned to them, the void type is removed and the variable then becomes whatever type was assigned to it. e.g. If a void type is created and has a number assigned to it, it can't then have a string or object assigned to it as it has become a number.

It is important to note that the only thing that can't be assigned to a variable of type void is a function. You can assign namespaces and closses to variables of type void and then use them as normal.

```
Example:
void v = null; // v is now an object pointing to null
void v2 = 42; // v2 is now a number with the value 42
void v3 = SomeNamespace; // v3 points to SomeNamespace
```

Variables

Variables are simply instances of types and can be declared within a initializer and with other variables using the following syntax:

```
modifier type name [= <expression>] [, ...] ;
```

- modifier

  This is used to define properties of a variable. Current properties are: **final** and this sets, whether or not after a variables first assignment, whether the variable is constant and therefore can't be changed. This is the same behavior shown by the final keyword within Java. **static**, which tells ferite whether or not the variable is tied to a class or object (discussed later on). And **atomic** which tells ferite to guarantee that all operations on the variable are atomic (and therefore threadsafe). It is up to the programmer to tell ferite this because atomic variables have an added overhead which is simply not necessary for 99% of program code.

- type

  This is the type of variable that you wish to declare. It can be void, number, string, array or object.

- name

The name of the variable to be declared. The name must start with a alpha character or underscore and after that may contain underscores _, numbers [0-9] and other alpha characters.

- [= <expression>]

  Variables can be declared with a non-default value rather than the internal defaults. (number = 0, string = "", object = null). It is recommended that you make sure that your variables are initialised before you use them purely to make the programs behavior more understandable.

  Please Note!

  When it is declared within a function you can specify any valid expression to be used as the variable's initialiser - eg. a return from a function, the adition of two previously declared variables. When the variable is declared globally, in a class or a namespace you can only use a simple initialiser by this I mean you can only initialise a number with a integer or real number (eg. 120 or 1.20), and a string with a double or single quoted string. It is **not** possible to initialise an array or object in a namespace, global, or class block - these will have to be done using a function.

- [, ...]

  Rather than having to declare modifiers and type again for a set of variables it is possible to simply add more names in a comma seperated list. Please see below for an example.

- [;]

  This terminates the statement.

Example

```
number mynumber = 10, another_number;
final string str = "Hello World";
object newObj = null;
array myarray;
```

A variable's scope is as local as the function they are declared in with the exception of explicit global variables. This means that a variable declared in function X can only be accessed within function X. Global variables can be accessed anywhere within a script, and are declared using the following syntax:

```
global {
 <variable declarations>
}
```

Unless explicitly defined a variable is considered local. There are a number of pre-defined global variables within a ferite script, these are null and err. null is used to allow checking of objects and instantiating, and err is the error object used for exception handling.

Expressions

Almost everything written in ferite is an expression - they are the building blocks of a program - they are combined to build other expressions which are in turned used in others using operators. Expressions are built up using various operators, variables and other expressions, an example being say that adding of numbers, or creating an instance of a new object. Expressions are made clearer when discussing operators as these are what are used to build them.

Truth Values

What constitues a truth value?

- A number that is not zero is considered as true, this also means that negative values are also true. It has to be noted that if a number has switched into real format it is never likely to be considered false. Currently ferite deals with this by binding false to the range $\pm$ 0.00001 (NB. This is likely to change later).
- A string that has zero characters is considered false, otherwise it's true.
- An array with no elements is false, otherwise is considered true.
- An object is considered to be false if it doesn't reference any instantiated object.
- A void variable can't be true.

There are currently two keywords that can be used 'true' and 'false' these are of type number.

Example:
```
number shouldKeepDoingThings = true;
```

Operators

Ferite comes with a whole bundle of operators to play with. They allow you to do basic things such as arithmetic operations all the way to on-the-fly code generation and execution. With each operator it's action on different types will be described. When an operator is applied to types it can not deal with, an exception is thrown and must be handled (see exception handling).

Arithmetic Operators

- Addition, this operator adds two variables together. Currently it only applies to numbers and strings. Adding strings together acts as concatenation, and adding a number onto a string will cause it to be converted to a string and concatenated. When an object gets asked to be added to a string, the operator checks for a `.toString()` function in the object and calls that - this means that you as the programmer can provide a custom string representation of the object. This is the same mechnism that is used when you reference variables in strings using '$'.

Table 2-1. Addition '+'

Left \ Right	void	number	string	array	object
void					

number		This will return a number. If the left hand side or the right hand side is in the floating point form the resulting number will be floating point.			
string	The left hand string with `(void)` at the end.	The left hand string with the value of the number converted to a string.	A string containing the left hand side with the right hand side concatenated onto the end.	The left hand string with the string representation of the array concatenated onto the end.	The left hand string with the result of the toString method in the object called.
array					
object					

- Subtraction

Table 2-2. Subtraction '-'

Left \ Right	void	number	string	array	object
void					
number		This will return a number. If the left hand side or the right hand side is in the floating point form the resulting number will be floating point.			
string			A string containing the left hand side with all the occurances of the right hand side removed.		

array					
object					

- Multiplication '*' only applies to number types. The result is the left hand side multiplied by the right hand side. If the left hand side or the right hand side is in the floating point form the resulting number will be floating point.
- Division '/' only applies to number types. The result is the left hand side divided by the right hand side. If the left hand side or the right hand side is in the floating point form the resulting number will be floating point. If neither are floating point then the division will be interger based division.
- Modulus (%) - Returns the remainder of integer division between two number variables. If the numbers are in real format they will be implicitly cast into integers and then the operation will be done.

Assignment Operators

The basic assignment operator is '='. This will make the left hand side variable equal to the right hand side. This is a copy value operator which means that the right hand side will be copied and then assigned to the left hand side. This is true with exception of objects where the left hand side will reference the object and it's internal reference count will be incremented.

It is possible to extend the operator by placing one of the Arithmetic operators in front al-la C. e.g. +=, -=, *=, /=, &=, |=, ^=, >>=, <<=. It will have the effect of taking the existing left hand side, applying the arithmetic operator with the right hand side and then assigning it back to the left hand side.

As was mentioned when discussing the **void** variable type, anything can be assigned to a variable of type void. This can only be done once as the void type mutates to the type to which has been assigned to it.

Comparison Operators

These are used to compare variables. It is only possible to compare like variable types, i.e you can only compare strings with strings, and numbers with numbers. They are all straight forward and act as would be expected from their name.

- Equal To (==) - true if both sides are equal.
- Not Equal To (!=) - true if both sides aren't equal.
- Less Than (<) - true if the left hand side is less than the right.
- Less Than Or Equal To (<=) - true if the left is less than or equal to the right hand side.
- Greater Than (>) - true if the left hand side is greater than the right.
- Greater Than Or Equal To (>=) - true if the left is less than or equal to the right hand side.
- isa (isa) - true if the left hand side expression is of type stated on the right hand side.

  eg.

```
"Hello World" isa string => true
42 isa string => false
```

- insantceOf (instanceof) - true if the left hand side expression is an instance of the class stated on the right hand side.

  eg.

```
Console.stdin instanceof Sys.StdioStream => true
Console.stdio instanceof Test => false
```

Incremental and Decremental Operators

These allow incrementing and decrementing of variables. Currently it only works with numbers.

- Prefix Increment (++someVariable)
- Postfix Increment (someVariable++)
- Prefix Decrement (--someVariable)
- Postfix Decrement (someVariable--)

If you have programmed within C or Java before you will know how these work. They both do what they say on the tin, but the difference between Pre and Post fix is subtle (but at the same time very very useful). With the prefix version the variable is in/decremented and the new value is returned, with the postfix the variable is in/decremented and the **previous** value is returned. e.g.

```
number i = 0, j = 0;
j = i++; // j = 0, i = 1
j = ++i; // j = 2, i = 2
```

Logical Operators

These apply to truth values and tend to be used for flow control.

- Not (!) - true if the expression it is applied to is false. You can also use the keyword **not** to represent the operator. This is useful when writing clean code.
- And (&&) - true if both variables/expressions are true. It is also possible to use the keyword `and` to represent this operator.
- Or (`||`) - true if either variable/expression is true. It is also possible to use the keyword `or` to represent this operator.

Bitwise Operators

It must be noted that when numbers are passed to the bitwise operators their values are explicitly cast into a natural number if they happen to be floating point. This does not modify the variable being passed.

Example:

`10 & 11.1` `will actually be` **`10 & 11`**

- AND (&) - does a bitwise AND on the two variables passed to it.
- OR (|) - does a bitwise OR on the two variables passed to it
- XOR (|) - does a bitwise XOR on the two variables passed to it
- Left Shift (<<) - does a bitwise left shift on the two variables passed to it. It is equivelent to dividing the left hand side by two a number of times which is specified by the right hand side.
- Right Shift (>>) - does a bitwise left shift on the two variables passed to it. It is equivelent to multiplying the left hand side by two a number of times which is specified by the right hand side.

Index Operator

The index operator allows for accessing information in string's and array's. There are three main forms that can be used.

- [] - This works on only the array type. The result of this is a void variable being added to the end of the array and being returned. This is the easiest way of adding variables to the end of the array.

  Example:
  ```
  array a;
  a[] = 1;
  a[] = 2;
  a[] = 3;
  ```

  The above example causes three items to be added to the array a with each one being set to a value. As arrays can contain any data type you need to assign something to the variable returned by [], otherwise you will end up with just a void.

- [expression] - This will index the array or string based upon the evaluated expression. If the expression results in a number - that variable will be returned from the array, or the character at that location within the string will be returned. If the expression is a string, it wont have any effect on a string, but will cause the hash value of that string to be taken out the array. If the hash location doesn't exist, the variable will be created (it will be placed on the end of the array as if [] had been used) and is returned. For historical reasons, the first element in the array is 0, the second 1 and so on.

  Example:

  ```
  array a;
  a[] = 1;
  a[] = 2;
  a["Hello World"] = 3;
  a[0] // This will get the first value within the array, in this exam-
  ple '1'
  a["Hello World"] // This will get the value pointed to by "Hello World", in this ex-
  ampe '3'
  a[2] // This will get the third value (created using the 4th line of the example)
  ```

- [expression..expression] (also refered to as the slice operator) - This is a range expression. With strings and arrays it allows you to take a slice of the variable. The range can be ascending - in which case the order in the variable is preserved, or descending in which case, the slice is made with the contents being reversed. It is

possible to leave out the upper or lower bound expression dictating that the operator should go to the end or from the beginning respectivly. If a negative number is given, it is taken to mean from the end of the variable.

Example:

```
string s = "Hello";
string t = s[-1..0]; // This will take a slice of the entire string and re-
verse it
string u = s[..2]; // This will take a slice of the first 3 characters in the string
```

Complex Operators

These operators are individual and slightly more complicated that the other operators.

- Object or namespace attribute (.) - To get an attribute or a method within a namespace or instantiated object you need to use '.'. It is not bound to the type of variable (ie. namespaces and objects act the same) like C.

  Example:

  ```
  Console.println( "Hello World" );
  ```

- Instantiate an object (**new**) - This operator is used to create an instance of a class (which can then be assigned to a an object variable. It is used as follows:

  ```
  new <class name>( <parameters> )
  ```

  <class name> The name of the class to be instantiated.

  <parameters> The arguments to be passed to the constructor of the class.

  It should be noted that multiple *object* variables can point to one object created using the new keyword. This is discussed later on within the *Classes* and *Objects* section.

  Example:

  ```
  object newObject = new SomeClass( "aString", 10 );
  newObject = new SomeOtherClass( "James", "Taylor" );

  (where SomeClass and SomeOtherClass have been defined elsewhere)
  ```

- Evaluate a string (**eval**) - This is a very powerful operator and can be very very useful. It also shows off the difference between a pre compiled language a scripting language. The eval operator allows you to on the fly compile and execute a script **and** get a return value. It is used like so:

  ```
  eval ( <some string with a script> )
  ```

  The string can be any value - but must be a valid script, if not an exception will be thrown.

  Example:

  ```
  eval( "Console.println(\"Hello World\");" );
  ```

  This script is the same as:

```
Console.println("Hello World");
```

To return a value, you just use the return keyword (mentioned within the function documentation in the next chapter). The code below will return '42', which will in then turn be assigned to the variable **value**.

```
number value = eval( "return 42;" );
```

This is of course a very simple example and doesn't show what a useful operator it is, but it does allow you to at runtime modify the behavior of code. It should also be noted that there are potential security risks involved with this operator and it should be considered carefully.

Later on in this manual, the operator **include** is discussed.

Regular Expressions

Ferite features regular expressions with a similar syntax to that of Perl. Currently there is only one operator concerning regular expressions within ferite (although this is likely to change).

Apply regular expression (=~)

This operator works be applying the regular expression defined on the right hand side to the string on the left hand side. Regular expressions can only be applied to strings. For more information regarding regular expressions see the section later on in the manual.

Statements

Statements are basically a collection of expressions followed by a ';'. A block of statements is defined as multiple statements between braces {}'s. It is as simple as that.

Example:

```
x = 1 + 2;
x++;
```

Control Structures

Ferite contains methods for changing the flow of a program, these are called control structures and are detailed below. The control structures can be placed within each other wihtout worry of mistake (ferite is clever like that!). Most of the structures follow the same rules as their counterparts in other languages although there are some differences.

if-then-else

This allows for the conditional execution of ferite scripts based upon the results of a test. The syntax is as follows:

Type one:

```
 if ( expression ) {
```

```
  statements if the expression is true
 }

Type two: (with an else block)

 if ( expression ) {
  statements if the expression is true
 } else {
  statements if the expression is false
 }
```

It is also not necessary to place braces around the statement block if it's only one statement.

When execution is happening the expression gets evaluated and then it's truth value is determined, if it's true then the first block is executed. If an else block exists then it will be executed if the expression evaluates to false.

Example:

```
if ( a < b )
 Console.println( "A is less than B" );

if ( b > c ) {
 Console.println( "B is greater than C" );
 Console.println( "This could be fun." );
} else {
 Console.println( "It's all good." );
}
```

while Loop

This construct provides a method of looping, and is used as follows:

```
while ( expression ) {
 statements if the expression is true
}
```

The body of the while construct will only be executed while the expression evaluates to true. The expression is evaluated upon every iteration.

```
number n = 0;
while ( n < 10 ) {
 Console.println( "$n" );
}
```

The above code will loop round while n is less than 10. On each iteration of the loop the value of n will be printed out.

for Loop

This construct provides a more controlled method of looping and is also ferite's most complicated loop. It's syntax is as follows:

```
for ( initiator; test; increment ) {
 statements if the expression is true
}
```

The initiator expression is executed unconditionally at the beginning of the loop. The loop will continue to loop until the test expression evaluates to false, and the increment expression is evaluated at the end of each loop.

As with C, each of the expressions may be empty, this will cause them to evaluate to true (therefore causing the loop to continue forever if there is no test expression).

Example:

```
number i = 0;
for ( i = 0; i < 10; i++ )
 Console.println( "I equals " + i ); // print out the value of i
```

foreach Loop

This construct provides a powerfull mechanism to itterate over data structures in a loop form. It can handle strings, arrays and objects. There are two forms, the first can be used on all three data types and the second is to itterate over an array's hash.

First Version

```
foreach( value, data ){
        statements to execute
}
```

This is used to itterate over the data values. *value* is the variable that the current value should be put in (**NOTE:** the variable must of the correct type otherwise an exception will be thrown). *data* is the item to itterate on. For a string, foreach set *value* to be a string containing each character within *data*. For an array, foreach will set *value* to be the same as each value within the array. When handling objects, foreach will call a method called *next* on the object. foreach will keep looping setting *value* as the return value from the method call, unless the return value is *void* at which point the loop will stop.

Second Version

```
foreach( key, value, data ){
        statements to execute
}
```

This version only works on arrays with hash values. It will itterate through each of the keys within the array's hash, setting *key* with the value of the hash key and *value* to the value at that hash key.

Examples:

```
array a = [ "Hello World", "From Chris" ];
string value = "";

foreach( value, a ){
  Console.println( "Value = $value" );
}
```

This will print out:

```
Value = Hello World
Value = From Chris
```

The second form works in much the same way. It should be noted that if the value for the *value* part of the foreach loop is of type void, it will be reset to void on every iteration. This allows you to go through an array with different types without throwing an exception.

do .. while Loop

The *do .. while* loop is a variation of the *while* loop, the one difference being that it guarantees at least one execution of it's body. It will only then complete looping until the expression evaluates to false. It's syntax:

```
do {
 statements if the expression is true
} while ( expression )
```

iferr-fix-else

This control structure provides the exception handling to within ferite. It is similar in a way to the **try-catch-finally** structure within Java but with one main difference. Within Java the finally block is always executed regardless of whether or not an exception occurs, in ferite the else block only gets executed if **no** exception occurs. The fix block is called when an exception occurs. This control structure is used as follows:

```
iferr {
 statements
} fix {
 statements to clean up incase of an exception
} else {
 statements if no exception has occurred
}
```

It is possible to nest iferr-fix-else blocks. When an exception does occur a global variable called err is instantiated. This has two attributes, string str and number num - these provide information on the error that occurred. Exceptions are propagated up through the system until a handler is found or the program has a forced termination.

switch statement

This allows you to write blocks of code that are executed when an expression holds a certain value. This is rougly equivelent to doing a number of successive **if** blocks. But it's cleaner and tidier, it takes the form of:

```
switch ( expression ) {
 case expression:
  ... code ...
 ... more case blocks ...
 default:
  ... code ...
}
```

When the switch statement is evaluated the expression at the top is executed and it's value saved. **case** blocks are evaluated to see whether the result of their expression matches that of the first expression. If it does - the code is executed until the end of the switch statement (including other case blocks). To only execute the one block you

have to use **break**, this will cause the execution to jump to the end of the switch statement. If you use **continue**, the first expression will be re-evaluated effectively causing the switch statement to start again. If there are no matches the the **default** block will be executed. This may sound quite complicated and is probably best described using an example:

```
switch( input ) {
 case 0:
  Console.println( "case 0" );
 case 1:
  Console.println( "case 1" );
  break;
 case 2:
  input++;
  Console.println( "case 2" );
  continue;
 case "3":
  Console.println( "case 3" );
  break;
 default:
  Console.println( "default" );
}

When input = 0, the following will be output:

case 0
case 1

(Because there is no 'break' the execution keeps going into
 the next case block)

When input = 1, the following will be output:

case 1

(Break causes execution to leave the switch block)

When input = 2, the following will be output:

case 2
default

(You would expect 'case 3' to be output - but the catch for
 that is a string with the character 3 in it)

When input = "3", the following will be output:

case 3

(Break causes execution to leave the switch block)

When input is anything else:

default
```

It is very important to note that you can use any valid expression within the case expression, it can even have different types than the first switch expression, there wont be any exceptions thrown. This makes switch a very powerfull construct. It is also not necessary to supply a **default** block if you do not wish to have one.

break

break will end the current *for, while, do .. while, switch or foreach* loop it is executed in.

continue

continue will cause execution flow to jump to the beginning to the current *for, while, do .. while, switch or foreach* loop it is executed in.

Functions

Functions are made up of variable declarations and statements [as described previously]. Each statement is terminated by means of a ; - as mentioned before. Functions are declared as follows:

```
function function_name( parameter declarations ){
 variable declarations
 statements
}
```

- *function_name* -- This is the name of the function to be called e.g. Print, Open.
- *parameter declarations* -- This is the signature of the arguments that can be passed to the function, and these are of the following form: <type> <name> (a comma seperated list)
- *variable declarations* -- See section *Variables*
- *statements* -- See section *Statements*

Example:

```
/*
  This function will add the string "foo" onto the end of the string it has been given
  return it.
*/
function foo( string bar ) {
 bar += "foo";
 return bar;
}
```

Functions provide an easy way of grouping statements together to perform a task. It must be noted that all variables must be declared **before** any other code - it is not possible to declare variables within the other statements - it will cause a compile time error.

Variable Argument Functions

Functions can take a varaible number of arguments by placing a **...** at the end of the argument list. An array can be obtained with **all** the variables passed to the function by using the function call **getArgs()**.

Example:

The following program listing shows how to access the array and make use of it.

```
uses "array", "console";

function test( string fmt, ... ){
   number i = 0;
```

```
   array fncArgs = getArgs();

   Console.println( "test() called with ${ Array.size(fncArgs) } args" );
   Console.println( fmt );

   for( i = 0; i < Array.size(fncArgs); i++ ){
   Console.println( "Arg[$i]: ${ fncArgs[i] }" );
   }
}

test( "nice" );
test( "nice", "two", "pretty" );
```

Returning A Value

If there is not an explicit return statement then the function will return a void variable. To return a variable it is as simple as using the **return** keyword:

Example:

```
return someValue * 10;
return 0;
return "Hello World";
```

Function Overloading

There are times when you wish to have the same operation applied to different data types, for example, an print method where you wish to handle various different types and/or number of arguments. Ferite provides a function overloading mechanism to combat this which allows you to write a set of functions all with the same name but with different parameters. When the program is run - ferite will automatically choose the best function for the job.

```
uses "console";

function print( number n ){
 Console.println( "Number: $n" );
}

function print( string s ){
 Console.println( "String: $s" );
}

print( 10 );
print( "Hello World" );
```

The above code declares two functions with the name `print`. If the script is run the following output would occur:

```
Number: 10
String: Hello World
```

One Line Functions

One last and final thing that should be noted about functions is that if you only have a one line function - you do not need to include the braces around the code. This is used to make things cleaner and tidier. For example, the above script written using this feature would looke like this [note the lack of braces]:

```
uses "console";

function print( number n )
```

```
 Console.println( "Number: $n" );

function print( string s )
 Console.println( "String: $s" );

print( 10 );
print( "Hello World" );
```

Classes and Objects (and references)

A class is a collection of data and methods, where the methods are intended to work on the data. Classes are templates for variables they describe how complex data types work. To use a class it is necessary to create and instance of a class (see the new keyword) and assign it to an object variable. The syntax of a class is as follows:

```
class <name of class> {
    <variable and functions declarations>
}
```

An example class:

```
class foo {
 string bar;

 function foo( string str ){ // constructor
  self.bar = str; // make bar equal to passed string
 }

 function printBar(){
  Console.println( self.bar ); // print bar
 }
}
```

This defines a class with a string and two methods. To create an instance of this class you would do the following:

```
object someObj = new foo( "Hello World" );
someObj.printBar(); // will output Hello World
```

To reference variables and methods from within the class it is necessary to prefix the variable with **self.** or simply a **.**. This merely tells ferite that you want the variable within the class (it is not necessary to do this for locally scoped variables within methods). The **self.** is the long notation, but you can leave the **self** bit out and ferite will work it out :-).

Classes can have constructors, these are within the form of a method with the same name as the class. The constructor will be called implicitly when an instance is created. It is suggested that you place you initialisation code here. (It should be noted that you can use all variables within a class in the constructor as they have already been created for you). An example of a constructor can been see above - method is called **foo**.

It is possible to extend classes by using inheritance, this is done using the **extends** keyword. There is no multiple inheritance and an example of inheritance is:

```
class Person {
 string name;
 number age;
```

```
 function Person( string n, number a ){ // constructor
  self.name = n;
  self.age = a;
 }
}

class Employee extends Person {
 number salary;

 function Employee( string n, number a, number sal ){ // constructor
  super.Person( n, a );
  self.salary = sal;
  self.name += " - Employee"; // change the name
 }
}
```

These classes are not usable in any fashion but merely highlight inheritance.

A couple of important facts need to be noticed:

1. When inheritance occurs and then an instance is made, the constructor of the super (parent) class is **not** called automatically. It is up to the subclass (child) to explicitly call it. This can either be done by doing super.NameOfConstructor() or super(). Either way is the same in the long run.
2. To get the object as a cast of the super class the super keyword is used. E.g. `super.someFunction()` will call the function `someFunction` as if it was coming from an object created from the parent class.
3. Function overloading works with objects and classes aswell.

Currently there is no support for private or protected members of a class. This is a planned addition in the future. As a general rule of thumb, it is considered bad practice to directly access an objects variables. If they need to be read or modified - methods should be supplied and used.

When an instance of a class is created it is added to the garbage collector so that is can keep an eye on it. Then a reference is returned - this is merely a pointer to the object within the system, this means that if you then assign one object variable to another - they both point to the same object.

Example:

```
class foo {
 string name;

 function foo( string n ){ //constructor
  self.name = n;
 }
}

object objA, objB;
objA = new foo("boris" );
objB = objA; // they both now point to the same object foo with name="boris"
```

Due to this and combined with the garbage collector, objects will automatically get cleaned up and removed from the system when they are not referenced anymore. It should also be noted that the garbage collector does work based on reference counting and is therefore susceptible to circular references. There is no guarantee as to when an object will be destroyed.

Static Members

Ferite supports static members within classes. These act the same as within Java and allow to have functions and variables on a per class basis rather than a per object basis. Static functions and variables are to classes what functions and variables are to namespaces. To reference them you simply use the class name then the member name, e.g. for a static function `bar` in the class `Foo` you would call it by doing `Foo.bar`. If you try and access a static member by using an object (rather than the class) an exception will be thrown.

This is used as follows (both function and variable shown):

```
static function_name( parameter declarations ){
 variable declarations
 statements
}

static number nameofvar;
```

Modifying Existing Classes

ferite has a number of features that allows you to modify existing classes. Why is this useful? Well, say you have a class that is used all over the place, lets say **File**, and you wish to debug a method, or reimplement a method to work around a bug, or even just add a method. It transparently allows you to shape an existing class to be how you want it to be.

To do this you use two keywords: modifies and rename. Here is an example:

```
class modifies File {

   rename readln oldReadln;
   rename open oldOpen;

   function readln(){
   return self.oldReadln(1024);
   }

   function open( string file, string mode ){
   self.oldOpen( file, mode, "" );
   }

   function toString(){
   string str = "";

   while( !self.eof() )
  str += self.readln();
   return str;
   }
}
```

To modify a class you use the syntax '**class modifies** nameOfClass', this will tell ferite that the target is that class, the class must exist otherwise you will get a compile error. Once this is done you can add new methods and variables, and twiddle with the existing ones.

rename - this takes two labels, the current name and the new name and renames it. The advantage of this approach is that you can drop in a replacement method and still call the old method within your new method. The above example re-implements the **.readln** method within the **File** class such that it doesn't require he passing of a number of bytes to read.

The above example also adds a new **toString()** which will return the file's contents in a string.

WARNING: you can potentially cause alot of confusion using these, but they are very useful for debuging and various other uses. You can modify any class.

Namespaces

Namespaces are defined in the following manner:

```
namespace name of namespace {
 variable, namespace, class, and function declarations
}
```

All languages have namespaces, Java's are governed by static members to classes but C has no means of explicitly defining them. The are a means of grouping likewise data and functions into a group such that there doesn't exist conflicts with names (hence namespace). Functions, Variables, Classes and even other namespaces can be defined within a namespace.

Example:

A standard error message for two different systems within the same program - Text and Graphical. In C it would have to be done like so:

```
void text_print_error_message( char *msg );
void graphical_print_error_message( char *msg );
```

Whereas in ferite you would use namespaces:

```
namespace Text {
 function printErrorMessage( string msg ){}
 // other stuff to do with text
}

namespace Graphical {
 function printErrorMessage( string msg ){}
 // other stuff to do with graphical
}
```

The ferite method is cleaner as it is more obvious what belongs to what, and also allows the programmer using the Graphical and the Text API's to know that if they want to show an error message they merely call the printErrorMessage() function in which ever namespace they want. If you combine this with the ability to assign a namespace reference to a void variable - very powerfull abilities become apparent. Using the above code:

```
void outputMechanism;

if( wantGUI )
 outputMechanism = Graphical;
else
 outputMechanism = Text;

outputMechanism.printErrorMessage( "We have an Error" );
```

They promote clean and precise code. When a function is defined within a namespace it has to reference stuff within the namespace as code out side does, e.g. `namespace.resource`.

It is important to note that you can't use the shortcut method to access functions and variables that is used in objects e.g. `.someVariable`. You must use the full name to access anything within namespaces.

Modifying Existing Namespaces

There is also an alternative syntax for namespaces allowing you to extend an already existing namespace or create a new one if it doesn't already exist. This is done like so:

```
namespace modifies name of namespace {
 variable, namespace, class and function declarations
}
```

When this modifies the namespace it places all items within it in the block in the namespace mentioned. Eg:

```
namespace foo {
 number i;
}

namespace modifies foo {
 number j;
}
```

In the above example the namespace **foo** has a number i and a number j. The main reason for this syntax was to allow module writers to easily intermingle native and script code within the namespace. There is also times when placing something in another namespace makes more sense. e.g. Placing a custom written network protocol within a Network namespace.

It is possible to use the same **delete** and **rename** functions as in classes, in namespaces.

Regular Expressions

Regular expressions provide a very powerful method of matching, modifying and extracting information from strings. Using special syntax, code that would usually require line after line of special matching code can be summarised within a one line regular expression (from here on in referred to as a regex). They can either be found within the language, e.g. Perl or ferite, or as an add in library, e.g. Python, php and C. ferite's regex's are providied by means of PCRE (Perl Compatible Regular Expressions, a C library that can be found at http://www.pcre.org) and as a result are almost identical in operation to Perl's. Regex's look like this:

Example:

```
s/1(2)3/456/
```

This one will match all occurrences of the string "123" and swap them with "456"

```
s/W(or(l))d/Ch\1ris\2/
```

This is more complicated and will match occurrences of "World" and swap them with "Chorlrisl". The reason being is due to back ticks which are discussed soon.

There are three types of regular expression support and that is match, swap and capture. They are used as follows:

```
m/expression to match/
s/expression to match/string to replace it with/
c/expression to match/comma,seperated,list,of,variables/
```

To match an **m** is used, to swap an **s** is used. It is possible to capture strings within the regular expression using the same method as in Perl i.e. By using brackets. The captured strings upon each match are placed into **r<bracket number>** - this is equivalent to the $1, $2, ... $n strings in Perl. The maximum number of captured strings is currently 99, and example of captured strings is in the above expressions, i.e. (2) would cause "2" to be place within r1, in the second expression (or(l)) would cause "orl" to be placed within r1 and "l" to be placed within r2.

Options

There are a number of options that can be used to modify the method that the regular expression's execution and processing, these are:

- **x** - This option allows the regular expression to be multi line, and also allows comments using the # character. This is useful for long regular expressions where it is important to remember what each individual part performs.
- **s** - This allows the **. (dot)** matching character to match newlines (\n's).
- **m** - This gets the **^** and **$** meta characters to match at newlines within the source string.

  Example:

  ```
  string foo = "Hello\nWorld\nFrom\nChris";
  foo =~ s/^(.*)$/Foo/sm;

  The above regex will be changed to "Foo\nFoo\nFoo\nFoo"
  ```

- **i** - This causes the regex engine to match cases without looking at the case of characters being processed.
- **e** - This causes the replace string to be evaluated as if it had been passed to **eval()**. The return value from the script will be used as the replacement text - the return needs to be a string.

  Example:

  ```
  string foo = "Hello World";
  foo =~ s/Hello/return "Goobye";/ge
  Console.println( foo );

  foo will now equal "Goodbye World"
  ```

- **g** - This forces all matches along a line to be matched. Normally it is only the first occurance that is matched.
- **o** - This causes the regular expression to be compiled at compile time rather than runtime. This is useful for regular expressions that dont change but are used alot within a script.

- **A** - The pattern will only match if it matches at the beginning of the string being searched.
- **D** - This option allows the user to have only the **$** tie to the end of a line when it is at the end of the regular expression.

Backticks

Backticks are used within the swap mode of the regular expressions. It allows you to used captured strings within string that should replace the matched expression. There are used within the second example above. They are used as follows: a '\' (back slash) followed by the number that you want to use.

This is only a brief insight into regular expressions, and a suggested read is "Mastering Regular Expressions" by Jeffrey E. F. Friedl (published by O'Reilly), and that will tell you everything you need to know about regular expressions. :-) It is also suggested that the libpcre documentation is worth reading on http://www.pcre.org.

Uses and Include

Both the **uses** and the **include()** instructions tell ferite to include another script within the current one. The main difference is that **uses** is a compile time directive and **include()** is a runtime directive.

It is **VERY** important to note a specific behavior with these two language constructs. When a script is imported - the script importing it obtains: global variables, classes, functions, namespaces, but **NOT** and I repeat **NOT** the anonymous start function. By this I mean the code that gets run when running the script. The logic behind this apparent maddness is that it allows the anonymous start function to be used to write test cases or examples that go **with** the imported script. This means that if you modify something within the imported script (say it's from a library of scripts) - testing it is just a case of running it through ferite. Or if someone wants an example on how to use it's features - just look at the script being imported. This reduces the number of files that need to be distributed in libraries of scripts and allow distinct test cases/examples not to be lost (and stay current to the API they are against).

Uses

The uses keyword is used to import API from other external modules and scripts. The uses keyword is a compile time directive and provides the method for building up the environment. It can either pull in an external module, or compile in another script. The syntax is as follows:

```
uses "name of module or script file", ...;
```

"name of module or script file"

The name must be in quotes. When ferite gets this call it will do the following: if there is no extension it will try loading a script in the system's library paths trying the extensions '.fe', '.fec' and '.feh'. The paths for the native and scripts are defined by the parent application. If an extension is given, ferite will check to see if it equals **.lib**, if it does it will load the correct native module, eg. `uses "array.lib";` will cause ferite to load `array.so` under unices and `array.dll` under windows. This gives a platform independant method to tell ferite to load a native library. This is the method used to load a native module. If it doesn't equal **.lib**, ferite will treat it as a script and load it.

In the case of the ferite command line application, `$prefix/lib/ferite` is searched for scripts plus any directories that are added on the command line by using the -I flag. Native modules are placed in `$prefix/lib/ferite/$platform`, where $platform is of the form `os-cpu`.

If either the script or the module can't be found the compilation of the script will cease with an error. It is suggested that these are placed at the top of the script (although this is not a requirement).

include()

include() operates the same way as **uses**, except that it can currently only import other scripts. Once the call has been made - the facilities provided by the imported script can be used. It should be noted that the return value from the include() call is the return of the main method when the script is loaded. This allows items to be passed to the parent script.

```
include ( "someScript.fe" );
```

Notes

1. http://www.pcre.org
2. http://www.pcre.org

Chapter 3. Application Interface

Where To Find It

Please see the document regarding embedding ferite in the developer section of the ferite website.

Chapter 4. Known Issues

The main known issue with version one of the ferite engine is that almost all bugs are found at runtime - there is only checking for correct data types at runtime, this is because of the way the engine operates and will be corrected in version two of the engine.