ZetScript 2.0.0

Jordi Espada

Version 2.0, January 30,2023

Table of Contents

Acknowledgment

I will like to give my sincerely acknowledgment to my girlfriend Maria Portal who thanks to her I got inspired to do this interesting project and also for the time I needed she give me the chance to finish it.

Also the authors of many script engines they developed because, thanks to them, I learned how current scripts works and what actually what people is expecting for a script engine. This was like a orientation for the development of this project.

Finally to StackOverflow community thanks of this I was able to solve some technical issues. I hope this community lives eternally whole-hearted!

Chapter 1. 1. Introduction

ZetScript is a script language with a syntax inspired in ECMAScript or Javascript but also it brings a easy way to bind parts of your C++ code. ZetScript provides a virtual machine so the execution is quite fast. Because ZetScript syntax is almost less or more equal to Javascript you can edit the code with any editor that supports Javascript syntax notation.

1.1. Compile

ZetScript has to be configured through (cmake,http://cmake.org) with the following compile toolchains,

sing gnu toolchain, mingw, clang and Visual Studio through cmake

To install ZetScript ,first download last source code from https://zetscript.org Then it has to compile the project with the following steps,

1. First we have to configure the project using cmake application.

   ```
   cmake CMakeLists.txt
   ```

2. Second we have to compile and install the project. Using GNU tool chain is done through this command,

   ```
   sudo [1] make install
   ```

1.2. Hello World

Once ZetScript is installed we present a quick sample of "HelloWorld" application,

1. Create a filename named helloworld.zs and type the following sentence

   ```
   Console::outln("Hello world!");
   ```

2. Save the file and do the following at command line,

   ```
   zs helloworld.zs
   ```

   You should see the "Hello world" message at the command line.

1.3. Compile

[1] sudo is required when the project is compiled in linux.

Chapter 2. The language

2.1. Statments

ZetScript Language is based on a set of statements formed by values, operators, expressions, keywords, declarations, expressions, conditionals, integrators and functions. All statements are separated by semicolons at the end.

Example of four statements,

```
var op1,op2,res;
op1 = 5;
op2 = 6;
res = 5+6;
```

The last example can be executed within one line,

```
var op1,op2,res; op1=5; op2=6; res = 5+6;
```

2.2. Comments

ZetScript support block and line comments.

2.2.1. Block comment

```
/*
this is a block comment
*/
```

2.2.2. Line comment

```
// This is a line comment
```

2.3. Literals

ZetScript supports the following literals,

- Boolean
- Integer
- Float
- String

2.3.1. Boolean

Boolean literals is represented as true or false.

Example,

```
true // true value
false // false value
```

2.3.2. Integer

Integer literals are represented as integer with range from -(2b-1) to 2b-1-1 where b=32 or 64 it depending whether ZetScript is compiled for 32bits or 64bits. The integer value can be represented as decimal value, hexadecimal,binary or char format.

Example,

```
10; // decimal value
0x1a; // hexadecimal value
11010b; // 0x1a as binary value
'b'; // char value
```

Integer literlas supports the following pre operators,

Operator	**Expression**	**Description**	**Example**
Negate	`-integer`	Negated value	`-10`
Bitwise complement	`~integer`	Invert all bits	`~011010b // (-27)`

2.3.3. Float

Float literals are represented as IEEE-754 floating point numbers in 32-bit or 64 bit it depending whether ZetScript is compiled for 32bits or 64bits. Float can be represented in decimal form or scientific notation form.

Example,

```
1.2 // decimal form
2.0e-2 // cientific notation form
```

Float literals supports the following pre operators,

Operator	Expression	Description	Example
Negate	`-float`	Negated value	`-10.5`

2.3.4. String

String literal is represented as string within quotes (").

Example,

```
"this is a string"
```

2.4. Variables and constants

2.4.1. Variable

A variable is a kind of element that can hold a variable. Because ZetScript variables are *dynamically typed*, it can hold different type of values on its lifetime. A variable is declared as **var** and, by default, is initialized as **undefined** value (see 2.5.1).

```
var i; // Declared variable 'i' with no inicialization (undefined value).
```

A variable can be initialized with a proper type value through '=' operator. The follow example initializes a variable with integer value (see section 2.2),

```
var i=10; // Declared variable 'i' and initialized as integer value.
```

2.4.2. Constant

A constant is a kind of element where its value is immutable on its in lifetime. A constant is declared as **const** and is mandatory to be initialized.

```
const NUM_ITEMS=10; // constant 'NUM_ITEMS' with value 10.
```

2.4.3. Scope

In ZetScript we have two types of basic scopes.

- Global
- Local

A ZetScript makes easy the concept of global local scope. Basically, variables declared on the main script are global and the others declared within a [block [1]], are local.

Example,

```
// Declares *i* variable as global
var i;

// block starts here
{
    // Declare *j* variable as local (you can also access to i).
    var j;
}
// block ends here, so *j* variable doesn't exist anymore
```

2.5. Data types

ZetScript supports the following builtin data types,

- Undefined
- Null
- Integer
- Number
- Boolean
- String
- Vector
- Object
- Function

2.5.1. Undefined

Undefined data type it defines a non initialized variable. A variable is instanced as *Undefined* once a its assings **undefined** value,

```
var a; // 'a' is undefined as default
var b=undefined; // assigns undefined value
```

2.5.2. Null

Null data type it defines an empty or not valid variable. A variable is instanced as *Null* once a its assings **null** value,

```
var i=null;
```

2.5.3. Integer

Integer data type it defines a integer variable with range from -(2b-1) to 2b-1-1 where b=32 or 64 it depending whether ZetScript is compiled for 32bits or 64bits. A variable is initialized as *Integer* once it assigns decimal, hexadecimal,binary or char value.

Example,

```
var a=10; // decimal value
var b=0x1a; // hexadecimal value
var c=01001b; // binary value
var d='b'; //  char value
```

Operators

Integer data operators are applied on integer variables before or after its read. Integer operators are the following,

Operator	Expression	Description	Result

Negate	`-variable`	Negates its value	` var i=10;`
			` var j=-i; // j=-10`

Bitwise complement	`~variable`	Invert all bits	` var i=00000011b;`
			` var j=~i; // j=11111100b`
			`or -4 decimal`

PreIncrement	`++variable`	Increments FIRST and THEN evaluates	` var i=0;`
			` var j=++i; // j=1, i =1`

PostIncrement	`variable++`	Evaluates FIRST and THEN increments	`var i=0;`
			`var j=i++; // j=0; i=1`

Predecrement	`--variable`	Decrements FIRST and THEN evaluates	`var i=0;`
			`var j=--i; // j=-1; i=-1`

Postdecrement	`variable--`	Evaluates FIRST and THEN decrements	`var i=0;`
			`var j=i--;//j=0;i=-1;`

Static functions

Integer::parse(_value)

Description

Converts input value as Integer

Parameters

- *_value* : String or Float literal or datatype to parse

Returns

Integer as parsed value

Example

```
Console::outln(Integer::parse("10"))
Console::outln(Integer::parse(15.5))
```

Output:

```
10
15
```

2.5.4. Float

Float type it defines a float variable represented as IEEE-754 floating point numbers in 32-bit or 64 bit it depending whether ZetScript is compiled for 32bits or 64bits. A variable is initialized as *float* once it assigns a decimal or scientific value notation forms.

Example,

```
var a=1.2 // decimal form value
var b=2.0e-2 // cientific notation form value
```

Operators

Float operators are applied on float variables before or after its read. Float operators are the following,

Operator	Expression	Description	Result

Negate	`-variable`	Evaluates negated variable	`var i=10.5;`
			`var j=-i; // j=-10.5`

PreIncrement	`++variable`	Increments FIRST and THEN evaluates	`var i=0.5;`
			`var j=++i; // j=1.5, i`
			`=1.5`

PostIncrement	`variable++`	Evaluates FIRST and THEN increments	`var i=0.5;` `var j=i++; // j=0.5; i=1.5`
Predecrement	`--variable`	Decrements FIRST and THEN evaluates	`var i=0.5;` `var j=--i; // j=-0.5; i=-0.5`
Postdecrement	`variable--`	Evaluates FIRST and THEN decrements	`var i=0.5;` `var j=i--;//j=0.5;i=-0.5;`

Static functions

Float::parse(_value)

Converts input as Float value

Parameters

- *_value* : Input value

Returns

Float as parsed value

Example

```
Console::outln(Float::parse("10.5"))
Console::outln(Float::parse(15))
```

Output:

```
10.500000
15.000000
```

2.5.5. Boolean

Boolean type it defines a boolean variable represented as *true* or *false*. A variable is initialized as *Boolean* once it assigns a boolean value.

Example,

```
var b=false;
```

2.5.6. String

String type it defines a string type represented as a sequence of chars. A variable is initialized as *String* once it assigns a string value.

Example

```
var s="this is a string";
```

Properties

String::length

Returns the length of the string

Parameters

None

Returns

The length of the string

Example

```
var s="hello world";
Console::outln("The length of '{0}' is '{1}'",s,s.length)
```

Output:

```
The length of 'hello world' is '11'
```

Static functions

String::format(_string,..._args)

Builds a string placing formated values in {} as string with a serie of variable args. The possible formats is described below,

`Format`	description
`{n}`	It replaces {*n*} by the argument *n*
`{n:dm}`	It replaces the argument *n* padding *m* 0s on its left
`{n,m}`	It replaces argument *n* padding *m* spaces on its left

Parameters

- *_string* : string value with

- *_args* : variable args

Returns

Formated string

Example

```
Console::outln(String::format("1st arg {0}",1))
Console::outln(String::format("1st,2nd as {0} and {1}",1,2))
Console::outln(String::format("1st,2nd same {0} and {0}",1))
Console::outln(String::format("Score:{0:d1}",1))
Console::outln(String::format("Score:{0:d4}",1))
Console::outln(String::format("Score:{0,1}",1))
Console::outln(String::format("One space:{0,4}",1))
```

Output:

```
1st arg 1
1st,2nd as 1 and 2
1st,2nd same 1 and 1
Score:1
Score:0001
Score:1
One space:   1
```

Member functions

***String::insertAt(_pos,_value)**

Inserts a character or string into the string at index

Example

```
var s="hell wd";

// example insert character usign String::insertAt
Console::outln(" Original string before insert character => '{0}'",s);
s.insertAt(4,'o')
Console::outln(" After insert character 'o' at position 4 => '{0}'",s)

Console::outln()

// example insert string usign String::insertAt
Console::outln(" Original string before insert string => '{0}'",s);
s.insertAt(7,"orl")
Console::outln(" After insert string \"orl\" at position 7 => '{0}'",s)
```

Output:

```
 Original string before insert character => 'hell wd'
 After insert character 'o' at position 4 => 'hello wd'

 Original string before insert string => 'hello wd'
 After insert string "orl" at position 7 => 'hello world'
```

String::eraseAt(_pos)

String::eraseAt removes the character right before the character indicated by _pos

Parameters:

None

Return

None

Example

```
var s="helilo world";

Console::outln(" Original string before erase character => '{0}'",s);
s.eraseAt(3)
Console::outln(" After erase character at position 3 => '{0}'",s)
```

Output:

```
 Original string before erase character => 'helilo world'
 After erase character at position 3 => 'hello world'
```

String::toUpperCase()

Returns the calling string value converted to uppercase

Parameters

None

Return

It returns the String, converted to uppercase

Example

```
Console::outln("\"Hello World\".toLowerCase() => \"{0}\"","Hello
World".toLowerCase());
```

Output:

```
"Hello World".toLowerCase() => "hello world"
```

String::toLowerCase()

Returns the calling string value converted to lowercase

Parameters

None

Return

It returns the String, converted to lowercase

Example

```
Console::outln("\"Hello World\".toLowerCase() => \"{0}\"","Hello
World".toLowerCase());
```

Output:

```
"Hello World".toLowerCase() => "hello world"
```

String::clear()

Parameters

Return

Example

```

```

String::replace()

Parameters

Return

Example

```

```

String::split()

Parameters

Return

Example

```
```

String::contains()

Parameters

Return

Example

```
```

String::indexOf()

Parameters

Return

Example

```
```

String::startsWith(_value)

Checks whether a string starts with the specified string passed by *_value*

Parameters

- *_value* : A string representing the value to check for

Return

Returns a Boolean value:

- true : if the string starts with the specified string passed by *_value*
- false: if the string does not starts with the specified string passed by *_value*

Example

```
var s = "Hello";
Console::outln("s.startsWith(\"Hel\") => {0}", s.startsWith("Hel"));
Console::outln("s.startsWith(\"llo\") => {0}",s.startsWith("llo"));
```

```
Console::outln("s.startsWith(\"o\") => {0}",s.startsWith("o"));
```

Output:

```
s.startsWith("Hel") => true
s.startsWith("llo") => false
s.startsWith("o") => false
```

String::endsWith(_value)

Checks whether a string ends with the specified string passed by *_value*

Parameters

- *_value* : A string representing the value to check for

Return

Returns a Boolean value:

- true : if the string ends with the specified string passed by *_value*
- false: if the string does not ends with the specified string passed by *_value*

Example

```
var s = "Hello";
Console::outln("s.endsWith(\"Hel\") => {0}", s.endsWith("Hel"));
Console::outln("s.endsWith(\"llo\") => {0}",s.endsWith("llo"));
Console::outln("s.endsWith(\"o\") => {0}",s.endsWith("o"));
```

Output:

```
s.endsWith("Hel") => false
s.endsWith("llo") => true
s.endsWith("o") => true
```

String::substring(_start_index, _end_index=-1)

Returns the part of the string from the start index and, optionally, a second parameter to specify the end index. If the end index is not provided it sets as -1 as default that means that will copy till string end

Parameters

Return

Example

```
var s="hello world";
Console::outln("s.substring(0) => {0} ",s.substring(0))
Console::outln("s.substring(3) => {0} ",s.substring(3))
Console::outln("s.substring(2,3) => {0} ",s.substring(2,3))
Console::outln("s.substring(3,-2) => {0} ",s.substring(3,-2))
```

Ouput:

```
s.substring(0) => hello world
s.substring(3) => lo world
s.substring(2,3) => ll
s.substring(3,-2) => lo worl
```

String::append()

Parameters

Return

Example

```

```

2.6. Vector

Vector type it defines a container type that stores values in a unidimensional array. A variable is initialized as *Vector* once it assings a open/closed square brackets (i.e '[]'),

Example,

```
[]; // a vector
```

Variable instanced as *Vector* can also have initialized with a sequence of elements separated with coma (i.e ',') within square brackets,

Example,

```
[1,"string",true,2.0]; // A vector with elements
```

To access element vector is done through integer as a index.

Example,

```
var v=[1,"this is a string",true,2.0]; // variable 'v' has 4 elements where its access
exist in [0..3]
```

```
v[1]; // It access vector's second element (i.e "this is a string")
```

Properties

Vector::length

```

```

Static functions

Member functions

Vector::push(_value)

```

```

Vector::pop()

```

```

Vector::insertAt(_pos,_value)

```

```

Vector::eraseAt(_pos)

```

```

Vector::clear()

```

```

Vector::join(_vector)

```

```

2.6.4. Object type

Object type is a anonymous container type to store values through attributes. A variable is initialized as *Object* once it assings a open/closed of curly brackets (i.e '{}'),

```
{}; // Object
```

Optionally we can init with some values with its attributes,

```
{
  i:1
 ,s:"this is a string"
 ,b:true
 ,f:2.0
};
```

To acces to its elements is done through the variable name followed by '.' and attribute name or attribute name as string within brackets (i.e ["attribute_name"]),

```
var o={
  i:1
 ,s:"this is a string"
 ,b:true
 ,f:2.0
};

var o_i=o.i; // get value 'i' by '.'
var o_i=o["i"]; // get value 'i' by '[]'
```

Static functions

Object::clear()

```
```

Object::erase(_id)

```
```

Object::contains()

```
```

Object::append()

```
```

Object::concat()

```
```

Object::keys()

```
```

2.6.5. Function

Function type is a anonymous function object to be assigned. A variable is initialized as *Function* once it assings an anonymous function (see section X.XX)

Example,

```
var f=function(){
    Console::outln("Call from function object")
};

f(); // it prints "Call from function object"
```

2.7. Operations

ZetScript has the following type of expressions

- Arithmetic operations
- Relational operations
- Logical operations
- Bit operations

2.7.1. Arithmetic expressions

The following operators it does evaluates arithmetic expressions,

Operator	`Symbol`	Description	Example

Add	`+`	It performs a add operation between two integer or number values or concatenates strings with other values	`5+10; // = 15`
			`1.5+6; // = 7.5`
			`"string_"+1;// ="string_1"`

Subtract	`-`	It performs a sub operation between two integer or number values	`10-5; // = 5`
			`2.5-1;// = 1.5`

Multiply	`*`	It performs a multiplication between two integer or number values	`10*5; //= 50`
			`1.5*2;//= 3.0`

Divide	`/`	It performs a division between two integer or number values	`10/2; // = 5`
			`3/2.0 // = 1.5`

Modulus	`%`	It performs a division between two integer or number values	`3%2; //it results 1`
			`10%2.5; // it results`

2.7.2. Relational expressions

The following operators it does evaluates relational expressions,

Operator	**symbol**	**Description**	**Example**

Equal	`==`	Check whether two values are equal	` 10==10;// = true`
			`"hello"=="bye"; // =`
			`false`

Not equal	`!=`	Check whether two values are not equal	`10!=10; // = false`
			`"hello"!="bye"; // = true`

Less than	`<`	Checks whether first value is less than second value	`10<20; // = true`
			`20<10; // = false`

Greater than	`>`	Checks whether first value is greater than second value	`10>20; // = false`
			`20>10; // = true`

Less equal than	`⇐`	Checks whether first value is less equal than second value	`10<=10; //= true`
			`11<=10; // = false`

Greater equal than	`>=`	Checks whether first value is greater equal than second value	`10>=11; // = false`
			`11>=10; // = true`

Instance of	`instanceof`	Checks if a value is instance of a type.	`0 instanceof Integer; //=`
			`true`
			`"hello" instanceof`
			`Integer;//= false`

Note: You cannot mix different types for relational expressions. For example, doing a relational expression with boolean and integer values is incompatible.

2.7.3. Logic expressions

Logic expressions are the ones that combines operations through boolean values,

Operator	symbol	Description	Example

Logic And	`&&`	it performs an AND operation between two Boolean values	`true && true;// = true`
			`true&& false;// = false`

Logic Or	`||`	It performs an OR operation between two Boolean values	`true || false;// = true`
			`false || false;// = false`

Logic Not	`!`	Negates Boolean value	`!true; // = false`
			`!false; // = true`

2.7.4. Binary operations

Binary operations are the ones that combines bit operations through integer values,

Operator	symbol	Description	Example

Binary And	`&`	Performs binary AND operation between two integers	`0xa & 0x2; // = 0x2`
			`0xff & 0xf0; // = 0xf0`

Binary Or	`|`	Performs binary OR operation between two integers	`0xa | 0x5; // = 0xf`
			`0x1 | 0xe; // = 0xf`

Binary Xor	`^`	Performs binary XOR between two integers	`0xa ^ 0xa; // = 0x0`
			`0xa ^ 0x5; // = 0xf`

Binary shift left	`<<`	Performs binary shift left	`0x1 << 2; // = 0x4`
Binary shift right	`>>`	Performs binary shift right	`0xff >> 1; // = 0x7f`

2.7.5. Priority operations

Each operator it has priority of evaluation. ZetScript it has the following operator order priority,

*,/,%,!=,+,-,^,&,|,<<,>>,==,⇐,>=,>,<,||,&&

For example this expression,

```
2+4*5; // will result 22
```

You can change the evaluation priority usign parenthesis.

For example,

```
(2+4)*5; // will result 36
```

2.8. Conditionals

A conditional statement are used to perform different actions based on different conditions. In ZetScript we have the following conditional statement:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false.
- Use ternary condition to have a short if/else statement into single statement.
- Use switch to specify manu alternative blocks of code to be executed

2.8.1. if-else statement

If statement is used to specify a block of ZetScript code to be executed if a condition is 'true'.

Syntax,

```
if(condition){
    //Block of code to be executed if the condition is true
}
```

Example,

```
if(n < 10) {
    // do something if condition is true
}
```

It can use *Else* statement is used to specify a block of code to be executed if the condition is 'false'.

Syntax,

```
if(n < 10) {
    // do something if condition is true
}else{
    // do something if condition is false
}
```

2.8.2. if-else statement

Use the else statement to specify a block of code to be executed if the condition is 'false'.

Syntax,

```
if(n < 10) {
    // do something if condition is true
}else if(n < 20){
    // do something if condition is true
}else{
    // do something if none of above conditions are true
}
```

2.8.3. Ternary condition

Use ternary condition to have a short if/else statement into single statement. It performs expression if the condition is true or the second expression if the condition is 'false'.

Syntax,

```
result = (condition)?first expression:2nd expression;
```

Example,

```
var j = 0>1? 0:1; // j = 1
```

2.8.4. switch

Switch statement is used to select one of many blocks of code to be executed.

Syntax,

```
switch(expression) {
    case value_0:
        code block
        break;
    case value_1:
        code block
        break;
        ...
        case value_n
        default:
        code block
        break;
}
```

Example,

```
switch (n) {
    case 0:
        // do something if n==0
        break;
    case 1:
        // do something if n==1
        break;
    default:
        // do something if n!=0 && n!=1
        break;
}
```

Switch can have common code blocks in different conditions

Example,

```
switch (n) {
    case 0:
    case 1:
        // do something if n==0 or n==1
        break;
    case 2:
    case 3:
        // do something if n==2 or n==3
    break;
    default:
        // do something if n!=0 && n!=1 && n!=2 && n!=3
        break;
}
```

2.9. Loops

ZetScript supports the following loop types,

- While Loop
- For Loop

2.9.1. while

The while loop loops through a block of code as long as a specified condition is true.

Syntax,

```
while(condition){
    // code block to be executed
}
```

Example,

```
var i = 0;
while (i < 5){
    // do something until i==5
    i++;
}
```

2.9.2. do-while

do-while loop is always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Syntax,

```
do{
    // do-while body
} while (condition);
```

Example,

```
var i = 0;
do {
    // do something until i==5
    i++;
} while (i < 5);
```

2.9.3. for

The for loop is often the tool you will use when you want to create a loop.

Syntax,

```
for(stament1;statment2;statment3){
    // code block to be executed
}
```

- Statement 1 is executed before the loop (the code block) starts. Normally you will use statement 1 to initialize the variable used in the loop (for example var i = 0).
- Statement 2 defines the condition for running the loop.
- Statement 3 is executed each time after the code block has been executed.

Example,

```
for(var i=0; i < 5; i++) {
    print("The number is "+i);
}
```

2.10. Functions

Function is a block of code to perform a particular task and is executed when in some part of the code it calls it.

2.10.1. Function syntax

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().

Syntax,

```
function fun_name(arg1, arg2, ..., argn){
    // code to be executed
}
```

Example,

```
function add(op1, op2){
    return op1+op2;
}
```

2.10.2. Call a function

The call of a function is done when in some part of the code it calls it as follow,

Syntax

```
fun_name(arg1, arg2, arg3,..., argN);
```

Note: If a function is called with less than N args the rest of arguments will remain undefined.

Example,

```
function add(op1,op2){
    return op1+op2;
}

var j=add(2,3); // calls add function. j=5
```

2.10.3. Function object

A function can be stored in variables through its reference,

```
function add(op1, op2){
    return op1+op2;
}

var fun_obj = add; // stored function add reference to fun_obj
var j=fun_obj(2,3);// calls fun_obj (aka add) function. J=5
```

Also is possible to create function objects,

Syntax

```
function(arg1, arg2, ..., argN){
    // code to be executed.
};
```

Example,

```
var add=function(op1, op2){
    return op1+op2;
};

var j=add(2,5); // j=5
```

2.11. Class

A class is a custom type that contains variables and functions that operates with its variables. A class is defined in ZetScript using keyword class followed by the name of class. To access class variables within functions use the this keyword in order to access to variable or functions inside class. In a class we can find member functions (functions that affects to class variable) or static functions (helper function of generic purposes about the class type).

Example,

```
class Test{

};
```

2.11.1. Post add function/variable member

In ZetScrip is possible to add more class member through "::" punctuator.

Example,

```
    // post declaration of variable member
    const Test::data2;

    // post declaration of function member
    function Test::function2(){
        this.data2="a string";
    }
```

2.11.2. Instance class

To instance a class is done through the keyword new

Example,

```
var t = new Test(); // Instantiate t as Test type.
```

2.11.3. Accessing to class functions

To access class variables/functions is done through "." operator.

Example,

```
var i=t.function1(2); // initializes data1 as 2 and return the value
```

2.11.4. Constructor

Each time class is instanced, their member variables are undefined. var t = new Test(); // The a class Test is instanced but data1 and data2 are undefined.

```
print("data1:"+t.data1); //  prints: data1:undefined"
```

The constructor is a function that is invoked automatically and with aim to initialize all member variables. To the define a constructor we have to define a function member with same name as the Class.

Example,

```
class Test{
    // Constructor function
    constructor(){
        this.data1 =10; // instantiate data1 as integer
    }
}
```

2.11.5. Inheritance

ZetScript supports inheritance through ":" punctuator after the name of the class followed the class name to be extended. The new extended class will inheritance all variable/functions members from base class.

Example,

```
class TestExtended: Test{
    function3(){
        this.data3=this.data1+this.function1(10);
    }
};
```

Call parent functions (super keyword)

The extended class can call parent functions through super keyword.

Example,

```
class TestExtended extends Test{
    function1(a){
        var t=super(a); // it calls Test::function1(2)
        this.data1+=t; // Now data1=5+2 = 7
        return this.data1+a;
    }

    function3(){
        this.data3=this.data1+this.function1(5);
    }
};
```

2.11.6. Metamethods

Metamethods are special functions members that links with operators seen on section section 3.6. ZetScript metamethods can be static or member function [2] depending whether the operation affects or not the object itself.

ZetScript supports the following metamethods:

- _equ
- _nequ
- _lt
- _lte

- _not
- _gt
- _gte
- _neg
- _btw
- _add
- _sub
- _div
- _mul
- _mod
- _and
- _or
- _xor
- _shl
- _shr
- _set
- _add_set
- _sub_set
- _mul_set
- _div_set
- _mod_set
- _and_set
- _or_set
- _xor_set
- _shl_set
- _shr_set
- _toString
- _post_inc
- _post_dec
- _pre_inc
- _pre_dec
- _in

_equ (aka ==)

@Description: Performs relational equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if equal, false otherwise.

Example how to use _equ metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    _equ(op1, op2){
        return op1.num==op2.num;
    }
};
```

var n1 = new MyNumber (1), n2=new MyNumber (1);

if(n1==n2){ // we use here the metamethod == print("n1 ("n1.num") is equal to n2 ("n2.num")"); }

_nequ (aka !=)

@Description: Performs relational not equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if not equal, false otherwise.

Example how to use _nequ metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }

    _nequ(op1, op2){
        return op1.num!=op2.num;
    }
};

var n1 = new MyNumber (1), n2=new MyNumber (0);
if(n1!=n2){
    Console::outln("n1 ("+n1.num+") is not equal to n2 ("+n2.num+")");
}
```

_lt (aka <)

@Description: Performs relational less equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if less equal, false otherwise.

Example how to use _lt metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _lt(op1, op2){
        return op1.num<op2.num;
    }
};


var n1 = new MyNumber (0), n2=new MyNumber (1);
if(n1<n2){
    Console::outln("n1 < n2");
}
```

_lte (aka ⇐)

@Description: Performs relational less equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if less equal, false otherwise.

Example how to use _lte metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _lte(op1, op2){
        return op1.num<=op2.num;
    }
};


var n1 = new MyNumber (1), n2=new MyNumber (1);
if(n1<=n2){
    System::outln("n1 <= n2");
}
```

_gt (aka >)

@Description: Performs relational greater operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if greater, false otherwise.

Example how to use _gt metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _gt(op1, op2){
        return op1.num>op2.num;
    }
};

var n1 = new MyNumber (1), n2=new MyNumber (0);
if(n1>n2){
    Console::outln("n1 ("+n1.num+") is greater than n2 ("+n2.num+")");
}
```

_gte (aka >=)

@Description: Performs relational greater equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if greater equal, false otherwise.

Example how to use _gte metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _gte(op1, op2){
        return op1.num>=op2.num;
    }
};

var n1 = new MyNumber (2), n2=new MyNumber (1);
if(n1>=n2){
    Console::outln("n1 >= n2");
}
```

_not (aka !)

@Description: Performs a not operation. @Param1 : Object custom class type. @Returns : A Boolean type as a result of not operation.

Example how to use _not metamethod within script class,

```
class MyBoolean{
    constructor(_b){
        this.b=_b;
    }
    _not(){
        return !_op.b;
    }
};

var b = new MyBoolean (true);
if(!b){
    Console::outln("b was false");
}
```

_neg (aka -)

@Description: Performs negate operation. @Param1 : operand to negate. @Returns : A new object custom class type with result of negate operation.

Example how to use _neg metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }

    _neg(){
        return new MyNumber(-op1.num);
    }
};

var n1 = new MyNumber (1);
var n2 = -n1;
```

_add (aka +)

@Description: Performs add operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result add operation.

Example how to use _add metamethod within script class,

```
class MyNumber{

    constructor(_n){
        this.num=_n;
    }
    static _add(op1,op2){
        return new MyNumber(op1.num+op2.num);
    }
};

var n1 = new MyNumber (20);
var n2 = new MyNumber (10);
var n3 =n1+n2;

print("n1 ("+n1.num+") n2 ("+n2.num+") = "+n3.num);
```

_div (aka /)

@Type: Static @Description: Performs divide operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result divide operation.

Example how to use metamethod _div within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _div(op1,op2){
        return new MyNumber(op1.num/op2.num);
    }
};

var n1 = new MyNumber (20);
var n2 = new MyNumber (10);
var n3 =n1/n2;
```

_mul (aka *)

@Type: Static @Description: Performs multiply operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result multiply operation.

Example how to use _mul metamethod within script class,

```
class MyNumber{

    constructor(_n){
        this.num=_n;
    }
    static _mul(op1,op2){
        return new MyNumber(op1.num*op2.num);
    }
};

var n1 = new MyNumber (20);
var n2 = new MyNumber (10);
var n3 = n1*n2;
```

_mod (aka %)

@Description: Performs modulus operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result modulus operation.

Example how to use _mod metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    _mod(op1,op2){
        return new MyNumber(op1.num%op2.num);
    }
};

var n1 = new MyNumber (20);
var n2 = new MyNumber (15);
var n3 = n1%n2;
```

_and (aka &)

@Description: Performs binary and operation between two integer operands. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result of binary and operation.

Example how to use _and metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _and(op1,op2){
        return new MyNumber(op1.num&op2.num);
    }
};

var n1 = new MyNumber (0xff);
var n2 = new MyNumber (0x0f);
var n3 =n1&n2;
```

_or (aka |)

@Description: Performs binary or operation between two integer operands. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result of binary or operation.

Example how to use _or metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _or(op1,op2){
        return new MyNumber(op1.num|op2.num);
    }
};

var n1 = new MyNumber (0xf0);
var n2 = new MyNumber (0x0f);
var n3 =n1|n2;
```

=====_xor (aka ^)

@Description: Performs a binary xor operation between two integer operands. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result of binary xor operation.

Example how to use _xor metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _xor(op1,op2){
        return new MyNumber(op1.num^op2.num);
    }
};
var n1 = new MyNumber (0xf1);
var n2 = new MyNumber (0x0f);
var n3 =n1^n2;
```

_shl (aka <<)

@Description: Performs shift left operation. @Param1 : Variable to apply shift left. @Param2 : Tells number shifts to the left. @Returns : A new object custom class type with n shifts left operation.

Example how to use _shl metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _shl(op1, n_shifts){
        return new MyNumber(op1.num<< n_shifts);
    }
};

var n1 = new MyNumber (0x1);
var n2 = n1 << 3;
```

_shr (aka >>)

@Description: Performs shift right operation. @Param1 : Variable to apply shift right. @Param2 : Tells number shifts to the right. @Returns : A new object custom class type with n shifts right operation.

Example how to use _shr metamethod within script class,

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    static _shr(op1,n_shifts){
        return new MyNumber(op1.num>>n_shifts);
    }
};

var n1 = new MyNumber (0xf);
var n2 = n1 >> 2;
```

_set (aka =)

@Description: Performs a set operation6. @Param1 : Source variable to set. @Returns : None.

We present a simple example how to use set metamethod within script class. In the set metamethod we can filter which type of parameter input is to perform the right operation and stop execution with error function if is required.

```
class MyNumber{
    constructor(_n){
        this.num=_n;
    }
    _set(v){
        if(v instanceof Integer){
            this.num = v;
        }else if(v instanceof MyNumber){
            this.num = v.num;
        }else{
            error("parameter not supported");
        }
    }
};

var n1 = new MyNumber (10);
var n2; // n3 is undefined!
n2 = n1; // it assigns n1=n2.
n2 = 50; // n3 now values 50
```

If variable is undefined ZetScript will assign reference object, in the case is not defined it will do a set operation (if it is implemented).

Mixing operand types

Working with metamethods might have situations where you are passing different type parameters. You can pass the object type, where metamethod function is implemented, or other type of parameters like integer, string, etc.The following example performs a sums of a combination of object, integers or floats.

```
var num1= new MyNumber(1), num2=new MyNumber(2);
var num3= 1.0 + num1 + 6 + 1 + 10.0 + num2 + 10 + num1 + num2;
```

The expression cannot be performed with only objects as we have been shown in the last sections. You can use instanceof operator to check each type of argument and perform the needed operation.

We present an example for _add metamethod function that implements a support to operate with MyNumber object, integer or float. Other types will cause a execution error.

Example,

```
class MyNumber{

    constructor(_n){
        this.num=_n;
    }
    static _add(op1,op2){
        var aux1, aux2;
        if(op1 instanceof MyNumber){
            aux1=op1.num;
        }else if(op1 instanceof Integer || op1 instanceof Float){
            aux1=op1;
        }else{
            System::errorln("arg op1 is not supported");
        }

        if(op2 instanceof MyNumber){
            aux2=op2.num;
        }else if(op2 instanceof Integer || op2 instanceof Float){
            aux2=op2;
        }else{
            System::errorln("arg op2 is not supported ");
        }

 return new MyNumber(aux1+aux2);
 }
};

var n1 = new MyNumber (20);
var n2 = new MyNumber (10);
var n3 =1+n1+5+7+n2+10.0+7.0+10; // mix operation with MyNumber, integer and number
```

2.11.7. Properties

A property is a member that defines a set of metamethods that to operate with.

[1] A block is a statement that starts with '{' and ends with '}'

[2] On script side, static function is defined as member function, but user should not access on variable/function members as well it happens on c++ static function.

Chapter 3. the API

3.1. Bind function

To bind a function first it has to declare a function with a signature compatible to ZetScript. All, functions will have ZetScript pointer as first parameter, and the others will be built-in types or types defined by the user.

binding supports **zs_int, zs_int *, zs_float *, bool *, char * and zs_string *** as arguments types. For return values it supports the same basic types as arguments plus **bool, zs_float and zs_string (no pointer)**. Is also possible to pass custom registered types (see section 4.3).

Is important to say that ZetScript has a constraint **maximum of 8 parameters** for any C function to be bind in the script engine. If a function more than 8 parameters is tried to be registered, ZetScript will throw an error. To bind C function is done through macro bindFunction. You have to provide the name that will be referenced in script side.

Syntax,

```
zs.bindFunction("function_name", c_function);
```

Example,

```
#include "ZetScript.h"

zetscript::zs_int add(zetscript::zs_int  op1, zetscript::zs_int op2)
{
    return op1+op2;
}

int main(int argc, char *argv[])
{
    zetscript::ZetScript zs;

    // binds native function "add"
    zs.bindFunction("add",add);

    zs.eval(
        "Console::outln(\"result:\"+add(5,4));" // prints "result: 9"
    );

    return 0;
}
```

3.1.1. Ambiguities

If we try to register another function with the same name the c++ compiler will fail to register function.

For instance,

```
#include "ZetScript.h"

// a function that adds two integers.
zetscript::zs_int add(zetscript::zs_int op1, zetscript::zs_int op2)
{
    return op1+op2;
}

// a function that adds two floats return op1+op2;
zetscript::zs_float add(zetscript::zs_float *op1, zetscript::zs_float *op2)
{
    return *op1+*op2;
}

int main(int argc, char *argv[])
{
    zetscript::ZetScript zs;

    // c++ compiler throws an error due function deduction.
    zs.bindFunction("add",add);

    return 0;
}
```

For instance gnu c++ compiler throws this error,

note: template argument deduction/substitution failed:

That means that indeed there're more than one function named 'add' and the compiler cannot deduce which function take to register. To achieve register more than a one function with the same name we have to make a cast to get the function we want with its signature.

Example,

```
#include "ZetScript.h"

// a function that adds two integers.
zetscript::zs_int add(zetscript::zs_int op1, zetscript::zs_int op2)
{
    return op1+op2;
}

```

```
// a function that adds two floats return op1+op2;
zetscript::zs_float add(zetscript::zs_float *op1, zetscript::zs_float *op2)
{
    return *op1+*op2;
}

int main(int argc, char *argv[])
{
    zetscript::ZetScript zs;

    // register function add(int,int)
    zs.bindFunction("add",static_cast<int (*)(int,int)>(add));

     // register function add(float *,float *)
    zs.bindFunction("add",static_cast<float (*)(float *,float *)>(add));

    zs.eval(
        "Console::outln(\"result:\"+add(5,4));" // prints "result:9"
        "Console::outln(\"result:\"+add(0.5,4.6));" // prints "result:5.1"
    );

    return 0;
}
```

3.2. Bind types

3.2.1. Bind static type

Binding a native class or type as static it means it’s not instantiable in the script side , so it cannot use **new** keyword. To bind type as static is done through **bindType** function by passing the type as template and the name that will be referenced in the script.

Example,

```
zs.bindType<type_class>("name_class");
```

The following code shows an example of a registering a C++ class,

```
class MyClass{
public:
    int data1;
    void init(int arg){
        printf("data1 is initialized as %i\n",arg);
        this->data1=arg;
    }

    void function1(int arg){
        this->data1 = arg;
        printf("c++ argument is %i\n",this->data1);
    }
};
```

List 4.1

Using the list 4.1, to bind MyClass as static (i.e no instantiable) in script side is proceded as follows,

```
void main(argc, char *argv[])
{
    zetscript::ZetScript zs;

    //register MyClass as static (i.e no instantiable) in script side.
    zs.bindType<MyClass>("MyClass");

    return 0;
 }
```

In the following example it evals a code to try instance MyClass type but it will throw an error that MyClass is not instanciable because is static,

```
void main(argc, char *argv[])
{
    zetscript::ZetScript zs;

    //register MyClass as static (i.e no instantiable) in script side.
    zs.bindType<MyClass>("MyClass");


    // It throws an error that MyClass is not instanciable because is static
    zs.eval(
        "var myclass= new MyClass();"
    );

    return 0;
 }
```

In order to use a MyClass instantiation it has to be done in the C++ aplication.

For example,

```
MyClass *my_class=NULL;

// interface function to get MyClass instantiation
MyClass *getMyClass(){
    return  my_class;
}
```

```
void main(argc, char *argv[])
{
    zetscript::ZetScript zs;

    // create MyClass instantiation from C++
    my_class=new MyClass();


    //register MyClass as static (i.e no instantiable) in script side.
    zs.bindType<MyClass>("MyClass");

     //register function interface to get MyClass instantiation
    zs.bindFunction("getMyClass",getMyClass);


    // It get MyClass reference
    zs.eval(
        "var my_class= getMyClass();"
    );

    // delete MyClass instantiation from C++
    delete my_class;

    return 0;
 }
```

Note: Is valuable to see that static types are safety because never are created in the script side, they are created by the C++ side

3.2.2. Bind instantiable type

Binding class or type as instantiable means that can it be instanced in the script side by using **new** keyword. To bind a instantiable type is done through **bindType** passing the type as template,the name and the interface new/delete functions,

Example,

```
 zs.bindType<MyClass>("MyClass",new_function, delete_function);
```

Using the list 4.1, to bind MyClass as instanciable in script side is proceded as follows,

```
MyClass *MyClass_new(){
    return new MyClass();
}

void MyClass_delete(MyClass *_this){
    delete _this;
}


void main(argc, char *argv[])
{
    zetscript::ZetScript zs;

    //register MyClass as instantiable type in script side.
    zs.bindType<MyClass>("MyClass",MyClass_new,MyClass_delete);

    // It instances MyClass
    zs.eval(
        "var my_class= new MyClass();"
    );

    return 0;
}
```

3.2.3. Delete C Class

ZetScript it has a garbage collector to delete unreferenced script variables when the end of scope is reached but it keeps alive its internal native pointer to avoid unintended segmentation faults. So to avoid memory leaks due this issue, the user has to delete manually any instanced C Class variable with delete keyword. The following code shows an example of using delete keyword,

```
 delete myclass; // script and c variable is destroyed.
```

3.3. Bind members

3.3.1. Bind Function Member

The binding of variable member is done like binding c function but in this case is done through the macro function register_C_FunctionMember. You have to provide the type class, the string name that will be referenced in script side and the function object reference.

register_C_FunctionMember<ObjectType>("function_name",&ObjectType::function_name);

As an example, the following code registers function member MyClass::function1 seen on List 4.1

```
zs.bindFunctionMember<MyClass>("function1",&MyClass::function1);
```

And then it can access to function1 member through field access ('.')

```
var myclass= new MyClass();
```

myclass.function1(10); // prints "c++ argument is10"

3.3.2. Bind function constructor

ZetScript always calls default C constructor when a variable is instanced with C type. ZetScript has no support of parameterized constructors but, instead, it can be done by registering a function with same name as the class name registered. As an example, the following code registers function member MyClass::init seen on List 4.1 as constructor3,

```
zs.bindFunctionMember<MyClass>("MyClass",&MyClass::init);
```

And then, when variable is intancedwe can instance the class passing a integer as parameter to the c contructor

```
var myclass= new MyClass(10); // prints "data1 is initialized as 10"
```

4.3.6 Inheritance

Inherited classes needs to know its base classes in order to register its parent variables and symbols already registered with the functions already seen in the section 4.3.3 and 4.3.4 respectively. To tell the which base class has an inherited class is done through class_C_baseof with two parameters: The first parameter as the inherited class type and second parameter as its base class type. Syntax,

```
zs.extends<class, base_class>();
```

If for example we want to register MyClassExtend and tell that is base of MyClass Is done with the following snipped,

```
class MyClassExtend:public MyClass{
public:
    float data2;
    void function2(float * arg){
        this->data2 = *arg;
        printf("Float argument is %.02f\n",this->data2);
    }
};

zs.bindType<MyClassExtend>("MyClassExtend"); // register MyClassExtend
zs.extends<MyClassExtend,MyClass>();
```

List 4.2

3 Note that the name of the function is the same as the name of the class

3.4. Bind static constant variable

The binding of variable member is done through the macro function **bindMemberVariable**. You have to provide the type class, the string name that will be referenced in script side and variable object reference.

Sintax,

```
zs.bindStaticConstantVariableMember<ObjectType>("variable_name",&ObjectType::variable_
name);
```

As an example, the following code register variable member MyClass::data1 seen on List 4.1, register_C_VariableMember<MyClass>("data1",&MyClass::data1); And then it can access to data1 member through field access ('.') var myclass= new MyClass(); print("data1"+myclass.data1);

```
 4.4 Inheritance script class from c++ class
An important feature of ZetScript is that it supports c++ class inheritance for any in
script class and the this (section 3.9) and super (seccion 3.9.5.1) keywords works as
a
normal behavior
For example, we could inherit MyClassExtend from 4.2 that is shown in the following
code,
class ScriptMyClassExtended: MyClassExtend{
 function function1(arg1){
 print("script argument is "+arg1)
 super(this.data1+arg1); // calls function1 c++
 }
}
var myclass=new ScriptMyClassExtend(10);
Myclass.function1(5);
It prints,
data1 is initialized as 10
script argument is 5
c++ argument is 15
```

Complete example #include "CZetScript.h" using namespace zetscript; class MyClass{ public: int data1; void init(int arg){ printf("data1 is initialized as %i\n",arg); this→data1=arg; } void function1(int arg){ this→data1 = arg; printf("c++ argument is %i\n",this→data1); } }; class MyClassExtend:public MyClass{ public: float data2; void function2(float *arg){ this→data2 = *arg; printf("Float argument is %.02f\n",this→data2); } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // instance zetscript

```
register_C_Class<MyClass>("MyClass"); //register MyClass as MyClass in script side
register_C_Class< MyClassExtend >("MyClassExtend"); // register MyClassExtend
class_C_baseof<MyClassExtend,MyClass>();
```

```
// register MyClass::constructor
register_C_FunctionMember<MyClass>("MyClass",&MyClass::init);
//reg MyClass:: data1
register_C_VariableMember<MyClass>("data1",&MyClass::data1);
//reg MyClass:: function1
register_C_FunctionMember<MyClass>("function1",&MyClass::function1);
```

```
 // eval print
 if(!zs->eval(
 "class ScriptMyClassExtend: MyClassExtend{\n"
 "function function1(arg1){\n"
 "print(\"script argument is \"+arg1);\n"
 "super(this.data1+arg1); // calls function1 c++\n"
 "}\n"
 "};\n"
 "var myclass=new ScriptMyClassExtend(10);\n"
 "myclass.function1(5);\n"
 "delete myclass; // script and c variable is destroyed.\n"
 )){
 fprintf(stderr,CZetScript::getInstance()->getErrorMsg());
 }
 return 0;
}
```

4.5 Call script function in C To bind script call in c it can be done through bind_function passing the function type as template parameter and the function name as parameter4 . It can bind a script function member from an already instanced object. Example, #include "CZetScript.h" using namespace zetscript; int main(){ CZetScript *zs = CZetScript::getInstance(); // instance zetscript zs→eval("class Test{" " var data1;" " function function1(arg){" " print(\"calling Test.Function:\"+arg);" " }" "};" "" "function delete_test(){" " delete test;" " print(\"test variable was deleted\");" "}" "" "var test=new Test();"); // delete_test function is evaluated now test variable is instanced as Test type, so it can // bind test.function1

std::function<void()> * delete_test=bind_function<void()>("delete_test"); std::function<void(int)> * test_function1=bind_function<void (int)>("test.function1"); (*test_function1)(10); // it calls "test.function" member function with 10 as parameter. (*delete_test)(); // it calls "delete_test" function with no parameters // delete functions when they are used anymore delete test_function1; delete delete_test; }

4 C++ function binding is limited by a maximum of 6 parameters

Chapter 4. 3.10 Metamethods

Metamethods are special functions members that links with operators seen on section section 3.6. ZetScript metamethods can be static or member function [1] depending whether the operation affects or not the object itself.

ZetScript supports the following metamethods:

- _equ
- _not_equ
- _lt
- _lte
- _gt
- _gte
- _not
- _neg
- _add
- _div
- _mul
- _mod
- _and
- _or
- _xor
- _shl
- _shr
- _set

4.1. 5.5.1 _equ (aka ==)

@Description: Performs relational equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if equal, false otherwise. Script Example Example how to use _equ metamethod within script class,

class MyNumber{ var num; function MyNumber(_n){ this.num=_n; } function _equ(op1, op2){ return op1.num==op2.num; } };

var n1 = new MyNumber (1), n2=new MyNumber (1);

if(n1==n2){ // we use here the metamethod == print("n1 ("n1.num") is equal to n2 ("n2.num")"); }

C Example The same it can be done with C. The C++ metamethod function associated with must be static. #include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num;

MyNumber(){ this→num=0; } void set(int _n){ this→num=_n; } static bool _equ(MyNumber *op1, MyNumber *op2){ return op1→num == op2→num; } };

int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _equ as metamethod register_C_StaticFunctionMember<MyNumber>("_equ",&MyNumber::_equ); if(!zs→eval("var n1 = new MyNumber (1), n2=new MyNumber (1); \n " "if(n1==n2){ // we use here the metamethod ==\n " " print(\"n1 (\"n1.num\") is equal to n2 (\"n2.num\")\");\n " "}\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.2. 5.5.2 _nequ (aka !=)

@Description: Performs relational not equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if not equal, false otherwise.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } void set(int _n){ this→num=_n; } static bool _nequ(MyNumber *op1, MyNumber *op2){ return op1→num != op2→num; } };

int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _not_equ as metamethod register_C_StaticFunctionMember<MyNumber>("_nequ",&MyNumber::_nequ); if(!zs→eval("var n1 = new MyNumber (1), n2=new MyNumber (0); \n " "if(n1!=n2){ // we use here the metamethod != \n " " print(\"n1 (\"n1.num\") is not equal to n2 (\"n2.num\")\");\n " "}\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.3. 5.5.3 _lt (aka <)

@Description: Performs relational less equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if less equal, false otherwise.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } void set(int _n){ this→num=_n; } static bool _lt(MyNumber *op1, MyNumber *op2){ return op1→num < op2→num; } };

int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _lt as metamethod register_C_StaticFunctionMember<MyNumber>("_lt",&MyNumber::_lt); if(!zs→eval("var n1 = new MyNumber (0), n2=new MyNumber (1);\n" "if(n1<n2){ \n " " print(\"n1 (\"n1.num\") is less than n2 (\"n2.num\")\");\n " "}\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.4. 5.5.4 _lte (aka ⇐)

@Description: Performs relational less equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if less equal, false otherwise.

C Example The same it can be done with C. The C++ metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } void set(int _n){ this→num=_n; } static bool _lte (MyNumber *op1, MyNumber *op2){ return op1→num ⇐ op2→num; } };

int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _lte as metamethod register_C_StaticFunctionMember<MyNumber>("_lte",&MyNumber::_lte); if(!zs→eval("var n1 = new MyNumber (1), n2=new MyNumber (1);\n" "if(n1⇐n2){\n" " print(\"n1 (\"n1.num\") is less equal than n2 (\"n2.num\")\");\n" "}\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.5. 5.5.5 _gt (aka >)

@Description: Performs relational greater operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if greater, false otherwise.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } void set(int _n){ this→num=_n; } static bool _gt(MyNumber *op1, MyNumber *op2){ return op1→num > op2→num; } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _gt as metamethod register_C_StaticFunctionMember<MyNumber>("_gt",&MyNumber::_gt); if(!zs→eval("var n1 = new MyNumber (1), n2=new MyNumber (0);\n" "if(n1>n2){ \n" " print(\"n1 (\"n1.num\") is greater than n2 (\"n2.num\")\");\n" "}\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.6. 5.5.6 _gte (aka >=)

@Description: Performs relational greater equal operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : true if greater equal, false otherwise.

C++ Example

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } void set(int _n){ this→num=_n; } static bool _gte(MyNumber *op1, MyNumber *op2){ return op1→num >= op2→num; } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _gte as metamethod register_C_StaticFunctionMember<MyNumber>("_gte",&MyNumber::_gte); if(!zs→eval("var n1 = new MyNumber (1), n2=new MyNumber (1); \n " "if(n1>=n2){ \n " " print(\"n1 (\"n1.num\") is greater equal than n2 (\"n2.num\")\");\n " "}\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.7. 5.5.7 static _not (aka !)

@Description: Performs a not operation. @Param1 : Object custom class type. @Returns : A Boolean type as a result of not operation.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyBoolean{ public: bool b; MyBoolean (){ this→b=false; } void set(bool _b){ this→b=_b; } static bool _not(MyBoolean *op1){ return !op1→b; } };

int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class< MyBoolean >("MyBoolean"); // register variable member num register_C_VariableMember<MyBoolean>("b", &MyBoolean::b); // register constructor through function MyNumber::set register_C_FunctionMember<MyBoolean>("MyBoolean", &MyBoolean:: set); // register static function _not as metamethod register_C_StaticFunctionMember<MyBoolean>("_not", &MyBoolean::_not); if(!zs→eval("var b = new MyBoolean (false);\n" "if(!b){ \n" " print(\"b was false\");\n" "}\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.8. 5.5.8 _neg (aka -)

@Description: Performs negate operation. @Param1 : operand to negate. @Returns : A new object custom class type with result of negate operation.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } MyNumber(int _num){ this→num=_num; } void set(int _n){ this→num=_n; } static

MyNumber * _neg(MyNumber *op1){ return new MyNumber(-op1→num); } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _neg as metamethod register_C_StaticFunctionMember<MyNumber>("_neg",&MyNumber::_neg); if(!zs→eval ("var n1 = new MyNumber (1);\n" "var n2 = -n1;\n" "print(\"neg of n1 (\"n1.num\") is (\"n2.num\")\");\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.9. 5.5.9 _add (aka +)

@Description: Performs add operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result add operation.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } static MyNumber * _add(MyNumber *op1, MyNumber *op2){ return new MyNumber(op1→num + op2→num); } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _add as metamethod register_C_StaticFunctionMember<MyNumber>("_add",&MyNumber::_add); if(!zs→eval("var n1 = new MyNumber (20);\n" "var n2 = new MyNumber (10); \n" "var n3 =n1+n2; \n " "print(\"n1 (\"n1.num\") + n2 (\"n2.num\") = \"+n3.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.10. 5.5.10 _div (aka /)

@Type: Static @Description: Performs divide operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result divide operation.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } static MyNumber *_div(MyNumber *op1, MyNumber *op2){ return new MyNumber(op1→num / op2→num); } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _div as metamethod register_C_StaticFunctionMember<MyNumber>("_div",&MyNumber::_div); if(!zs→eval("var n1 =

new MyNumber (20);\n" "var n2 = new MyNumber (10);\n" "var n3 =n1/n2;\n" "\n" "print(\"n1 (\"n1.num\") / n2 (\"n2.num\") = \"+n3.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.11. 5.5.11 _mul (aka *)

@Type: Static @Description: Performs multiply operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result multiply operation.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } static MyNumber *_mul(MyNumber *op1, MyNumber *op2){ return new MyNumber(op1→num * op2→num); } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _mul as metamethod register_C_StaticFunctionMember<MyNumber>("_mul",&MyNumber::_mul); if(!zs→eval("var n1 = new MyNumber (20);\n" "var n2 = new MyNumber (10);\n" "var n3 =n1*n2;\n" "\n" "print(\"n1 (\"n1.num\") * n2 (\"n2.num\") = \"+n3.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.12. 5.5.12 _mod (aka %)

@Description: Performs modulus operation. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result modulus operation.

C++ Example

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } static MyNumber *_mod(MyNumber *op1, MyNumber *op2){ return new MyNumber(op1→num % op2→num); } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _mod as metamethod register_C_StaticFunctionMember<MyNumber>("_mod",&MyNumber::_mod); if(!zs→eval("var n1 = new MyNumber (20);\n" "var n2 = new MyNumber (15);\n" "var n3 =n1%n2;\n" "\n" "print(\"n1 (\"n1.num\") % n2 (\"n2.num\") = \"+n3.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.13. 5.5.13 _and (aka &)

@Description: Performs binary and operation between two integer operands. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result of binary and operation.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } static MyNumber * _and(MyNumber *op1, MyNumber *op2){ return new MyNumber (op1→num & op2→num); } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _and as metamethod register_C_StaticFunctionMember<MyNumber>("_and",&MyNumber::_and); if(!zs→eval("var n1 = new MyNumber (0xff);\n" "var n2 = new MyNumber (0x0f);\n" "var n3 =n1&n2;\n" "\n" "print(\"n1 (\"n1.num\") & n2 (\"n2.num\") = \"+n3.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.14. 5.5.14 _or (aka |)

@Description: Performs binary or operation between two integer operands. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result of binary or operation.

C Example The same it can be done with C. The C++ metamethod function associated with must be static. #include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } static MyNumber * _or(MyNumber *op1, MyNumber *op2){ return new MyNumber(op1→num | op2→num); } };

int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _or as metamethod register_C_StaticFunctionMember<MyNumber>("_or",&MyNumber::_or); if(!zs→eval("var n1 = new MyNumber (0xf0);\n" "var n2 = new MyNumber (0x0f);\n" "var n3 =n1|n2;\n" "\n" "print(\"n1 (\"n1.num\") | n2 (\"n2.num\") = \"+n3.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.15. 5.5.15 _xor (aka ^)

@Description: Performs a binary xor operation between two integer operands. @Param1 : 1st operand. @Param2 : 2nd operand. @Returns : A new object custom class type with result of binary xor operation.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } static MyNumber *_xor(MyNumber *op1, MyNumber *op2){ return new MyNumber(op1→num ^ op2→num); } };

int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _xor as metamethod register_C_StaticFunctionMember<MyNumber>("_xor",&MyNumber::_xor); if(!zs→eval("var n1 = new MyNumber (0xf1);\n" "var n2 = new MyNumber (0x0f);\n" "var n3 =n1^n2;" "\n" "print(\"n1 (\"n1.num\") ^ n2 (\"n2.num\") = \"+n3.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.16. 5.5.16 _shl (aka <<)

@Description: Performs shift left operation. @Param1 : Variable to apply shift left. @Param2 : Tells number shifts to the left. @Returns : A new object custom class type with n shifts left operation.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } static MyNumber *_shl(MyNumber *op1, int n_shifts){ return new MyNumber(op1→num << n_shifts); } };

int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _shl as metamethod register_C_StaticFunctionMember<MyNumber>("_shl",&MyNumber::_shl); if(!zs→eval("var n1 = new MyNumber (0x1);\n" "var n2 = n1 << 3;\n" "\n" "print(\"n1 (\"n1.num\") << 3 = \"+n2.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.17. 5.5.17 _shr (aka >>)

@Description: Performs shift right operation. @Param1 : Variable to apply shift right. @Param2 : Tells number shifts to the right. @Returns : A new object custom class type with n shifts right operation.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){

this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } static MyNumber * _shr(MyNumber *op1,int n_shifts){ return new MyNumber(op1→num >> n_shifts); } };

int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register static function _shr as metamethod register_C_StaticFunctionMember<MyNumber>("_shr",&MyNumber::_shr); if(!zs→eval("var n1 = new MyNumber (0xf);\n" "var n2 = n1 >> 2;\n" "\n" "print(\"n1 (\"n1.num\") >> 2 = \"+n2.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.18. 5.5.19 _set (aka =)

@Description: Performs a set operation6. @Param1 : Source variable to set. @Returns : None.

The same it can be done with C. The C metamethod function associated with must be static.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: int num; MyNumber(){ this→num=0; } void _set(int _n){ this→num=_n; } void _set(MyNumber **_n){ this→num=_n→num; } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); // register variable member num register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::_set register_C_FunctionMember<MyNumber>("MyNumber" , static_cast<void (MyNumber::**)(int)>(&MyNumber::_set)); // register two types function _set as metamethod (same as constructor) register_C_FunctionMember<MyNumber>("_set" ,static_cast<void (MyNumber::*)(int)>(&MyNumber::_set)); register_C_FunctionMember<MyNumber>("_set" , static_cast<void (MyNumber::*)(MyNumber *)>(&MyNumber::_set)); if(!zs→eval("var n1 = new MyNumber (10);\n" "var n2 = new MyNumber (20); \n" "var n3; // □ n3 is undefined! \n" "n3 = n2; // □ it assigns n2 pointer. \n" "print(\"n3:\"+n3.num); \n" "n3=n1; // □n3.num = n2.num = n1.num. \n" "print(\"n3:\"+n3.num); \n" "n3=50; // □n3.num = n2.num = 10. \n" "print(\"n3:\"+n3.num); \n" "n3=false; // □stops execution with error because is not supported.\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

4.19. 5.5.20 Mixing operand types

Working with metamethods might have situations where you are passing different type parameters. You can pass the object type, where metamethod function is implemented, or other type of parameters like integer, string, etc. The following example performs a sums of a combination of object, integers or floats.

var num1= new MyNumber(1), num2=new MyNumber(2); var num3= 1.0 + num1 + 6 + 1 + 10.0 + num2 + 10 + num1 + num2;

The expression cannot be performed with only objects as we have been shown in the last sections. You can use instanceof operator to check each type of argument and perform the needed operation.

We present an example for _add metamethod function that implements a support to operate with MyNumber object, integer or float. Other types will cause a execution error.

The same example for C++ we can to do an extra effort. We have to implement all possibilities that operator contemplates with operation within MyNumber, int or float.

#include "CZetScript.h" using namespace zetscript; class MyNumber{ public: float num; MyNumber(){ this→num=0; } MyNumber(int _n){ this→num=_n; } void set(int _n){ this→num=_n; } // MyNumber,MyNumber combination static MyNumber * _add(MyNumber **op1, MyNumber *op2){ return new MyNumber(op1→num + op2→num); } // int,MyNumber combination static MyNumber * _add(int op1, MyNumber *op2){ return new MyNumber(op1 + op2→num); } // MyNumber,int combination static MyNumber * _add(MyNumber *op1, int op2){ return new MyNumber(op1→num + op2); } // float,MyNumber combination static MyNumber * _add(float *op1, MyNumber *op2){ return new MyNumber(*op1 + op2→num); } // MyNumber,float combination static MyNumber * _add(MyNumber *op1, float *op2){ return new MyNumber(op1→num + *op2); } }; int main(){ CZetScript *zs = CZetScript::getInstance(); // register class MyNumber register_C_Class<MyNumber>("MyNumber"); register_C_VariableMember<MyNumber>("num",&MyNumber::num); // register constructor through function MyNumber::set register_C_FunctionMember<MyNumber>("MyNumber",&MyNumber:: set); // register 1st _add metamethod function to satisfy operand (MyNumber,MyNumber) combination register_C_StaticFunctionMember<MyNumber>("_add",static_cast< MyNumber * ()**(MyNumber **, MyNumber *)>(&MyNumber::_add)); // register 2nd _add metamethod function to satisfy operand (int,MyNumber) combination register_C_StaticFunctionMember<MyNumber>("_add",static_cast< MyNumber * ()**(int, MyNumber **)>(&MyNumber::_add)); // register 3rd _add metamethod function to satisfy operand (MyNumber,int) combination register_C_StaticFunctionMember<MyNumber>("_add",static_cast< MyNumber * ()**(MyNumber **, int)> (&MyNumber::_add)); // register 4th _add metamethod function to satisfy operand (float,MyNumber) combination register_C_StaticFunctionMember<MyNumber>("_add",static_cast< MyNumber * ()**(float , **MyNumber *)>(&MyNumber::_add)); // register 5th _add metamethod function to satisfy operand (MyNumber,float) combination register_C_StaticFunctionMember<MyNumber>("_add",static_cast< MyNumber * ()**(MyNumber *, float *)>(&MyNumber::_add)); if(!zs→eval("var n1 = new MyNumber (20);\n" "var n2 = new MyNumber (10);\n" "var n3 =1+n1+5+7+n2+10.0+7.0+10; // mix operation with MyNumber, integer and number\n" "print(\"n3:\"+n3.num);\n")){ fprintf(stderr,ZS_GET_ERROR_MSG()); } return 0; }

[1] On script side, static function is defined as member function, but user should not access on variable/function members as well it happens on c++ static function.

5 Modules

Zetscript implements the following modules,

- String
- Console
- System
- Math
- Json
- DateTime

Chapter 5. 5.1 String

String module exposes the following set of functions,

- String::format

5.1. 5.1.1 String::format

String::format formats a string

Syntax,

```
String::format("string", ...args)
```

The possible formats is described below,

`Format`	description	Example

`{n}`	It replaces {*n*} by the argument *n*	`// prints '1st arg 1'`
		`String::format("1st arg {0}",1)`
		
		`// prints '1st,2nd as 1 and 2'`
		`String::format("1st,2nd as {0} and {1}",1,2)`
		
		`// prints '1st,2nd same 1 and 1'`
		`String::format("1st,2nd same {0} and {0}",1)`

`{n:dm}`	It replaces the argument *n* padding *m* 0s on its left	`// out == 'Score:01'`
		`String::format("Score:{0:d1}",1)`
		
		`// out == 'Score:0001'`
		`String::format("Score:{0:d4}",1)`

`{n,m}`	It replaces argument *n* padding *m* spaces on its left	`// out == 'Score: 1'`
		`var out=String::format("Score:{0,1}",1)`
		`// out == 'Score:   1'`
		`out=String::format("One space:{0,4}",1)`

Chapter 6. 5.2 Console

Console module exposes the following set of functions,

- outln
- errorln
- readChar

Chapter 7. 5.2.1 Console::outln

Console::outln prints message to the console.

Syntax,

```
Console::outln(object,...args)
```

If *object* is constant string it can be formated (see in table X.X)

Example,

```
 // prints 'Hello world'
 Console::outln("Hello world");

 // prints 'Score:0001'
 Console::outln("Score:{0:d4}",1);
```

Chapter 8. 5.2.2 Console::errorln

Console::errorln prints an error message to the console.

Syntax,

```
Console::errorln(object,...args)
```

If *object* is constant string it can be formated (see in table X.X)

Example,

```
 // prints 'Error'
 Console::errorln("Error");

 // prints 'Error:02'
 Console::outln("Error:{0:d2}",2);
```

Chapter 9. 5.2.3 Console::readChar

Console::readChar it waits untill reads a character from console

Syntax,

```
Console::readChar()
```

Example,

```
 // waits until reads a char
 var c=Console::readChar();

 // prints readed char
 Console::outln("Key pressed: {0}",c)
```

Chapter 10. 5.3 System

System module exposes the following set of functions,

- clock
- eval
- assert
- error

Chapter 11. 5.3.1 Clock

Returns the amount of time in milliseconds

Syntax,

```
System::clock()
```

Example,

```
 var start=System::clock()

 // waste some time...
 for(var i=0;i<1000; i++){
 }

 // print elapsed time in milliseconds
 Console::outln("Elapsed time: {0} ms",System::clock()-start_time)
```

Chapter 12. 5.3.2 Eval

Chapter 13. 5.3.3 Assert

Chapter 14. 5.3.4 Error

Chapter 15. 5.4 Math

Math module exposes the following set of static members,

- PI: PI number (i.e 3.14159265359)

Math module exposes the following set of static functions,

- sin
- cos
- abs
- pow
- degToRad
- random
- max
- min
- sqrt
- floor
- ceil
- round

Chapter 16. 5.4.1 Math::sin

Chapter 17. 5.4.2 Math::cos

Chapter 18. 5.4.3 Math::abs

Chapter 19. 5.4.4 Math::pow

Chapter 20. 5.4.5 Math::degToRad

Chapter 21. 5.4.6 Math::random

Chapter 22. 5.4.7 Math::max

Chapter 23. 5.4.8 Math::min

Chapter 24. 5.4.9 Math::sqrt

Chapter 25. 5.4.10 Math::floor

Chapter 26. 5.4.11 Math::ceil

Chapter 27. 5.4.12 Math::round

Chapter 28. 5.5 Json

Json exposes the following set of static members,

- serialize
- deserialize

Chapter 29. 5.5.1 Json::serialize

Chapter 30. 5.5.2 Json::serialize

Chapter 31. 5.6 DateTime

Json exposes the following set of member variables,

- week_day (zs_int): The week's day
- month_day (zs_int): The month's day
- year_day (zs_int): The year's day
- second (zs_int): seconds
- minute (zs_int): minutes
- hour (zs_int): hours
- day (zs_int): day
- month (zs_int):month
- year (zs_int): year

Json exposes the following set of member functions,

- addSeconds
- addMinutes
- addHours
- addDays
- addMonths
- addYears

Json exposes the following set of member metamethods,

- _add
- _sub
- _toString

Chapter 32. 5.6.1 DateTime::addSeconds

Chapter 33. 5.6.2 DateTime::addMinutes

Chapter 34. 5.6.3 DateTime::addHours

Chapter 35. 5.6.4 DateTime::addDays

Chapter 36. 5.6.5 DateTime::addMonths

Chapter 37. 5.6.6 DateTime::addYears

Chapter 38. 5.6.7 DateTime::_add

Chapter 39. 5.6.8 DateTime::_sub

Chapter 40. 5.6.9 DateTime::_toString

Appendix A: 2. Eval

The source of evaluation can came from strings or file.

Chapter 41. 2.1. Eval string

To eval a string is proceed as follows,

```
#include "ZetScript.h"

int main(){

    zetscript::ZetScript zs;

    zs.eval(
        "Console::outln( \"Hello world\")"
    );

    return 0;
}
```

Chapter 42. 2.2. Eval file

Having a file called 'file.zs',

file.zs

```
Console::outln("Hello World");
```

To eval a file is proceed as follow.

```
#include "ZetScript.h"

int main(){

    zetscript::ZetScript zs;

    zs.evalFile(
        "file.zs"
    );

    return 0;
}
```

Chapter 43. 2.3 Eval options

43.1. 2.3.1 No execution

By default, evaluation process compiles and execute but it can tell to not execute passing second parameter with 'EvalOption::EVAL_OPTION_NO_EXECUTE' value,

```
    zs.eval(
     "Console::outln(\"Hello world\")"
     ,EvalOption::EVAL_OPTION_NO_EXECUTE
    );
```

43.2. 2.3.2 Show byte code

To show the current byte code after evaluate expression, pas’EvalOption::EVAL_OPTION_SHOW_BYTE_CODE' property option,

```
    zs.eval(
     "var i=0;\n"
     "i=1;\n"
     ,EvalOption::EVAL_OPTION_SHOW_BYTE_CODE
    );
```

For the code in the list X.XX it’d show the following byte code,

```
 Function: '@MainFunction'
 Stack code: 2
 Stack local vars: 1
 Total stack required: 3

 Scopes: 5
 NUM |RS|AS|               INSTRUCTION
----+--+--+---------------------------------------------
[0000| 1|01]    LOAD_INT                0
[0001| 1|02]    PUSH_STK_LOCAL          i
[0002|-1|00]    STORE                   n:1 [RST]
[0003| 1|01]    LOAD_INT                1
[0004| 1|02]    PUSH_STK_LOCAL          i
[0005|-1|00]    STORE                   n:1 [RST]
```

Where

- Num: Instruction number
- RS: Required Stack

- AS: Acumulated Stack

TODO Explain the marks found in this type of output

Chapter 44. 2.5 Eval exceptions

44.1. 2.5.1 Script exception

To catch eval exceptions or runtime errors it can be done through c++ try/catch, Example,

```
    try{
        zs.evalFile(
            "file.zs"
        );
    }catch(std::exception & ex){
        fprintf(stderr,"Error:%s\n", ex.what());
    }
```

44.2. 2.5.2 User error

If the user wants to stop virtual machine execution due an unexpected value during part of the code, it can call 'System:error' function passing the error message as follows,

```
if(n == undefined){
    System::error("Unexpected error");
}
```

Chapter 45. 2.6 Save/Restore state

ZetScript supports a way to save current compiled state. Save/restore operation is useful when, for instance, user wants to reload a expresions or files and keep previous state.

45.1. 2.6.1 Save state

To save current state we have to invoke 'ZetScript::saveState'. This function returns an index that tells compiled state index saved.

```
zs.saveState()
```

45.2. 2.6.2 Clear

To clear and restore last state saved it’s done by calling 'ZetScript::clear',

```
zs.clear(i)
```

TODO: explain example save/restore state