ZetScript Reference Manual

November 30, 2017

Historic version

Version	ZetScript Version	Author	Description
1.00	1.1.3	Jordi Espada	First version
1.01	1.2.0	Jordi Espada	Changes since 1.2
1.02	1.3.0	Jordi Espada	Changes since 1.3

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!

Contents

1. Introduction 8

1.1 Install 8

1.2. Hello World 8

2. Evaluating and executing scripts 9

2.1. Evaluate string 9

2.2 Evaluate file 10

2.3 Evaluate but no execute 10

2.5 Catch errors from ZetScript 11

2.5.1 Error on evaluating code 11

2.5.2 User errors 11

2.6 Save/Restore state 12

2.6.1 Save state 12

2.6.2 Restore state 12

3. The language 13

3.1 Statments 13

3.2 Comments 13

3.2.1 Block comment 13

3.2.2 Line comment 13

3.3 Constants values 13

3.3.1 Boolean type 13

3.3.2 Integer type 14

3.3.3 Number type 14

3.3.4 String type 14

3.4 Variables 15

3.4.1 Variable declaration 15

3.4.2 Variable instantiation 15
3.4.3 Scope 15
3.5 Built-in types 16
3.5.1 Undefined type 16
3.5.2 Null type 16
3.5.3 Integer type 16
3.5.4 Number type 17
3.5.5 Boolean type 17
3.5.6 string type 18
3.5.7 Vector type 18
3.5.8 Structure type 19
3.5.9 Function object 20
3.6 Operations 21
3.6.1 Arithmetic expressions 21
3.6.2 Relational expressions 21
3.6.3 Logic expressions 22
3.6.4 Binary expressions 22
3.6.5 Priority operations 22
3.7 Conditionals 23
3.7.1 The if statement 23
3.7.2 The else statement 23
3.7.3 The if else statement 24
3.7.4 Ternary condition 24
3.7.5 Switch 24
3.8 Loops 25
3.8.1The While Loop 25
3.8.2 The For Loop 26

3.8.3 Functions 27
3.8.4 Function syntax 27
3.8.5 Call a function 27
3.8.6 Function object 28
3.9 Class 28
3.9.1 Post add function/variable member 29
3.9.2 Instance class 29
3.9.3 Accessing to class functions 29
3.9.4 Constructor 29
3.9.5 Inheritance 30
4 ZetScript API 31
4.1 Bind C variables 31
4.2 Bind C Function 32
4.2.1 Ambiguities 33
4.3 Bind C Class 35
4.3.1 Instance C Class 36
4.3.2 Delete C Class 36
4.3.3 Bind Variable Member 36
4.3.4 Bind Function Member 36
4.3.5 Bind custom class Constructor 37
4.3.6 Inheritance 37
4.4 Inheritance script class from c++ class 38
Complete example 39
4.5 Call script function in C++ 40
5 Metamethods 41
5.1 _equ (aka ==) 42
5.2 _nequ (aka !=) 44

5.3 _lt (aka <)..... 46

5.4 _lte (aka <=)..... 48

5.5 _gt (aka >)..... 50

5.6 _gte (aka >=)..... 52

5.7 static _not (aka !) 54

5.8 _neg (aka -)..... 56

5.9 _add (aka +)..... 58

5.10 _div (aka /)..... 60

5.11 _mul (aka *)..... 62

5.12 _mod (aka %)..... 64

5.13 _and (aka &) 66

5.14 _or (aka |)..... 68

5.15_xor (aka ^)..... 70

5.16 _shl (aka <<) 72

5.17 _shr (aka >>)..... 74

5.19 _set (aka =) 76

5.20 Mixing operand types..... 78

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 Install

To install ZetScript ,first download last source code from http://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
```

Note: For MSVC 2017, it has a feature a Open Folder that easily configures a CMakeFile project just opening the folder where the project is.

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,

```
print("Hello world!");
```

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

```
zs helloworld.zs
```

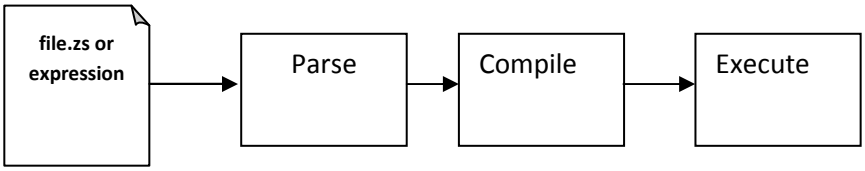
You should see the “Hello world” message at the command line.

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

2. Evaluating and executing scripts

The ZetScript pipeline has three main processes,

1. **Parsing:** It decomposes expressions from input file or input strings into AST Nodes. Also it registers classes and function as it needs.
2. **Compile:** AST Nodes are compiled and it turns in byte code operands.
3. **Execute:** Execute current compiled code.



To evaluate scripts it can be done through a set of string expressions (eval function) or file/s (*eval_file* function)

2.1. Evaluate string

Example evaluating a script through string in c++ ,

main.cpp

```
#include "CZetScript.h"

using namespace zetscript;

int main(){

	CZetScript *zs = CZetScript::getInstance();

	zs->eval(
		"print( \"Hello world\")"
	);

}
```

2.2 Evaluate file

Example evaluating a script through a file,

file.zs

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

main.cpp

```
#include "CZetScript.h"

using namespace zetscript;

int main(){

	CZetScript *zs = CZetScript::getInstance();
	zs->eval_file(
		"file.zs"
	);

}
```

2.3 Evaluate but no execute

By default, evaluation process compiles and execute but it can tell to not execute passing second parameter as false,

```
CZetScript *zs = CZetScript::getInstance();
zs->eval_file(
	"file.zs",
	false //<-- Tell to ZetScript no execute the compiled state
);
```

2.5 Catch errors from ZetScript

2.5.1 Error on evaluating code

To evaluate compilation or runtime errors it can be done through function *CZetScript::getErrorMsg* or with *ZS_GET_ERROR_MSG* macro,

Example,

main.cpp

```
#include "CZetScript.h"

using namespace zetscript;

int main(){

        CZetScript *zs = CZetScript::getInstance();
        if(!zs->eval_file(
                "file.zs"
        )){
          printf("Error:%s\n", CZetScript::getErrorMsg()); // show error
        }

}
```

2.5.2 User errors

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

Example,

file.zs

```
if(n == undefined){

        error("Unexpected error");

}
```

And then it can get the message through getErrorMsg() inside C++ application.

2.6 Save/Restore state

ZetScript supports a way to save current compiled state. This operation saves AST nodes, global variables, registered C variables, C functions and C classes.

Save/restore operation is useful when, for instance, user wants to reload a set of script files. Because global variables must be visible for all script files evaluated, global variables are persistent, so they are not removed on each evaluation, it can only removed by restoring a state.

2.6.1 Save state

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

```
int idx=CState::saveState()
```

2.6.2 Restore state

To restore a previous state we have to invoke *CState::restoreState* passing compiled state index.

```
CState::setState(idx)
```

3. The language

3.1 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;
```

3.2 Comments

ZetScript support block and line comments.

3.2.1 Block comment

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

3.2.2 Line comment

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

3.3 Constants values

ZetScript supports the following constants or literal types,

- boolean
- integer
- float
- string

3.3.1 Boolean type

ZetScript represents boolean value as *true* or *false*.

Example,

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

3.3.2 Integer type

ZetScript represents integer values within ranges from $-(2^{b-1})$ to $2^{b-1}-1$ where b=32 or 64 it depending whether ZetScript is compiled for 32bits or 64bits. The integer value can be represented as negative/positive decimal value, hexadecimal or binary format.

Example,

```
-1;  // negative decimal value
10;  // positive decimal value
0x1a; // hexadecimal value
01001b; // 0x9 in binary value
```

3.3.3 Number type

ZetScript represents number value in 32-bit IEEE-754 floating point number. Number can be represented as integer and decimal form or scientific notation form.

Example,

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

3.3.4 String type

ZetScript represents string value within quotes (“”).

Example,

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

3.4 Variables

3.4.1 Variable declaration

A variable is a type of container that is used to store a value in it. In ZetScript, a variable is declared as **var** keyword.

Example,

```
var i; // declares i variable
```

3.4.2 Variable instantiation

A variable is **undefined** (see 2.5.1) when is declared until is instanced with any value through operator '='. As an example we can instantiate a variable with integer value (see section 2.2),

```
var i=0; // variable v is instancied as integer.
```

3.4.3 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[2], are local.

Example,

```
var i; // this is a global variable
{ // ← block starts here
    var j;//this is a local variable declared inside a block (you can also access to i).
} // ←block ends here, so the variable j doesn't exist anymore
```

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

3.5 Built-in types

ZetScript supports the following built-in types

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

3.5.1 Undefined type

A undefined variable type is the default value that is instanced when any variable is declared. To instantiate a variable with undefined value assign it typing the undefined keyword.

```
var i=undefined;
```

3.5.2 Null type

A null type is like undefined but in this case it has a null value. To instantiate a variable with undefined value type undefined keyword.

```
var i=null;
```

3.5.3 Integer type

A integer variable type is instanced once a variable instance a integer constant or integer variable

Example,

```
var i=10;
```

Pre/Post variable operations

Pre/Post operations modifies the value of variable itself before/after the variable is read. Integer variable type has the following pre/post operations,

Operator	Expression	Description	Result
PreIncrement	`++variable`	Performs an increment BEFORE evaluate the variable	`var i=0;` `var j=++i; // j=1, i =1`
PostIncrement	`variable++`	Performs increment AFTER evaluate the variable	`var i=0;` `var j=i++; // j=0; i=1`
Predecrement	`--variable`	Performs a decrement BEFORE evaluate the variable	`var i=0;` `var j=--i; // j=-1; i=-1`
Postdecrement	`variable--`	Performs decrement AFTER evaluate the variable	`var i=0;` `var j=i--;//j=0;i=-1;`

3.5.4 Number type

A number variable type is instanced once a variable instance a number constant or number variable

Example,

```
var f=1.5;
```

Pre/Post variable operations

Pre/Post operations modifies the value of variable itself before/after the variable is read. Number variable type has the following pre/post operations,

Operator	Expression	Description	Result
PreIncrement	`++variable`	Performs an increment BEFORE evaluate the variable	`var i=0.5;` `var j=++i; // j=1.5, i =1.5`
PostIncrement	`variable++`	Performs increment AFTER evaluate the variable	`var i=0.5;` `var j=i++; // j=0.5; i=1.5`
Predecrement	`--variable`	Performs a decrement BEFORE evaluate the variable	`var i=0.5;` `var j=--i; // j=-0.5; i=-0.5`
Postdecrement	`variable--`	Performs decrement AFTER evaluate the variable	`var i=0.5;` `var j=i--;//j=0.5;i=-0.5;`

3.5.5 Boolean type

A boolean variable type is instanced once a variable instance a boolean constant or boolean variable

Example,

```
var b=false;
```

3.5.6 string type

A integer variable type is instanced once a variable instance a string constant or string variable

Example,

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

3.5.7 Vector type

A vector is an special type to store multiple values in a unidimensional array. the instantiation is done with '[' and ']'.

Example empty vector,

```
var v=[];
```

the user can instantiate a vector with values separated by comas.

Example vector instantiated with values,

```
var v=[1,"string",true,2.0];
```

If the vector has any value its access is done through integer index.

Example,

```
var v=[1,"this is a string",true,2.0]; // its has 4 elements where its access [0..3]
print("v:"+v[1]); // v:this is a string. It acces to 2nd element.
```

Runtime Management

Vector type It has the following functions in order to manage vector at runtime.

Function	Description	Example
`size`	It returns the number of current elements	`var v=[1,2];` `v.size(); // =2`
`push`	Adds value at the end of the vector	`var v=[1,2];` `v.push(3); //v=[1,2,3]`
`pop`	Returns the last value and removes it.	`var v=[1,2];` `v.pop(); //=[2], v=[1]`

3.5.8 Structure type

A structure is an special type to store multiple values in a container giving a name per value so the reference is done through its name instead its index. The instantiation is done within a block.

Example,

```
var t={};
```

Optionally we can init structure with some values.

Example,

```
var t={
        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 '.' Followed by the field name.

Example,

```
print("v:"+t.i); // v:1
print("v:"+t.s); // v:this is a string
print("v:"+t.b); // v:true
print("v:"+t.f); // v:2.0
```

Runtime Management

Struct type It has the following functions in order to manage struct at runtime.

Function	Description	Example
`size`	It returns the number of current elements	`var v=[1,2];` `v.size(); // =2`
`add`	Adds an attribute	`var s={};` `v.add("a",0); // s={a:0}`
`remove`	Remove attribute.	`var s={a:0};` `v.remove("a"); // s={}`

3.5.9 Function object

A function is an object that holds information about a function and is able to call it (see section 3.8.3 for more information)

Example,

```
function add(op1, op2){ // function that returns the sum of two vars.
	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
```

Another example by anonymous function,

```
var fun_obj = function (op1, op2){ // function object that returns the sum of two vars.
	return op1+op2;
};
var j=fun_obj(2,3);// calls fun_obj (aka anonymous function). function. J=5
```

3.6 Operations

ZetScript has the following type of expressions

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

3.6.1 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`

3.6.2 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 int; //= true` `"hello" instanceof int;//= false`

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

3.6.3 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`

3.6.4 Binary expressions

Binary expressions 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`

3.6.5 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
```

3.7 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

3.7.1 The if statement

Use the if statement 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) {
    print("n < 10");
}
```

3.7.2 The else statement

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

Syntax,

```
if(n < 10) {
    print("n < 10");
}else{
    print("n >= 10");
}
```

3.7.3 The 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) {
    print("var n < 10");
}else if(n < 20){
    print("n < 20");
}else{
    print("n >= 20");

}
```

3.7.4 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
```

3.7.5 Switch

Use the switch statement to select one of many blocks of code to be executed.

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

Example,

```
switch (n) {// ← n is a random integer.

    case 6:
       print("6");
       break;
    case 0:
       print("0");
       break;
    default:
       print("n:"+n);
       break;
}
```

Switch can have common code blocks in different conditions

Example,

```
switch (n) { // ← n is a random integer.
    case 4:
    case 5:
        print("4 or 5");
        break;
    case 0:
    case 6:
        print("0 or 6");
        break;
    default:
        print("default");
}
```

3.8 Loops

ZetScript supports the following loop types,

- While Loop
- For Loop

3.8.1The While Loop

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

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

Example,

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

Also it can support Do-While Statement that will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

```
do{

// do-while body

} while (condition);
```

Example,

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

3.8.2 The For Loop

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);
}
```

3.8.3 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.

3.8.4 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;
}
```

3.8.5 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
```

3.8.6 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
```

3.9 Class

A class is a type of structure that contains variables and functions that operates with this 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{
  // member variable
  var data1;

  // member function
  function function1 (a){
    this.data1 =a;
    print("function1:"+this.data1);
    return this.data1;
  }

  // static function that performs an add operation between two Test type objects
  function add_test (a,b){
    return a.data1+b.data1;
  }
};
```

3.9.1 Post add function/variable member

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

Example,

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

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

3.9.2 Instance class

To instance a class is done through the keyword **new**

Example,

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

3.9.3 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
print("data1 is: "+t.data1); // prints value of data1
```

3.9.4 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{
  var data1;

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

var t = new Test(); // Instantiate t as Test type. Now, member variables are instanced.
print("data1:"+t.data1); // ← prints "data1: 10"
```

3.9.5 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{
  var data3;
  function function3(){
    this.data3=this.data1+this.function1(10);
  }
};
```

3.9.5.1 Call parent functions (super keyword)

The extended class can call parent functions through *super* keyword.

Example,

```
class TestExtended: Test{
  var data3;
  function function1(a){
    var t=super(a); // it calls Test::function1(2)
    this.data1+=t; // ← Now data1=5+2 = 7
    print("ext function1:"+this.data1);
    return this.data1+a;
  }

  function function3(){
    this.data3=this.data1+this.function1(5);
    print("ext function3:"+this.data3);

  }
};
```

4 ZetScript API

4.1 Bind C variables

ZetScript supports binding for basic types as int, float, bool, char * and string types. Is also possible to pass custom registered classes (see section 4.3)

Bind C variable is done through macro *register_C_Variable* .You have to provide the name that will be referenced in script side.

Sintax,

```
register_C_Variable("variable_name", c_variable);
```

Example,

```
#include "CZetScript.h"
using namespace zetscript;

int main(int argc, char *argv[]){

  CZetScript *zs = CZetScript::getInstance(); // instance zetscript
  int int_var=0;
  register_C_Variable("int_var",int_var); // registers int_var named int_var at script
side.

  return 0;
}
```

At script side, to have access on the registered c++ variable, just make reference to it. Unlike script variables, C variables is strong type and it doesn't allow to assign value that differs from its type.

Example,

```
int_var =0; //← ok, my_var is int and you assign a int.
int_var =10.0; //← ok, my_var is int it does a cast from float to int.
int_var = "this is a string"; // ← invalid due cannot convert string to int.
```

4.2 Bind C Function

ZetScript C function binding supports int, *float, *bool, char * and string* as arguments types. For return values it supports the same basic types as arguments plus **bool** and **float**. Is also possible to pass custom registered types (see section 4.3)

Is important to say that ZetScript has a constraint of a maximum of 6 parameters for any C function to be bind in the script engine. If a function more than 6 parameters is tried to be registered, ZetScript will throw an error. The value of 6 is not taken by some performance reason or any restriction. It can modify this constraint in ZetScript to have more than 6 parameters, but it will depend throughout the time by what user really needs.

To bind C function is done through macro *register_C_F*unction. You have to provide the name that will be referenced in script side.

Sintax,

```
register_C_Variable("function_name", c_function);
```

Example,

```
#include "CZetScript.h"

using namespace zetscript;

int add(int  op1, int op2){
  return op1+op2;
}

int main(int argc, char *argv[]){
  CZetScript *zs = CZetScript::getInstance(); // instance zetscript

  register_C_Function("add",add);
  return 0;

}
```

4.2.1 Ambiguities

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

For instance,

```
#include "CZetScript.h"

using namespace zetscript;

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

float add(float  *op1, float *op2){ // a function that adds two floats .return op1+op2;
  return *op1+*op2;
}

int main(int argc, char *argv[]){
  CZetScript *zs = CZetScript::getInstance(); // instance zetscript

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

  zs->eval(
       "print(\"result 5+4:\"+add(5,4));"       // prints "result 5+4:9"
  );
  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 "CZetScript.h"

using namespace zetscript;

int add(int  op1, int op2){
  return op1+op2;
}

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

```
int main(){
  CZetScript *zs = CZetScript::getInstance(); // instance zetscript

  // register function add(int,int)
  register_C_Function("add",static_cast<int (*)(int,int)>(add));

  // register function add(float *,float *)
  register_C_Function("add",static_cast<float (*)(float *,float *)>(add));

    zs->eval(
    "print(\"result 5+4:\"+add(5,4));"          // prints "result 5+4:9"
    "print(\"result 0.5+4.6:\"+add(0.5,4.6));"  // prints "result 5.1"
  );
  return 0;

}
```

4.3 Bind C Class

To bind a C Class is done through *register_C_Class* passing the type as template and the name that will be referenced in the script.

Example,

```
register_C_Class<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

MyClass is registered as follow,

```
register_C_Class<MyClass>("MyClass"); //register MyClass as MyClass in script side.
```

4.3.1 Instance C Class

To instance registered C Class it can done through keyword **new**, as it's shown in the following example,

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

4.3.2 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 c 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.
```

4.3.3 Bind Variable Member

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

Sintax,

```
register_C_VariableMember<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);
```

4.3.4 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

```
register_C_FunctionMember<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"
```

4.3.5 Bind custom class 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 constructor[3],

```
register_C_FunctionMember<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,

```
class_C_baseof<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);
  }
};


register_C_Class<MyClassExtend>("MyClassExtend"); // register MyClassExtend

class_C_baseof<MyClassExtend,MyClass>();
```

List 4.2

[3] Note that the name of the function is the same as the name of the class.

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 parameter[4]. 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


	// instance function delete_test function.
	std::function<void()> * delete_test=bind_function<void()>("delete_test");

	// instance member function test.function1.
	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

5 Metamethods

Metamethods are special functions members that links with operators seen on section section 3.6. ZetScript metamethods can be static or member function[5] 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

[5] 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.

5.1 _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;
}
```

5.2 _nequ (aka !=)

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

Script Example

Example how to use *_nequ* metamethod within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _nequ(op1, op2){
    return op1.num!=op2.num;
  }
};
var n1 = new MyNumber (1), n2=new MyNumber (0);
if(n1!=n2){
  print("n1 ("+n1.num+") is not 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 _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;
}
```

5.3 _lt (aka <)

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

Script Example

Example how to use *_lt* metamethod within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _lt(op1, op2){
    return op1.num<op2.num;
  }
};
var n1 = new MyNumber (0), n2=new MyNumber (1);
if(n1<n2){
  print("n1 ("+n1.num+") is less than 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 _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;
}
```

5.4 _lte (aka <=)

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

Script Example

Example how to use *_lte* metamethod within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _lte(op1, op2){
    return op1.num<=op2.num;
  }
};
var n1 = new MyNumber (1), n2=new MyNumber (1);
if(n1<=n2){
  print("n1 ("+n1.num+") is less equal than 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 _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;
}
```

5.5 _gt (aka >)

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

Script Example

Example how to use *_gt* metamethod within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _gt(op1, op2){
    return op1.num>op2.num;
  }
};
var n1 = new MyNumber (1), n2=new MyNumber (0);
if(n1>n2){
  print("n1 ("+n1.num+") is greater than 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 _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;
}
```

5.6 _gte (aka >=)

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

Script Example

Example how to use *_gte* metamethod within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _gte(op1, op2){
    return op1.num>=op2.num;
  }
};
var n1 = new MyNumber (1), n2=new MyNumber (1);
if(n1>=n2){
  print("n1 ("+n1.num+") is greater equal than 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 _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;
}
```

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.

Script Example

Example how to use *_not* metamethod within script class,

```
class MyBoolean{
  var b;

  function MyBoolean(_b){
    this.b=_b;
  }

  function _not(_op){
    return !_op.b;
  }
};
var b = new MyBoolean (false);
if(!b){
  print("b was false");
}
```

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 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;
}
```

5.8 _neg (aka -)

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

Script Example

Example how to use *_neg* metamethod within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _neg(op1){
    return new MyNumber(-op1.num);
  }
};
var n1 = new MyNumber (1);
var n2 = -n1;
print("neg of n1 ("+n1.num+") is ("+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;
  }

  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;
}
```

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.

Script Example

Example how to use *_add* metamethod within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _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);
```

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 * _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;
}
```

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.

Script Example

Example how to use metamethod *_div* within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _div(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);
```

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  *_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;
}
```

*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.

Script Example

Example how to use *_mul* metamethod within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _mul(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);
```

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  *_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;
}
```

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.

Script Example

Example how to use *_mod* metamethod within script class,

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

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


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

  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;
}
```

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.

Script Example

Example how to use *_and* metamethod within script class,

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

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


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

  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;
}
```

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.

Script Example

Example how to use *_or* metamethod within script class,

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

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


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

  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;
}
```

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.

Script Example

Example how to use *_xor* metamethod within script class,

```
class MyNumber{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _xor(op1,op2){
    return new MyNumber(op1.num^op2.num);
  }
};

var n1 = new MyNumber (0xf1);
var n2 = new MyNumber (0x0f);
var n3 =n1^n2;


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

  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;
}
```

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.

Script Example

Example how to use ***_shl*** metamethod within script class,

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

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


print("n1 ("+n1.num+") << 3 = "+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;
  }

  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;
}
```

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.

Script Example

Example how to use *_shr* metamethod within script class,

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

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


print("n1 ("+n1.num+") >> 2 = "+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;
  }

  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;
}
```

5.19 _set (aka =)

@Description: Performs a set operation[6].
@Param1 : Source variable to set.
@Returns : None.

Script Example

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{
  var num;
  function MyNumber(_n){
    this.num=_n;
  }
  function _set(v){
    if(v instanceof int){
       this.num = v;
    }else if(v instanceof MyNumber){
       this.num = v.num;
    }else{
       error("parameter not supported");
    }
  }
};

var n1 = new MyNumber (10);
var n2 = new MyNumber (20);
var n3; // ← n3 is undefined!

n3 = n2; // ← it assigns n2 pointer.
print("n3:"+n3.num);
n3=n1; // ← n3.num = n2.num = n1.num.
print("n3:"+n3.num);
n3=50; // ← n3.num = n2.num = 10.
print("n3:"+n3.num);
n3=false; // ← stops execution with error "parameter not supported".
```

[6] 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).

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;
  }

  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;
}
```

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.

Example,

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

    var aux1, aux2;
    if(op1 instanceof MyNumber){
      aux1=op1.num;
    }else if(op1 instanceof int || op1 instanceof number){
      aux1=op1;
    }else{
      error("arg op1 is not supported");
    }

    if(op2 instanceof MyNumber){
      aux2=op2.num;
    }else if(op2 instanceof int || op2 instanceof number){
      aux2=op2;
    }else{
      error("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
```

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;
}
```