Table of Contents

Chapter 1. The language

This part of the document describes the syntax and semantics of the language.

Lexical structure

Identifiers

Identifiers start with a alphabetic character or '_' followed by any number of alphabetic characters, '_' or digits ([0-9]). Squirrel is a case sensitive language, this means that the lowercase and uppercase representation of the same alphabetic character are considered different characters. For instance "foo", "Foo" and "fOo" will be treated as 3 distinct identifiers.

```
id:= [a-zA-Z_]+[a-zA-Z_0-9]*
```

Keywords

The following words are reserved words by the language and cannot be used as identifiers:

```
break       case        catch       class       clone       continue
const       default     delegate    delete      else        enum
extends     for         function    if          in          local
null        resume      return      switch      this        throw
try         typeof      while       parent      yield       constructor
vargc       vargv       instanceof  true        false       static
```

Keywords are covered in detail later in this document.

Operators

Squirrel recognizes the following operators:

```
!       !=      ||      ==      &&      <=      =>      >
+       +=      -       -=      /       /=      *       *=
%       %=      ++      --      <-      =       &       ^
|       ~       >>      <<      >>>
```

Other tokens

Other used tokens are:

```
{     }     [     ]     .     :     ::    '     ;     "     @"
```

Literals

Squirrel accepts integer numbers, floating point numbers and stings literals.

`34`	Integer number(base 10)
`0xFF00A120`	Integer number(base 16)
`0753`	Integer number(base 8)
`'a'`	Integer number
`1.52`	Floating point number
`1.e2`	Floating point number
`1.e-2`	Floating point number
`"I'm a string"`	String
`@"I'm a verbatim string"`	String
`@" I'm a multiline verbatim string "`	String

```
IntegerLiteral := [0-9]+ | '0x' [0-9A-Fa-f]+ | ''' [.]+ ''' | 0[0-7]+
FloatLiteral := [0-9]+ '.' [0-9]+
FloatLiteral := [0-9]+ '.' 'e'|'E' '+'|'-' [0-9]+
StringLiteral:= '"'[.]* '"'
VerbatimStringLiteral:= '@'"'[.]* '"'
```

Comments

A comment is text that the compiler ignores but that is useful for programmers. Comments are normally used to embed annotations in the code. The compiler treats them as white space.

The `/*` (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the `*/` characters. This syntax is the same as ANSI C.

```
/*
this is
a multiline comment.
this lines will be ignored by the compiler
*/
```

The `//` (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. It is commonly called a "`single-line comment.`"

```
//this is a single line comment. this line will be ignored by the compiler
```

Values and Data types

Squirrel is a dynamically typed language so variables do not have a type, although they refer to a value

that does have a type. Squirrel basic types are integer, float, string, null, table, array, function, generator, class, instance, bool, thread and userdata.

Integer

An Integer represents a 32 bits (or better) signed number.

```
local a = 123 //decimal
local b = 0x0012 //hexadecimal
local c = 075 //octal
local d = 'w' //char code
```

Float

A float represents a 32 bits (or better) floating point number.

```
local a=1.0
local b=0.234
```

String

Strings are an immutable sequence of characters to modify a string is necessary create a new one.

Squirrel's strings, behave like C or C++, are delimited by quotation marks(") and can contain escape sequences(\t,\a,\b,\n,\r,\v,\f,\\,\",\',\0,\x*hhhh*).

Verbatim string literals begin with @" and end with the matching quote. Verbatim string literals also can extend over a line break. If they do, they include any white space characters between the quotes:

```
local a = "I'm a wonderful string\n"
// has a newline at the end of the string
local x = @"I'm a verbatim string\n"
// the \n is copied in the string same as \\n in a regular string "I'm a verbatim
```

The only exception to the "no escape sequence" rule for verbatim string literals is that you can put a double quotation mark inside a verbatim string by doubling it:

```
local multiline = @"
        this is a multiline string
        it will ""embed"" all the new line
        characters
"
```

Null

The null value is a primitive value that represents the null, empty, or non-existent reference. The type Null has exactly one value, called null.

```
local a=null
```

Bool

the bool data type can have only two. They are the literals `true` and `false`. A bool value expresses the validity of a condition (tells whether the condition is true or false).

```
local a = true;
```

Table

Tables are associative containers implemented as pairs of key/value (called a slot).

```
local t={}
local test=
{
    a=10
    b=function(a) { return a+1; }
}
```

Array

Arrays are simple sequence of objects, their size is dynamic and their index starts always from 0.

```
local a=["I'm","an","array"]
local b=[null]
b[0]=a[2];
```

Function

Functions are similar to those in other C-like languages and to most programming languages in general, however there are a few key differences (see below).

Class

Classes are associative containers implemented as pairs of key/value. Classes are created through a 'class expression' or a 'class statement'. class members can be inherited from another class object at creation time. After creation members can be added until a instance of the class is created.

Class instance

Class instances are created by calling a `class object`. Instances, as tables, are implemented as pair of key/value. Instances members cannot be dyncamically added or removed however the value of the members can be changed.

Generator

Generators are functions that can be suspended with the statement 'yield' and resumed later (see Generators).

Userdata

Userdata objects are blobs of memory(or pointers) defined by the host application but stored into Squirrel variables (See Userdata and UserPointers).

Thread

Threads are objects that represents a cooperative thread of execution, also known as coroutines.

Weak References

Weak References are objects that point to another(non scalar) object but do not own a strong reference to it. (See Weak References).

Execution Context

The execution context is the union of the function stack frame and the function environment object(this). The stack frame is the portion of stack where the local variables declared in its body are stored. The environment object is an implicit parameter that is automatically passed by the function caller (see Functions). During the execution, the body of a function can only transparently refer to his execution context. This mean that a single identifier can refer either to a local variable or to an environment object slot; Global variables require a special syntax (see Variables). The environment object can be explicitly accessed by the keyword this.

Variables

There are two types of variables in Squirrel, local variables and tables/arrays slots. Because global variables are stored in a table, they are table slots.

A single identifier refers to a local variable or a slot in the environment object.

```
derefexp := id;
```

```
_table["foo"]
_array[10]
```

with tables we can also use the '.' syntax

```
derefexp := exp '.' id
```

```
_table.foo
```

Squirrel first checks if an identifier is a local variable (function arguments are local variables) if not it checks if it is a member of the environment object (this).

For instance:

```
function testy(arg)
{
    local a=10;
    print(a);
    return arg;
}
```

will access to local variable 'a' and prints 10.

```
function testy(arg)
{
    local a=10;
    return arg+foo;
}
```

in this case 'foo' will be equivalent to 'this.foo' or this["foo"].

Global variables are stored in a table called the root table. Usually in the global scope the environment object is the root table, but to explicitly access the global table from another scope, the slot name must be prefixed with '::' (::foo).

```
exp:= '::' id
```

For instance:

```
function testy(arg)
{
    local a=10;
    return arg+::foo;
}
```

accesses the global variable 'foo'.

However (since squirrel 2.0) if a variable is not local and is not found in the 'this' object Squirrel will search it in the root table.

```
function test() {
        foo = 10;
}
```

is equivalent to write

```
function test() {
        if("foo" in this) {
                this.foo = 10;
        }else {
                ::foo = 10;
        }
}
```

Statements

A squirrel program is a simple sequence of statements.

```
stats := stat [';'|'\n'] stats
```

Statements in squirrel are comparable to the C-Family languages (C/C++, Java, C# etc...): assignment, function calls, program flow control structures etc.. plus some custom statement like yield, table and array constructors (All those will be covered in detail later in this document). Statements can be separated with a new line or ';' (or with the keywords case or default if inside a switch/case statement), both symbols are not required if the statement is followed by '}'.

Block

```
stat := '{' stats '}'
```

A sequence of statements delimited by curly brackets ({ }) is called block; a block is a statement itself.

Control Flow Statements

squirrel implements the most common control flow statements: `if`, `while`, `do-while`, `switch-case`, `for`, `foreach`.

true and false

Squirrel has a boolean type(bool) however like C++ it considers null, 0(integer) and 0.0(float) as *false*, any other value is considered *true*.

if/else

```
stat:= 'if' '(' exp ')' stat ['else' stat]
```

Conditionally execute a statement depending on the result of an expression.

```
if(a>b)
    a=b;
else
    b=a;
////
if(a==10)
{
    b=a+b;
    return a;
}
```

while

```
stat:= 'while' '(' exp ')' stat
```

Executes a statement until the condition is false.

```
function testy(n)
{
    local a=0;
    while(a<n) a+=1;

        while(1)
        {
        if(a<0) break;
        a-=1;
    }
}
```

do/while

stat:= 'do' stat 'while' '(' expression ')'

Executes a statement once, and then repeats execution of the statement until a condition expression evaluates to false.

```
local a=0;
do
{
    print(a+"\n");
    a+=1;
} while(a>100)
```

switch

```
stat := 'switch' ''( exp ')' '{'
        'case' case_exp ':'
                stats
        ['default' ':'
                stats]
'}'
```

Is a control statement allows multiple selections of code by passing control to one of the case statements within its body. The control is transferred to the case label whose case_exp matches with exp if none of the case match will jump to the default label (if present). A switch statement can contain any number if case instances, if 2 case have the same expression result the first one will be taken in account first. The default label is only allowed once and must be the last one. A break statement will jump outside the switch block.

Loops

for

stat:= 'for' '(' [initexp] ';' [condexp] ';' [incexp] ')' statement

Executes a statement as long as a condition is different than false.

```
for(local a=0;a<10;a+=1)
    print(a+"\n");
//or
glob <- null
for(glob=0;glob<10;glob+=1){
    print(glob+"\n");
}
//or
for(;;){
    print(loops forever+"\n");
}
```

foreach

'foreach' '(' [index_id','] value_id 'in' exp ')' stat

Executes a statement for every element contained in an array, table, class, string or generator. If exp is a generator it will be resumed every iteration as long as it is alive; the value will be the result of 'resume' and the index the sequence number of the iteration starting from 0.

```
local a=[10,23,33,41,589,56]
foreach(idx,val in a)
    print("index="+idx+" value="+val+"\n");
//or
foreach(val in a)
    print("value="+val+"\n");
```

break

stat := 'break'

The break statement terminates the execution of a loop (for, foreach, while or do/while) or jumps out of switch statement;

continue

stat := 'continue'

The continue operator jumps to the next iteration of the loop skipping the execution of the following statements.

return

stat:= return [exp]

The return statement terminates the execution of the current function/generator and optionally returns the result of an expression. If the expression is omitted the function will return null. If the return statement is used inside a generator, the generator will not be resumable anymore.

yield

```
stat := yield [exp]
```

(see Generators).

Local variables declaration

```
initz := id [= exp][',' initz]
stat := 'local' initz
```

Local variables can be declared at any point in the program; they exist between their declaration to the end of the block where they have been declared. EXCEPTION: a local declaration statement is allowed as first expression in a for loop.

```
for(local a=0;a<10;a+=1)
    print(a);
```

Function declaration

```
funcname := id ['::' id]
stat:= 'function' id ['::' id]+ '(' args ')'[':' '(' args ')'] stat
```

creates a new function.

Class declaration

```
memberdecl := id '=' exp [';'] | '[' exp ']' '=' exp [';'] | functionstat | 'const
stat:= 'class' derefexp ['extends' derefexp] '{'
                        [memberdecl]
                '}'
```

creates a new class.

try/catch

```
stat:= 'try' stat 'catch' '(' id ')' stat
```

The try statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a throw statement. The catch clause provides the exceptionhandling code. When a catch

clause catches an exception, its id is bound to that exception.

throw

```
stat:= 'throw' exp
```

Throws an exception. Any value can be thrown.

const

```
stat:= 'const' id '=' 'Integer | Float | StringLiteral
```

Declares a constant (see Constants & Enumerations).

enum

```
enumerations := ( ‘id’ '=' Integer | Float | StringLiteral ) [‘,’]
stat:= 'enum' id '{' enumerations '}'
```

Declares an enumeration (see Constants & Enumerations).

expression statement

```
stat := exp
```

In Squirrel every expression is also allowed as statement, if so, the result of the expression is thrown away.

Expressions

Assignment(=) & new slot(<-)

```
exp := derefexp '=' exp
exp:= derefexp '<-' exp
```

squirrel implements 2 kind of assignment: the normal assignment(=)

```
a=10;
```

and the "new slot" assignment.

```
a <- 10;
```

The new slot expression allows to add a new slot into a table(see Tables). If the slot already exists in the table it behaves like a normal assignment.

Operators

?: Operator

```
exp := exp_cond '?' exp1 ':' exp2
```

conditionally evaluate an expression depending on the result of an expression.

Arithmetic

```
exp:= 'exp' op 'exp'
```

Squirrel supports the standard arithmetic operators +, -, *, / and %. Other than that is also supports compact operators (+=,-=,*=,/=,%=) and increment and decrement operators(++ and --);

```
a+=2;
//is the same as writing
a=a+2;
x++
//is the same as writing
x=x+1
```

All operators work normally with integers and floats; if one operand is an integer and one is a float the result of the expression will be float. The + operator has a special behavior with strings; if one of the operands is a string the operator + will try to convert the other operand to string as well and concatenate both together. For instances and tables, _tostring is invoked.

Relational

```
exp:= 'exp' op 'exp'
```

Relational operators in Squirrel are : == < <= > >= !=

These operators return null if the expression is false and a value different than null if the expression is true. Internally the VM uses the integer 1 as true but this could change in the future.

Logical

```
exp := exp op exp
exp := '!' exp
```

Logical operators in Squirrel are : && || !

The operator && (logical and) returns null if its first argument is null, otherwise returns its second argument. The operator || (logical or) returns its first argument if is different than null, otherwise returns the second argument.

The '!' operator will return null if the given value to negate was different than null, or a value different than null if the given value was null.

in operator

exp:= keyexp 'in' tableexp

Tests the existence of a slot in a table. Returns a value different than null if keyexp is a valid key in tableexp

```
local t=
{
    foo="I'm foo",
    [123]="I'm not foo"
}

if("foo" in t) dostuff("yep");
if(123 in t) dostuff();
```

instanceof operator

exp:= instanceexp 'instanceof' classexp

Tests if a class instance is an instance of a certain class. Returns a value different than null if instanceexp is an instance of classexp.

typeof operator

exp:= 'typeof' exp

returns the type name of a value as string.

```
local a={},b="squirrel"
print(typeof a); //will print "table"
print(typeof b); //will print "string"
```

comma operator

exp:= exp ',' exp

The comma operator evaluates two expression left to right, the result of the operator is the result of the expression on the right; the result of the left expression is discarded.

```
local j=0,k=0;
for(local i=0; i<10; i++ , j++)
{
        k = i + j;
}
```

```
local a,k;
a = (k=1,k+2); //a becomes 3
```

Bitwise Operators

```
exp:= 'exp' op 'exp'
exp := '~' exp
```

Squirrel supports the standard c-like bit wise operators &,|,^,~,<<,>> plus the unsigned right shift operator >>>. The unsigned right shift works exactly like the normal right shift operator(>>) except for treating the left operand as an unsigned integer, so is not affected by the sign. Those operators only work on integers values, passing of any other operand type to these operators will cause an exception.

Operators precedence

-,~,!,typeof ,++,--	highest
/, *, %	...
+, -	
<<, >>,>>>	
<, <=, >, >=	
==, !=	
&	
^	
|	
&&, in	
||	
?:	
+=,=,-=	...
,(comma operator)	lowest

Table constructor

```
tslots := ( 'id' '=' exp | '[' exp ']' '=' exp ) [',']
exp := '{' [tslots] '}'
```

Creates a new table.

```
local a={} //create an empty table
```

A table constructor can also contain slots declaration; With the syntax:

id = exp [',']

a new slot with id as key and exp as value is created

```
local a=
{
    slot1="I'm the slot value"
}
```

An alternative syntax can be

'[' exp1 ']' = exp2 [',']

A new slot with exp1 as key and exp2 as value is created

```
local a=
{
    [1]="I'm the value"
}
```

both syntaxes can be mixed

```
local table=
{
    a=10,
    b="string",
    [10]={},
    function bau(a,b)
    {
        return a+b;
    }
}
```

The comma between slots is optional.

delegate

exp:= ‘delegate’ parentexp : exp

Sets the parent of a table. The result of parentexp is set as parent of the result of exp, the result of the expression is exp (see Delegation).

clone

exp:= ‘clone’ exp

Clone performs shallow copy of a table, array or class instance (copies all slots in the new object without recursion). If the source table has a delegate, the same delegate will be assigned as delegate (not

copied) to the new table (see Delegation).

After the new object is ready the “_cloned” meta method is called (see Metamethods).

When a class instance is cloned the constructor is not invoked(initializations must rely on `_cloned` instead

Array constructor

exp := ‘[’ [explist] ‘]’

Creates a new array.

```
a <- [] //creates an empty array
```

arrays can be initialized with values during the construction

```
a <- [1,"string!",[],{}] //creates an array with 4 elements
```

Tables

Tables are associative containers implemented as pairs of key/value (called slot); values can be any possible type and keys any type except 'null'. Tables are squirrel's skeleton, delegation and many other features are all implemented through this type; even the environment, where global variables are stored, is a table (known as root table).

Construction

Tables are created through the table constructor (see Table constructor)

Slot creation

Adding a new slot in a existing table is done through the "new slot" operator '<-'; this operator behaves like a normal assignment except that if the slot does not exists it will be created.

```
local a={}
```

The following line will cause an exception because the slot named 'newslot' does not exist in the table ‘a’

```
a.newslot = 1234
```

this will succeed:

```
a.newslot <- 1234;
```

or

```
a[1] <- "I'm the value of the new slot";
```

Slot deletion

exp:= delete derefexp

Deletion of a slot is done through the keyword delete; the result of this expression will be the value of the deleted slot.

```
a <- {
    test1=1234
    deleteme="now"
}

delete a.test1
print(delete a.deleteme); //this will print the string "now"
```

Arrays

An array is a sequence of values indexed by a integer number from 0 to the size of the array minus 1. Arrays elements can be obtained through their index.

```
local a=[“I’m a string”, 123]
print(typeof a[0]) //prints "string"
print(typeof a[1]) //prints "integer"
```

Resizing, insertion, deletion of arrays and arrays elements is done through a set of standard functions (see built-in functions).

Functions

Functions are first class values like integer or strings and can be stored in table slots, local variables, arrays and passed as function parameters. Functions can be implemented in Squirrel or in a native language with calling conventions compatible with ANSI C.

Function declaration

Functions are declared through the function expression

```
local a= function(a,b,c) {return a+b-c;}
```

or with the syntactic sugar

```
function ciao(a,b,c)
{
    return a+b-c;
}
```

that is equivalent to

```
this.ciao <- function(a,b)
{
    return a+b-c;
}
```

is also possible to declare something like

```
T <- {}
function T::ciao(a,b,c)
{
    return a+b-c;
}

//that is equivalent to write

T.ciao <- function(a,b,c)
{
    return a+b-c;
}

//or

T <- {
        function ciao(a,b,c)
        {
                return a+b-c;
        }
}
```

Default Paramaters

Squirrel's functions can have default parameters.

A function with default parameters is declared as follows:

```
function test(a,b,c = 10, d = 20)
{
        ....
}
```

when the function `test` is invoked and the parameter c or d are not specified, the VM autometically as-

signs the default value to the unspecified parameter. A default parameter can be any valid squirrel expression. The expression is evaluated at runtime.

Function with variable number of paramaters

Squirrel's functions can have variable number of parameters(varargs functions).

A vararg function is declared by adding three dots (`...´) at the end of its parameter list.

When the function is called all the extra parameters will be accessible through the *pseudo array* called `vargv`.

`vargv` can only indexed with a numeric object(float or integer). The number of parameter contained in `vargv` is stored in the *pseudo variable* `vargc`.
Note that vargv is not a real object, it can't be assigned or passed as parameter.

```
function test(a,b,...)
{
        for(local i = 0; i< vargc; i++)
        {
                ::print("varparam "+i+" = "+vargv[i]+"\n");
        }
}

test("goes in a","goes in b",0,1,2,3,4,5,6,7,8);
```

Function calls

exp:= derefexp ‘(‘ explist ‘)’

The expression is evaluated in this order: derefexp after the explist (arguments) and at the end the call.

Every function call in Squirrel passes the environment object ‘this’ as hidden parameter to the called function. The ‘this’ parameter is the object where the function was indexed from.

If we call a function with this syntax

```
table.foo(a)
```

the environment object passed to foo will be ‘table’

```
foo(x,y) // equivalent to this.foo(x,y)
```

The environment object will be ‘this’ (the same of the caller function).

Binding an environment to a function

while by default a squirrel function call passes as environment object 'this', the object where the function was indexed from. However, is also possible to statically bind an evironment to a closure using the built-in method `closure.bindenv(env_obj)`. The method bindenv() returns a new instance of a closure with the environment bound to it. When an environment object is bound to a function, every time the

function is invoked, its 'this' parameter will always be the previously bound environent. This mechanism is useful to implement callbacks systems similar to C# delegates.

Note

The closure keeps a weak reference to the bound environmet object, because of this if the object is deleted, the next call to the closure will result in a `null` environment object.

Free variables

Free variables are variables referenced by a function that are not visible in the function scope. In the following example the function foo() declares x, y and testy as free variables.

```
local x=10,y=20
testy <- “I’m testy”

function foo(a,b):(x,y,testy)
{
    ::print(testy);
    return a+b+x+y;
}
```

The value of a free variable is frozen and bound to the function when the function is created; the value is passed to the function as implicit parameter every time is called.

Tail recursion

Tail recursion is a method for partially transforming a recursion in a program into an iteration: it applies when the recursive calls in a function are the last executed statements in that function (just before the return). If this happenes the squirrel interpreter collapses the caller stack frame before the recursive call; because of that very deep recursions are possible without risk of a stack overflow.

```
function loopy(n)
{
    if(n>0){
        ::print(“n=”+n+”\n”);
        return loopy(n-1);
    }
}

loopy(1000);
```

Classes

Squirrel implements a class mechanism similar to languages like Java/C++/etc... however because of its dynamic nature it differs in several aspects. Classes are first class objects like integer or strings and can be stored in table slots local variables, arrays and passed as function parameters.

Class declaration

A class object is created through the keyword 'class' . The class object follows the same declaration syntax of a table(see tables) with the only difference of using ';' as optional separator rather than ','.

For instance:

```
class Foo {
        //constructor
        constructor(a)
        {
                testy = ["stuff",1,2,3];
        }
        //member function
        function PrintTesty()
        {
                foreach(i,val in testy)
                {
                        ::print("idx = "+i+" = "+val+" \n");
                }
        }
        //property
        testy = null;

}
```

the previous code examples is a syntactic sugar for:

```
Foo <- class {
        //constructor
        constructor(a)
        {
                testy = ["stuff",1,2,3];
                testy = a;
        }
        //member function
        function PrintTesty()
        {
                foreach(i,val in testy)
                {
                        ::print("idx = "+i+" = "+val+" \n");
                }
        }
        //property
        testy = null;

}
```

in order to emulate namespaces, is also possible to declare something like this

```
//just 2 regular nested tables
FakeNamespace <- {
        Utils = {}
}

class FakeNamespace.Utils.SuperClass {
        constructor()
        {
                ::print("FakeNamespace.Utils.SuperClass")
        }
        function DoSomething()
        {
                ::print("DoSomething()")
        }
}
```

```
function FakeNamespace::Utils::SuperClass::DoSomethingElse()
{
        ::print("FakeNamespace::Utils::SuperClass::DoSomethingElse()")
}

local testy = FakeNamespace.Utils.SuperClass();
testy.DoSomething();
testy.DoSomethingElse();
```

After its declaration, methods or properties can be added or modified by following the same rules that apply to a table(operator `<-` and `=`).

```
//adds a new property
Foo.stuff <- 10;

//modifies the default value of an existing property
Foo.testy = "I'm a string";

//adds a new method
function Foo::DoSomething(a,b)
{
        return a+b;
}
```

After a class is instantiated is no longer possible to add new properties or methods to it.

Static variables

Squirrel's classes support static member variables. A static variable shares its value between all instances of the class. Statics are declared by prefixing the variable declaration with the keyword `static`; the declaration must be in the class body.

Note

Statics are read-only.

```
class Foo {
        constructor()
        {
                //..stuff
        }
        name = "normal variable";
        //static variable
        static classname = "The class name is foo";
};
```

Class attributes

Classes allow to associate attributes to it's members. Attributes are a form of metadata that can be used to store application specific informations, like documentations strings, properties for IDEs, code generators etc... Class attributes are declared in the class body by preceding the member declaration and are delimited by the symbol `</` and `/>`. Here an example:

```
class Foo </ test = "I'm a class level attribute" />{
        </ test = "freakin attribute" /> //attributes of PrintTesty
        function PrintTesty()
        {
                foreach(i,val in testy)
                {
                        ::print("idx = "+i+" = "+val+" \n");
```

```
                }
        }
        </ flippy = 10 , second = [1,2,3] /> //attributes of testy
        testy = null;

}
```

Attributes are, matter of fact, a table. Squirrel uses `</ />` syntax instead of curly brackets `{}` for the attribute declaration to increase readability.

This means that all rules that apply to tables apply to attributes.

Attributes can be retrieved through the built-in function `classobj.getattributes(membername)` (see built-in functions). and can be modified/added through the built-in function `classobj.setattributes(membername,val)`.

the following code iterates through the attributes of all Foo members.

```
foreach(member,val in Foo)
{
        ::print(member+"\n");
        local attr;
        if((attr = Foo.getattributes(member)) != null) {
                foreach(i,v in attr)
                {
                        ::print("\t"+i+" = "+(typeof v)+"\n");
                }
        }
        else {
                ::print("\t<no attributes>\n")
        }
}
```

Class instances

The class objects inherits several of the table's feature with the difference that multiple instances of the same class can be created. A class instance is an object that share the same structure of the table that created it but holds is own values. Class *instantiation* uses function notation. A class instance is created by calling a class object. Can be useful to imagine a class like a function that returns a class instance.

```
//creates a new instance of Foo
local inst = Foo();
```

When a class instance is created its member are initialized with the same value specified in the class declaration.

When a class defines a method called 'constructor', the class instantiation operation will automatically invoke it for the newly created instance. The constructor method can have parameters, this will impact on the number of parameters that the *instantiation operation* will require. Constructors as normal functions can have variable number of parameters (using the parameter `...`).

```
class Rect {
        constructor(w,h)
        {
                width = w;
                height = h;
        }
```

```
        x = 0;
        y = 0;
        width = null;
        height = null;
}

//Rect's constructor has 2 parameters so the class has to be 'called'
//with 2 parameters
local rc = Rect(100,100);
```

After an instance is created, its properties can be set or fetched following the same rules that apply to tables. Methods cannot be set.

Instance members cannot be removed.

The class object that created a certain instance can be retrieved through the built-in function `instance.getclass()`(see built-in functions)
The operator `instanceof` tests if a class instance is an instance of a certain class.

```
local rc = Rect(100,100);
if(rc instanceof ::Rect) {
        ::print("It's a rect");
}
else {
        ::print("It isn't a rect");
}
```

Inheritance

Squirrel's classes support single inheritance by adding the keyword `extends`, followed by an expression, in the class declaration. The syntax for a derived class is the following:

```
class SuperFoo extends Foo {
        function DoSomething() {
                ::print("I'm doing something");
        }
}
```

When a derived class is declared, Squirrel first copies all base's members in the new class then proceeds with evaluating the rest of the declaration.

A derived class inherit all members and properties of it's base, if the derived class overrides a base function the base implementation is shadowed. It's possible to access a overridden method of the base class by fetching the method from the base class object.

Here an example:

```
class Foo {
        function DoSomething() {
                ::print("I'm the base");
        }
};

class SuperFoo extends Foo {
```

```
        //overridden method
        function DoSomething() {
                //calls the base method
                ::Foo.DoSomething();
                ::print("I'm doing something");
        }
}
```

Same rule apply to the constructor. The constructor is a regular function (apart from being automatically invoked on contruction).

```
class Base {
        constructor()
        {
                ::print("Base constructor\n");
        }
}

class Child extends Base {
        constructor()
        {
                ::Base.constructor();
                ::print("Child constructor\n");
        }
}

local test = Child();
```

The base class of a derived class can be retrieved through the keyword `parent`. `parent` is a 'pseudo slot'. The `parent` slot cannot be set.

```
local thebaseclass = SuperFoo.parent;
```

Note that because methods do not have special protection policies when calling methods of the same objects, a method of a base class that calls a method of the same class can end up calling a overridden method of the derived class.

```
class Foo {
        function DoSomething() {
                ::print("I'm the base");
        }
        function DoIt()
        {
                DoSomething();
        }
};

class SuperFoo extends Foo {
        //overridden method
        function DoSomething() {
                ::print("I'm the derived");

        }
        function DoIt() {
                ::Foo.DoIt();
        }
}
```

```
//creates a new instance of SuperFoo
local inst = SuperFoo();

//prints "I'm the derived"
inst.DoIt();
```

Metamethods

Class instances allow the customization of certain aspects of the their semantics through metamethods(see Metamethods). For C++ programmers: "metamethods behave roughly like overloaded operators". The metamethods supported by classes are `_add`, `_sub`, `_mul`, `_div`, `_unm`, `_modulo`, `_set`, `_get`, `_typeof`, `_nexti`, `_cmp`, `_call`, `_delslot`,`_tostring`

Class objects instead support only 2 metamethods : `_newmember and _inherited`
the following example show how to create a class that implements the metamethod `_add`.

```
class Vector3 {
        constructor(...)
        {
                if(vargc >= 3) {
                        x = vargv[0];
                        y = vargv[1];
                        z = vargv[2];
                }
        }
        function _add(other)
        {
                return ::Vector3(x+other.x,y+other.y,z+other.z);
        }

        x = 0;
        y = 0;
        z = 0;
}

local v0 = Vector3(1,2,3)
local v1 = Vector3(11,12,13)
local v2 = v0 + v1;
::print(v2.x+","+v2.y+","+v2.z+"\n");
```

Since version 2.1, classes support 2 metamethods `_inherited` and `_newmember`. `_inherited` is invoked when a class inherits from the one that implements `_inherited`. `_newmember` is invoked for each member that is added to the class(at declaration time).

Generators

A function that contains a yield statement is called ‘generator function’. When a generator function is called, it does not execute the function body, instead it returns a new suspended generator. The returned generator can be resumed through the resume statement while it is alive. The yield keyword, suspends the execution of a generator and optionally returns the result of an expression to the function that resumed the generator. The generator dies when it returns, this can happen through an explicit return statement or by exiting the function body; If an unhandled exception (or runtime error) occurs while a gener-

ator is running, the generator will automatically die. A dead generator cannot be resumed anymore.

```
function geny(n)
{
    for(local i=0;i<n;i+=1)
        yield i;
    return null;
}

local gtor=geny(10);
local x;
while(x=resume gtor) print(x+"\n");
```

the output of this program will be

```
0
1
2
3
4
5
6
7
8
9
```

generators can also be iterated using the foreach statement. When a generator is evaluated by `foreach`, the generator will be resumed for each iteration until it returns. The value returned by the `return` statement will be ignored.

Constants & Enumerations

Squirrel allows to bind constant values to an identifier that will be evaluated compile-time. This is archieved though constants and enumarations.

Constants

Constants bind a specific value to an indentifier. Constants are similar to global values, except that they are evaluated compile time and their value cannot be changed.

constants values can only be integers, floats or string literals. No expression are allowed. are declared with the following syntax.

```
const foobar = 100;
const floatbar = 1.2;
const stringbar = "I'm a contant string";
```

constants are always globally scoped, from the moment they are declared, any following code can reference them. Constants will shadow any global slot with the same name(the global slot will remain visible

by using the :: syntax).

```
local x = foobar * 2;
```

Enumerations

As Constants, Enumerations bind a specific value to a name. Enumerations are also evaluated compile time and their value cannot be changed.

An enum declaration introduces a new enumeration into the program. Enumerations values can only be integers, floats or string literals. No expression are allowed.

```
enum Stuff {
  first, //this will be 0
  second, //this will be 1
  third //this will be 2
}
```

or

```
enum Stuff {
  first = 10
  second = "string"
  third = 1.2
}
```

An enum value is accessed in a manner that's similar to accessing a static class member. The name of the member must be qualified with the name of the enumeration, for example `Stuff.second`. Enumerations will shadow any global slot with the same name(the global slot will remain visible by using the :: syntax).

```
local x = Stuff.first * 2;
```

Implementation notes

Enumerations and Contants are a compile-time feature. Only integers, string and floats can be declared as const/enum; No expressions are allowed(because they would have to be evaluated compile time). When a const or an enum is declared, it is added compile time to the `consttable`. This table is stored in the VM shared state and is shared by the VM and all its threads. The `consttable` is a regular squir-

rel table; In the same way as the `roottable` it can be modified runtime. You can access the `const-table` through the built-in function `getconsttable()` and also change it through the built-in function `setconsttable()`

here some example:

```
//create a constant
getconsttable()["something"] <- 10"
//create an enumeration
getconsttable()["somethingelse"] <- { a = "10", c = "20", d = "200"};
//deletes the constant
delete getconsttable()["something"]
//deletes the enumeration
delete getconsttable()["somethingelse"]
```

This system allows to procedurally declare constants and enumerations, it is also possible to assign any squirrel type to a constant/enumeration(function,classes etc...). However this will make serialization of a code chunk impossible.

Threads

Squirrel supports cooperative threads(also known as coroutines). A cooperative thread is a subroutine that can suspended in mid-execution and provide a value to the caller without returning program flow, then its execution can be resumed later from the same point where it was suspended. At first look a Squirrel thread can be confused with a generator, in fact their behaviour is quite similar. However while a generator runs in the caller stack and can suspend only the local routine stack a thread has its own execution stack, global table and error handler; This allows a thread to suspend nested calls and have it's own error policies.

Using threads

Threads are created through the built-in function 'newthread(func)'; this function gets as parameter a squirrel function and bind it to the new thread objecs(will be the thread body). The returned thread object is initially in 'idle' state. the thread can be started with the function 'threadobj.call()'; the parameters passed to 'call' are passed to the thread function.

A thread can be be suspended calling the function suspend(), when this happens the function that wokeup(or started) the thread returns (If a parametrer is passed to suspend() it will be the return value of the wakeup function , if no parameter is passed the return value will be null). A suspended thread can be resumed calling the funtion 'threadobj.wakeup', when this happens the function that suspended the thread will return(if a parameter is passed to wakeup it will be the return value of the suspend function, if no parameter is passed the return value will be null).

A thread terminates when its main function returns or when an unhandled exception occurs during its execution.

```
function coroutine_test(a,b)
{
        ::print(a+" "+b+"\n");
        local ret = ::suspend("suspend 1");
        ::print("the coroutine says "+ret+"\n");
        ret = ::suspend("suspend 2");
        ::print("the coroutine says "+ret+"\n");
        ret = ::suspend("suspend 3");
```

```
        ::print("the coroutine says "+ret+"\n");
        return "I'm done"
}

local coro = ::newthread(coroutine_test);

local susparam = coro.call("test","coroutine"); //starts the coroutine

local i = 1;
do
{
        ::print("suspend passed ("+susparam+")\n")
        susparam = coro.wakeup("ciao "+i);
        ++i;
}while(coro.getstatus()=="suspended")

::print("return passed ("+susparam+")\n")
```

the result of this program will be

```
test coroutine
suspend passed (suspend 1)
the coroutine says ciao 1
suspend passed (suspend 2)
the coroutine says ciao 2
suspend passed (suspend 3)
the coroutine says ciao 3
return passed (I'm done).
```

the following is an interesting example of how threads and tail recursion can be combined.

```
function state1()
{
        ::suspend("state1");
        return state2(); //tail call
}

function state2()
{
        ::suspend("state2");
        return state3(); //tail call
}

function state3()
{
        ::suspend("state3");
        return state1(); //tail call
}

local statethread = ::newthread(state1)

::print(statethread.call()+"\n");

for(local i = 0; i < 10000; i++)
        ::print(statethread.wakeup()+"\n");
```

Weak References

The weak references allows the programmers to create references to objects without influencing the lifetime of the object itself. In squirrel Weak references are first-class objects created through the built-in method obj.weakref(). All types except null implement the weakref() method; however in bools,integers and float the method simply returns the object itself(this because this types are always passed by value). When a weak references is assigned to a container (table slot,array,class or instance) is treated differently than other objects; When a container slot that hold a weak reference is fetched, it always returns the value pointed by the weak reference instead of the weak reference object. This allow the programmer to ignore the fact that the value handled is weak. When the object pointed by weak reference is destroyed, the weak reference is automatically set to null.

```
local t = {}
local a = ["first","second","third"]
//creates a weakref to the array and assigns it to a table slot
t.thearray = a.weakref();
```

The table slot 'thearray' contains a weak reference to an array. The following line prints "first", because tables(and all other containers) always return the object pointed by a weak ref

```
print(t.thearray[0]);
```

the only strong reference to the array is owned by the local variable 'a', so because the following line assigns a integer to 'a' the array is destroyed.

```
a = 123;
```

When an object pointed by a weak ref is destroyed the weak ref is automatically set to null, so the following line will print "null".

```
::print(typeof(t.thearray))
```

Handling weak references explicitly

If a weak reference is assigned to a local variable, then is treated as any other value.

```
local t = {}
local weakobj = t.weakref();
```

the following line prints "weakref".

```
::print(typeof(weakobj))
```

the object pointed by the weakref can be obtained through the built-in method weakref.ref().

The following line prints "table".

```
::print(typeof(weakobj.ref()))
```

Delegation

Squirrel supports implicit delegation. Every table or userdata can have a parent table (delegate). A parent table is a normal table that allows the definition of special behaviors for his child. When a table (or userdata) is indexed with a key that doesn't correspond to one of its slots, the interpreter automatically delegates the get (or set) operation to its parent.

```
Entity <- {
}

function Entity::DoStuff()
{
    ::print(_name);
}

local newentity=delegate Entity : {
    _name="I'm the new entity"
}

newentity.DoStuff(); //prints "I'm the new entity"
```

The parent of a table can be retreived through keyword `parent`. `parent` is a 'pseudo slot'. The `parent` slot cannot be set, the `delegete` statement has to be used instead.

```
local thedelegate = newentity.parent;
```

Metamethods

Metamethods are a mechanism that allows the customization of certain aspects of the language semantics. Those methods are normal functions placed in a table parent(delegate) or class declaration; Is possible to change many aspect of a table/class instance behavior by just defining a metamethod. Class objects(not instances) supports only 2 metamethods _newmember,_inherited.

For example when we use relational operators other than '==' on 2 tables, the VM will check if the table has a method in his parent called '_cmp' if so it will call it to determine the relation between the tables.

```
local comparable={
    _cmp = function (other)
    {
        if(name<other.name)return -1;
        if(name>other.name)return 1;
        return 0;
    }
}

local a=delegate comparable : { name="Alberto" };
local b=delegate comparable : { name="Wouter" };
```

```
if(a>b)
    print("a>b")
else
    print("b<=a");
```

for classes the previous code become:

```
class Comparable {
        constructor(n)
        {
                name = n;
        }
        function _cmp(other)
        {
                if(name<other.name) return -1;
                if(name>other.name) return 1;
                return 0;
        }
        name = null;
}

local a = Comparable("Alberto");
local b = Comparable("Wouter");

if(a>b)
    print("a>b")
else
    print("b<=a");
```

_set

invoked when the index idx is not present in the object or in its delegate chain

```
function _set(idx,val) //returns val
```

_get

invoked when the index idx is not present in the object or in its delegate chain

```
function _get(idx) //return the fetched values
```

_newslot

invoked when a script tries to add a new slot in a table.

```
function _newslot(key,value) //returns val
```

if the slot already exists in the target table the method will not be invoked also if the “new slot” operator is used.

_delslot

invoked when a script deletes a slot from a table.

if the method is invoked squirrel will not try to delete the slot himself

```
function _delslot(key)
```

_add

the + operator

```
function _add(op) //returns this+op
```

_sub

the – operator (like _add)

_mul

the * operator (like _add)

_div

the / operator (like _add)

_modulo

the % operator (like _add)

_unm

the unary minus operator

```
function _unm()
```

_typeof

invoked by the typeof operator on tables ,userdata and class instances

```
function _typeof() //returns the type of this as string
```

_cmp

invoked to emulate the < > <= >= operators

```
function _cmp(other)
```

```
returns an integer:
```

>0	if this > other

0	if this == other
<0	if this < other

_call

invoked when a table, userdata or class instance is called

```
function _call(original_this,params…)
```

_cloned

invoked when a table or class instance is cloned(in the cloned table)

```
function _cloned(original)
```

_nexti

invoked when a userdata or class instance is iterated by a foreach loop

```
function _nexti(previdx)
```

if previdx==null it means that it is the first iteration. The function has to return the index of the 'next' value.

_tostring

invoked when during string conacatenation or when the `print` function prints a table, instance or userdata. The method is also invoked by the sq_tostring() api

```
function _tostring()
```

must return a string representation of the object.

_inherited

invoked when a class object inherits from the class implementing `_inherited` the `this` contains the new class.

```
function _inherited(attributes)
```

return value is ignored.

_newmember

invoked for each member declared in a class body(at declaration time).

```
function _newmember(index,value,attributes)
```

if the function is implemented, members will not be added to the class.

Built-in functions

The squirrel virtual machine has a set of built utility functions.

Global symbols

`array(size,[fill])`
create and returns array of a specified size.if the optional parameter *`fill`* is specified its value will be used to fill the new array's slots. If the *`fill`* paramter is omitted `null` is used instead.

`seterrorhandler(func)`
sets the runtime error handler

`setdebughook(hook_func)`
sets the debug hook

`enabledebuginfo(enable)`
enable/disable the debug line information generation at compile time. enable != null enables . enable == null disables.

`getroottable()`
returns the root table of the VM.

`setroottable(table)`
sets the root table of the VM.

`getconsttable()`
returns the const table of the VM.

`setconsttable(table)`
sets the const table of the VM.

`assert(exp)`
throws an exception if exp is null

`print(x)`
prints x in the standard output

`compilestring(string,[buffername])`
compiles a string containing a squirrel script into a function and returns it

```
local compiledscript=compilestring("::print(\"ciao\")");
//run the script
compiledscript();
```

`collectgarbage()`
calls the garbage collector and returns the number of reference cycles found(and deleted)

`type(obj)`
return the 'raw' type of an object without invoking the metatmethod '_typeof'.

`getstackinfos(level)`
returns the stack informations of a given call stack level. returns a table formatted as follow:

```
{
        func="DoStuff", //function name
        src="test.nut", //source file
        line=10,         //line number
        locals = {         //a table containing the local variables
                a=10,
                testy="I'm a string"
        }
}
```

level = 0 is the current function, level = 1 is the caller and so on. If the stack level doesn't exist the function returns null.

`newthread(threadfunc)`
creates a new cooperative thread object(coroutine) and returns it

`_version_`
string values describing the version of VM and compiler.

`_charsize_`
size in bytes of the internal VM rapresentation for characters(1 for ASCII builds 2 for UNICODE builds).

`_intsize_`
size in bytes of the internal VM rapresentation for integers(4 for 32bits builds 8 for 64bits builds).

Default delegates

Except null and userdata every squirrel object has a default delegate containing a set of functions to manipulate and retrieve information from the object itself.

Integer

`tofloat()`
convert the number to float and returns it

`tostring()`
converts the number to string and returns it

`tointeger()`
returns the value of the integer(dummy function)

`tochar()`
returns a string containing a single character rapresented by the integer.

`weakref()`
dummy function, returns the integer itself.

Float

`tofloat()`
returns the value of the float(dummy function)

`tointeger()`
converts the number to integer and returns it

`tostring()`
converts the number to string and returns it

`tochar()`
returns a string containing a single character rapresented by the integer part of the float.

`weakref()`
dummy function, returns the float itself.

Bool

`tofloat()`
returns 1.0 for `true` 0.0 for `false`

`tointeger()`
returns 1 for `true` 0 for `false`

`tostring()`
returns "true" for `true` "false" for `false`

`weakref()`
dummy function, returns the bool itself.

String

`len()`
returns the string length

`tointeger()`
converts the string to integer and returns it

`tofloat()`
converts the string to float and returns it

`tostring()`
returns the string(dummy function)

`slice(start,[end])`
returns a section of the string as new string. Copies from start to the end (not included). If start is negative the index is calculated as length + start, if end is negative the index is calculated as length + start. If end is omitted end is equal to the string length.

`find(substr,[startidx])`
search a sub string(substr) starting from the index startidx and returns the index of its first occurrence. If startidx is omitted the search operation starts from the beginning of the string. The function returns null if substr is not found.

`tolower()`
returns a lowercase copy of the string.

`toupper()`
returns a uppercase copy of the string.

`weakref()`
returns a weak reference to the object.

Table

`len()`
returns the number of slots contained in a table

`rawget(key)`
tries to get a value from the slot ‘key’ without employ delegation

`rawset(key,val)`
sets the slot ‘key’ with the value ‘val’ without employing delegation. If the slot do not exists , it will be created.

`rawdelete()`
deletes the slot key without emplying delegetion and retunrs his value. if the slo does not exists returns always null.

`rawin(key)`
returns true if the slot ‘key’ exists. the function has the same eddect as the operator 'in' but does not employ delegation.

`weakref()`
returns a weak reference to the object.

`tostring()`
tries to invoke the _tostring metamethod, if failed. returns "(table : pointer)".

`clear()`
removes all the slot from the table

Array

`len()`
returns the length of the array

`append(val)`
appends the value ‘val’ at the end of the array

`push(val)`
appends the value ‘val’ at the end of the array

`extend(array)`
Extends the array by appending all the items in the given array.

`pop()`
removes a value from the back of the array and returns it.

`top()`
returns the value of the array with the higher index

`insert(idx,val)`
inserst the value ‘val’ at the position ‘idx’ in the array

`remove(idx)`
removes the value at the position ‘idx’ in the array

`resize(size,[fill])`
resizes the array, if the optional parameter *`fill`* is specified its value will be used to fill the new array's slots(if the size specified is bigger than the previous size) . If the *`fill`* paramter is omitted `null` is used instead.

`sort([compare_func])`

sorts the array. a custom compare function can be optionally passed.The function prototype as to be the following.

```
function custom_compare(a,b)
{
        if(a>b) return 1
        else if(a<b) return -1
        return 0;
}
```

`reverse()`
reverse the elements of the array in place

`slice(start,[end])`
returns a section of the array as new array. Copies from start to the end (not included). If start is negative the index is calculated as length + start, if end is negative the index is calculated as length + start. If end is omitted end is equal to the array length.

`weakref()`
returns a weak reference to the object.

`tostring()`
returns the string "(array : pointer)".

`clear()`
removes all the items from the array

Function

`call(_this,args…)`
calls the function with the specified environment object('this') and parameters

`pcall(_this,args…)`
calls the function with the specified environment object('this') and parameters, this function will not invoke the error callback in case of failure(pcall stays for 'protected call')

`acall(array_args)`
calls the function with the specified environment object('this') and parameters. The function accepts an array containing the parameters that will be passed to the called function.

`pacall(array_args)`
calls the function with the specified environment object('this') and parameters. The function accepts an array containing the parameters that will be passed to the called function.This function will not invoke the error callback in case of failure(pacall stays for 'protected array call')

`weakref()`
returns a weak reference to the object.

`tostring()`
returns the string "(closure : pointer)".

`tostring()`
returns the string "(closure : pointer)".

`bindenv(env)`
clones the function(aka closure) and bind the enviroment object to it(table,class or instance). the `this` parameter of the newly create function will always be set to env. Note that the created function holds a weak reference to its environment object so cannot be used to control its lifetime.

Class

`instance()`
returns a new instance of the class. this function does not invoke the instance constructor. The constructor must be explicitly called(eg. class_inst.constructor(class_inst)).

`getattributes(membername)`
returns the attributes of the specified member. if the parameter member is null the function returns the class level attributes.

`getattributes(membername,attr)`
sets the attribute of the specified member and returns the previous attribute value. if the parameter member is null the function sets the class level attributes.

`rawin(key)`
returns true if the slot ‘key’ exists. the function has the same eddect as the operator 'in' but does not employ delegation.

`weakref()`
returns a weak reference to the object.

`tostring()`
returns the string "(class : pointer)".

Class Instance

`getclass()`
returns the class that created the instance.

`rawin(key)`
returns true if the slot ‘key’ exists. the function has the same eddect as the operator 'in' but does not employ delegation.

`weakref()`
returns a weak reference to the object.

`tostring()`
tries to invoke the _tostring metamethod, if failed. returns "(insatnce : pointer)".

Generator

`getstatus()`
returns the status of the generator as string : “running”, ”dead” or ”suspended”.

`weakref()`
returns a weak reference to the object.

`tostring()`
returns the string "(generator : pointer)".

Thread

`call(...)`
starts the thread with the specified parameters

`wakeup([wakeupval])`
wakes up a suspended thread, accepts a optional parameter that will be used as return value for the function that suspended the thread(usually suspend())

`getstatus()`
returns the status of the thread ("idle","running","suspended")

`weakref()`
returns a weak reference to the object.

`tostring()`
returns the string "(thread : pointer)".

Weak Reference

`ref()`
returns the object that the weak reference is pointing at, null if the object that was point at was destroyed.

`weakref()`
returns a weak reference to the object.

`tostring()`
returns the string "(weakref : pointer)".