

Game Monkey Script Reference

COMMENTS

```
// Comment to the end of line like c++

/*
    Block comment like c and c++
*/
```

Comments are completely ignored by the compiler, they are purely for documenting the code, or temporarily disabling blocks of code.

VARIABLES and CONSTANTS

GM is not a strong typed language like C, Pascal etc. It is more like basic.
A variable may be one of the following types:

null - no value
int - a 32 bit signed integer
float - a 32 bit floating point number
string - a null terminated ansi character string
table - an array \ hash container
function - a function
user - user type

Variables are case sensitive (in the current build). `__t0` and `__t1` are reserved for internal use.
[a..zA..Z_][a..zA..Z_0..9]

Example

```
a = null;           // a has no value
b = 4;              // b is an int
c = 4.4;            // c is a float
d = "hello";        // d is a string
e = table();        // e is an empty table
f = function() {};  // f is a function
```

More examples:

```
a = 'SMID';          // a is an int ('S'<<24 | 'M'<<16 | 'I'<<8 | 'D')
b = .23;             // b is a float
c = 2.4f;            // c is a float
d = `c:\windows\sys`; // d is a string
```

The language and its standard functions try to preserve values, but otherwise preserve type. The rule is that whenever different types are used together, the higher type is preserved or used. The types from low to high are *int*, *float*, *string*.

Examples:

```
print("hello " + 4); // output is "hello 4", 4 was upgraded
print(2 + 5);        // output is "7", int type was preserved
print(2.0 + 5);      // output is "7.0", 5 was upgraded
print( sqrt(17.0) ); // output is "4.1231", float type was preserved
print( sqrt(17) );   // output is "4", int type was preserved
```

Integer assignment examples:

```
a = 179;             // decimal
a = 0xB3;            // hexadecimal
a = 0b10110011;      // binary
```

```
a = 'BLCK'; // characters as four bytes making integer
```

Float assignment examples:

```
b = 45.23235; // float with decimal places
b = .2367; // number begins with .
b = 289.0; // only while number, but sill has .
b = 12.45f; // C style float
b = 2.3E-3; // scientific notation
```

String assignment examples:

```
c = "c:\\path\\file.ext"; // standard double quotes, uses '\\' as escape code
c = `c:\path\file.ext`; // back dash does no processing other than
                        // collapse `` to insert a ` in the string.
                        // (useful for file paths).
c = "Mary says \"hello\""; // creates the string 'Mary says "hello"'
c = `Chris`s bike`; // creates the string 'Chris`s bike'
```

Basic types can be explicitly converted when using the standard binding library.

Int(), Float(), String() are bound to the basic types.

For example:

```
a = 10;
b = a.String(); // GOOD: Explicitly call a type bound function
                // that returns a converted value.
b = "" + a; // BAD: Assignment upgrades 'a' to higher type of 'string',
            // but is inefficient.
b = (10).String(); // UGLY: This works, but brackets are required to prevent
                       the decimal place from being misinterpreted
                       as a float.
b = 10.String(); // ERROR: This wont compile because the compiler expects
                  // more numbers after the dot.
```

Reference type variables are types String, Function, Table, User. When a variable is assigned to one of these, it does not make a full copy of the variable, instead it merely references it.

Example:

```
a = table("apple", "orange"); // Let a be a table
b = a; // b merely references the value of a
b[1] = "banana"; // Set element 1 of table
print(a[0], a[1]);
print(b[0], b[1]);
```

Output:

```
apple banana
apple banana
```

When a variable is assigned a value, then later assigned another value, the original value may be lost if it is not assigned to another variable.

Example:

```
Operation = function(a, b) // Operation is assigned a function
{
  return a + b
};

Operation = "hello"; // Operation is assigned a string, the previous
                    // function will be lost (and garbage collected).
```

FUNCTIONS

Syntax: **function** (<params>) { <statements> };

Functions are just another type of variable.

Note: Remember the semicolon after assigning a function to a variable.

Example:

```
// CreateRect is assigned to a function that will create a rect table.
```

```
CreateRect = function(posX, posY, sizeX, sizeY)
{
    rect = table(x = posX, y = posY, width = sizeX, height = sizeY);
    rect.Area = function() { return .width * .height; };

    return rect;
};
```

```
myRect = CreateRect(0, 0, 5, 10); // Construct a table that describes
                                a rectangle
```

```
area = myRect.Area(); // myRect is automatically assigned to 'this'
                      within the area method.
```

Example use of '.' to explicitly pass 'this'.

```
Size = function()
{
    return .width * .height;
};
```

```
s = myRect:Size(); // Calls Size function passing 'myRect' as 'this'
```

SCOPING *Keywords **global**, **local**, **member** and **this***

Syntax: **global** <variable>
 Local <variable>
 member <variable>
 this
 this.<variable>
 .<variable>

Variables in functions.

A variable used inside a function is local to the function by default. A variable may be set to the global scope using the 'global' keyword. Members must be accessed from 'this' or declared using the 'member' keyword. Variables may be accessed from the local scope using the 'local' keyword, useful if they have previously been set to another scope.

Example:

```
Access = function() // Access is local to the function it is in
{
    apple = 3; // 'apple' variable is local to function
    global apple; // Declare 'apple' to be accessed from global scope
    local apple; // Declare 'apple' to be accessed locally again
    member apple; // Declare 'apple' to be a member of 'this'
    this.apple; // Explicitly access 'apple' off 'this'
    .apple; // Implicitly access 'apple' off 'this'
};
```

Example:

```
a = 13; // a is a local variable (eg. local to file or string)
```

```

print(b);           // b is null, unless there is a b global variable

global b = function() // b is a global variable of type GM_FUNCTION
{
    global c = 2;      // c is a global variable
    d = 3;             // d is local to entire scope of function

    {
        if(c == 2)
        {
            local e = 3; // e is local to entire scope of function,
                          // from this point on.
        }
    }

    print(e);         // e is 3
};

```

The Variable lookup order from within a function is *locals & parameters* then *globals*.

Members are slightly different

```

h = function()      // h is a local variable
{
    global a = 3;    // a is global
    member n;        // n is now accessed or created from 'this'

    d = 3;           // d is local to the function
    this.b = 3;      // b is on calling this
    .b = .x + 1;     // b and x are on calling this

    print(b);        // b is null (as there was no local b)
    print(n);        // same as this.n from within function
};

```

Statements in file global scope.

```

x = 7;              // this is a local, not be a global variable, and
                    // cannot be accessed from functions within the file.
global x = 8;       // this is global and can be accessed from
                    // within functions and from other threads.

a = function(y)
{
    local x = 5;     // this is function scoped x
    dostring("print(x);"); // this will print 8. Finds global x.
                        dostring cannot access vars or params
                        from parent function.
};

```

Variables can be scoped globally to the whole virtual machine, locally within a function, or can be members of something such as a table. Note that when a file or string is executed, it is actually an unnamed function, thus variables used outside of functions are local to the unnamed file/string function by default.

The *this* is always present. It is either *null* or a valid table. You may pass a *this*, or override the default *this* using the *:* (colon) operator. This feature has many uses such as creating *template* style behavior, where the object operated on is unknown until run-time. It is also used to *this* to a new thread. For example:

```

obj:thread(obj.DoThings)    // Start a thread and pass 'obj' as this.

obj:MakeFruit(apple, orange) // Call MakeFruit() and pass 'obj' as this.

```

SYMBOLS and OPERATORS

!	(exclamation)	Not
~	(tilda)	Binary Compliment (flips all bits).
^	(caret)	Bit wise XOR (exclusive or)
	(verical pipe)	Bit wise OR
&	(ampersand)	Bit wise AND
>>	(double greater)	Bit shift right
<<	(double less)	Bit shift left
=	(equal)	Assignment
'	(single quote)	Character(s) as integer
"	(double quote)	Character(s) as string with escape codes
`	(back dash)	Character(s) as string with no processing
[]	(squre brackets)	Index table member
.	(dot)	Member of table
:	(colon)	Pass 'this' to function
+	(plus)	Math Addition
-	(minus)	Math Subtraction
*	(multiply)	Math Multiplication
/	(divide)	Math Division
%	(percent)	Math Modulo, or remainder from division.
{ }	(curly braces)	Delimit statement block
;	(semicolon)	End statement
<	(less)	Comparison less than
<=	(less equal)	Comparison less than or equal to
>	(greater)	Comparison greater than
>=	(greater equal)	Comparison greater than or equal to
==	(double equal)	Comparison equal to
&&	(double ampersand)	Conditional AND
and		Conditional AND
	(double vertical pipe)	Conditional OR
or		Conditional OR

TABLES

Syntax: **table** (<key> = <value> , ...);
table (<value> , ...);

Tables can be considered to be both Arrays and Maps. Since tables can contain tables, they can also be Trees.

Example initialisation:

```
fruit = table("apple", "bannana", favorite = "tomato", "cherry");
```

The table 'fruit' now logically contains:

```
fruit[0] = "apple";  
fruit[1] = "bannana";  
fruit[2] = "cherry";  
fruit["favorite"] = "tomato"; (otherwise written as) fruit.favorite = "tomato";
```

Note that fruit.favorite = "tomato" did not go in element [2], but is logically separate from the other elements because it is an associated, not an indexed member.

Examples of getting members from a table.

```

a = thing.other;    // 'other' is a member of the table 'thing'
b = thing["other"]; // has the same effect as b = thing.other
c = thing[2];       // c gets assigned the 3rd indexed member of table 'thing'

index = 3;
d = thing[index];   // Access the table as Array using integer variable.

assoc = "fav";
e = thing[assoc];   // Access the table as Map using string variable.

```

Note that `thing["Fav"]` and `thing["fav"]` are different even if the language is set to Case Insensitive. This is necessary because a) Associations should can be strings or values of any type. and b) Checking for lower case version would add significant processing cost.

Examples of setting members in a table.

```

thing.other = 4;    // 'other' is (or now is) a member of 'thing' and
                    // is assigned the value 4.
thing[3] = "hello"; // The 4th indexed member is assigned the value of "hello".

```

Keywords *if* and *else*

Syntax: **if** (<condition>) { <statements> }
 Or **if** (<condition>) { <statements> } **else** { <statements> }
 Or **if** (<condition>) { <statements> } **else if** (<condition>) { <statements> } **else** { <statements> }

Example:

```

foo = 3;
bar = 5;
if ( (foo * 2) > bar )
{
    print( foo * 2, "is greater than", bar );
}
else
{
    print( foo * 2, "is less than", bar );
}

```

Output:

6 is greater than 5

'If' evaluates a condition then executes some statements if that condition was true. If the condition was not true, the statements following the 'else' are executed.

Inside an 'if' statement, the condition components are evaluated according to standard precedence rules, and otherwise left to right.

The condition "`if (3 * 4 + 2 > 13)`" is the same as "`if ((3*4) + 2 > 13)`".

In the condition "`if (3 > 0 || 2 < 1)`" the `(3>0)` is evaluated first so the second part will never be evaluated due to the 'or' (`||`).

Note to C programmers; You cannot use single line (no statement block) style control flow statements.

Example:

```

if ( a > 4 ) b = 3; // Error. Must have { } around statements belonging to 'if'.

```

Keyword **for**

Syntax: **for** (<statement> ; <condition> ; <statement>) { <statements> }

Example:

```
for(index = 0; index < 6; index = index + 2)
{
    print("index =",index);
}
```

Output:

```
index = 0
index = 2
index = 4
```

The order in which 'for' statements are executed is:

1. The first statement (before the first semicolon).
2. The statement block (inside braces).
3. The condition (between the semicolons).
4. The last statement (after the second semicolon).

Although the example shows a very common usage, the 'for' statement could have been written:

```
index = 0;
for (; index < 6;)
{
    print("index =",index);
    index = index + 2;
}
```

Keyword **foreach**

Syntax: **foreach** (<key> and <value> in <table>) { <statements> }
foreach (<value> in <table>) { <statements> }

Example:

```
fruitnveg = table("apple", "orange", favorite = "pear",
                  yucky = "turnip", "pinapple");

foreach( keyVar and valVar in fruitnveg)
{
    print( keyVar, "=", valVar);
}
```

Output:

```
2 = pinapple
0 = apple
favorite = pear
1 = orange
yucky = turnip
```

Note that the table was not iterated in any particular order. In fact the order could change from run to run as the internal table data is reordered as it expands and contracts.

'foreach' iterates over a tables contents returning the key and value as local variables for use within the statement block. Although the foreach iteration is 'delete safe', the behaviour of adding and removing items from a table while iterating is undefined.

Keyword **while**

Syntax: **while** (<condition>) { <statements> }

Example:

```
index = 0;
while ( index < 3 )
{
    print( "index =", index);
    index = index + 1;
}
```

Output:

```
index = 0
index = 1
index = 2
```

The 'while' statement evaluates a condition and upon its success, executes the statement block, then repeats this process. Because the condition is evaluated first, the statement block may never execute if the condition evaluates to false the first time.

Keyword **dowhile**

Syntax: **dowhile** (<condition>) { <statements> }

Example:

```
index = 0;
dowhile ( index > 0 )
{
    print("index =", index);
    index = index - 2;
}
```

Output:

```
index = 0
```

The 'dowhile' executes the statements, then tests the condition. This is the opposite order to 'while'. Had this example used a 'while' instead of 'dowhile', there would have been no output.

Keywords **break**, **continue** and **return**

Example with break:

```
for (index = 0; index<4; index=index+1)
{
    if(index == 2)
    {
        break;
    }

    print("index =", index);
}
```

Output:

```
index = 0
index = 1
```

Example with continue:

```
for (index = 0; index<4; index=index+1)
{
    if(index == 2)
    {
        continue;
    }

    print("index =", index);
}
```

Output:

```
index = 0
index = 1
index = 3
```

Example with returned value:

```
Add = function(a, b)
{
    return a + b;
};

print ( "Add result =", Add(3,4) );
```

Output:

```
Add result = 7
```

Example with return

```
Early = function (a)
{
    if ( a <= 0)
    {
        print ( "Don't want zero here.");
        return;
    }
    print ("Above zero we handle.");
};

Early( -2 );
```

Output:

```
Don't want zero here.
```

'break' and 'continue' are used to exit or skip execution within 'for', 'while', 'dowhile', 'foreach' style loops. 'break' causes the execution to exit to after the statement block.

'continue' causes the execution to skip the rest of the statement block and test the loop condition again immediately.

'return' will also break out of a loop, but will exit the entire function, not just the statement block of a loop.

Keywords *true*, *false* and *null*

In this language, 'true' and 'false' merely represent 0 (zero) and 1 (one). They have no other meaning. These keywords are provided to allow easy reading of code that is logically uses Boolean integers. They will usually be return values from functions wanting to signal success or failure and will rarely be used in actual comparison.

Example:

```
a = 3;
if ( a == true)
{
    print (a, " == ",true);
}
else
{
    print (a, " != ", true);
}
```

Output:

```
3 != 1
```

In this language 'null' is a type. It is generally used an error type. When used in an expression with higher types it can be interpreted as 0 (zero). When a variable is declared but not assigned any value, its value is 'null'. If a value in a table is set to null, it is removed from the table.

Example:

```
local var;                                // Un-initialised variable is null

if(var)                                    // null interpreted as zero here
{
    print ("var was initialised or non zero :", var);
}
else
{
    print ("var is zero or null : ", var);
}
```

Output:

```
var is zero or null : null
```

THREADS

int thread(function a_function, ...)

Create a new thread.

Param a_function The function to execute.

Param ... Parameters passed to a_function.

Return A thread identifier that may be used to control or query the thread.

void yield()

Cause current execution to yield control to the virtual machine.

void exit()

Cause the current thread to terminate immediately.

void threadKill(int a_threadId)

Kill a thread. The thread identified will not execute again.
Param a_threadId The identifier of the thread returned by thread().
If this parameter is not present, the current thread will be killed.

void sleep(float a_time)

Stop execution of the thread for period of time.
Param a_time The time in seconds to sleep.

int threadTime()

Return the time the thread has been running in milliseconds.

int threadId()

Return the identifier of the current thread.

void signal(var a_event)

Signal an event.
Param a_event A variable of any type used as signal.

void block(var a_event, ...)

Block thread execution on one or more events. Control is yielded unless a signal is currently in the queue to allow continuation.
Param a_event Event to wait on.
Param ... Multiple events to wait on.

void assert(int a_condition)

Check that a_condition is true (not zero) otherwise cause exception and exit thread.

STATES

Game programs often use the concept of states to describe entity behavior, and implement what is known as a Finite State Machine. In GameMonkey, *states* allow the thread stack to be immediately destroyed and execution began at a new point. Optionally an *exit* function may be called before the state changes. The previous state may also be queried.

void stateSet(function a_function, ...)

Set execution of this thread to a new state function immediately.
Param a_function The function to execute.
Param ... The parameters passed to a_function.

function stateGet()

Return the current state function being executed. If setstate() was never called, the return value is null.

function stateGetLast()

Return the last state that was set before the current one. Useful to know what the transition was.

void stateSetExitFunction(function a_function)

Set a function to call when state is about to change. Allows a state function to clean itself up before execution transitions.

Binding C functions to be called from script.

C bound functions can be bound to a type, or bound as a global variable.

Example with strict types:

```
// int GetHealth( GObj* a_obj, int a_limit)

int __cdecl GetHealth(gmThread * a_thread)
{
    GM_CHECK_NUM_PARAMS(2); // Check for 2 params
    GM_CHECK_USER_PARAM(GObj::s_scrUserType, userObj, 0) // Check and assign param 1 as user
    GM_CHECK_INT_PARAM(limit, 1); // Check and assign param 2 as int

    GObj* gob = (GObj*)userObj->m_user; // Cast gmUserObject to our type

    If(gob->m_health > a_limit) // Do something with gob & limit
    {
        gob->m_health = a_limit;
    }

    a_thread->PushInt(gob->m_health); // Return a int value to script

    return GM_OK; // Return function succeeded
}
```

Example that uses varying types:

```
//
// Declare function as cdecl with single gmThread parameter and int return
//
// int|float sqrt( int|float a_value );
//
int __cdecl gmfSqrt(gmThread * a_thread)
{
    GM_CHECK_NUM_PARAMS(1); // Check for number of parameters
                                // or return exception
    if(a_thread->ParamType(0) == GM_INT) // Check if first parameter is a integer
    {
        int intValue = a_thread->Param(0).m_value.m_int;
        a_thread->PushInt((int)sqrt(intValue)); // Return an integer
        return GM_OK; // Return GM_OK on success
    }
    else if(a_thread->ParamType(0) == GM_FLOAT) // Check if first parameter is a float
    {
        float floatValue = a_thread->Param(0).m_value.m_float;
        a_thread->PushFloat(sqrtf(floatValue)); // Return a float
        return GM_OK; // Return GM_OK on success
    }
    return GM_EXCEPTION; // Return GM_EXCEPTION to signal failure
}
```

Register this function like:

```
static gmFunctionEntry s_mathLib[] =
{
    {"sqrt", gmfSqrt},
};
machine->RegisterLibrary(s_mathLib, sizeof(s_mathLib) / sizeof(s_mathLib[0]));
```

Example of type function:

```
//
// int String.Compare( string a_other );
//
int __cdecl gmfStringCompare(gmThread * a_thread)
{
    GM_CHECK_NUM_PARAMS(1); // Check number of parameters or
                           // return exception

    if(a_thread->ParamType(0) == GM_STRING) // Check parameter type is another string
    {
        gmVariable * var = a_thread->GetThis(); // Get variable 'this' since we work only
        // off a type eg. "hello".Compare()

        GM_ASSERT(var->m_type == GM_STRING); // Paranoid check that 'this' type is a string

        gmStringObject * strObj = (gmStringObject *) GM_OBJECT(var->m_value.m_ref); // Get the
        // string object from variable
        const char* thisStr = (const char *) *strObj; // Get the const char* from string object
        const char* otherStr = a_thread->ParamString(0); // Get the const char* from parameter
        // string object

        a_thread->PushInt(strcmp(thisStr, otherStr)); // Return int result from strcmp

        return GM_OK; // Signal success
    }

    return GM_EXCEPTION; // Signal failure
}
```

Register this function like:

```
static gmFunctionEntry s_stringLib[] =
{
    {"Compare", gmfStringCompare},
};
machine->RegisterTypeLibrary(GM_STRING, s_stringLib, sizeof(s_stringLib)
    / sizeof(s_stringLib[0]));
```

Calling Script functions from C

Calling script functions from C is easy when using the gmCallScript helper class. To call script functions manually, look at the source code in gmCallScript.h.

Example:

```
#include "gmcallscrip.h" // This header contains the helper class

gmCallScript::SetMachine(&machine); // Call this once after creating the virtual machine.

// Calls a script function: " global Add = function(a, b) {return a + b;} "

int resultInt = 0; // The variable to store our future return value.
if(gmCallScript::BeginGlobal("Add")) // A global function called 'Add'.
    // False is returned here if function was not found.
{
    gmCallScript::SetReturnInt(resultInt); // We want to know about the return value which is an int
    gmCallScript::AddParamInt(3); // Parameter 1 is an int of this value
    gmCallScript::AddParamInt(5); // Parameter 2 is an int of this value
    gmCallScript::End(); // Finish the call and set the return value
}
// resultInt now contains 8
```

WARNING:

If you want a returned string from a function, use it or copy it immediately. Do not hold onto the pointer. The string may be garbage collected in the very next execution cycle and will no longer be valid.

GAME OBJECT EXTENSIONS

How do I extend the language with a user type that behaves like a table.
Something that could represent a game object?

```
struct GameObject
{
    gmTableObject * m_table;           // Contain the table functionality
    gmUserObject * m_userObject;

    static gmType s_typeId;           // Store our user type
};

gmType GameObject::s_typeId = GM_NULL; // Instanciate static used to store user type

#if GM_USE_INCGC

static bool GM_CDECL GCTrace(gmMachine * a_machine, gmUserObject* a_object, gmGarbageCollector*
a_gc, const int a_workLeftToDo, int& a_workDone)
{
    GM_ASSERT(a_object->m_userType == GameObject::s_typeId);
    GameObject * object = (GameObject *) a_object->m_user;
    if(object->m_table) a_gc->GetNextObject(object->m_table);
    a_workDone += 2; // contents + this

    return true;
}

static void GM_CDECL GCDestruct(gmMachine * a_machine, gmUserObject* a_object)
{
    GM_ASSERT(a_object->m_userType == GameObject::s_typeId);
    GameObject * object = (GameObject *) a_object->m_user;
    object->m_table = NULL;
}

#else //GM_USE_INCGC

// Garbage collect 'Mark' function
void GM_CDECL GameObjectMark(gmMachine * a_machine, gmUserObject * a_object, gmuint32 a_mark)
{
    GM_ASSERT(a_object->m_userType == GameObject::s_typeId);
    GameObject * object = (GameObject *) a_object->m_user;
    object->m_table->Mark(a_machine, a_mark);
}

// Garbage collect 'Garbage Collect' function
void GM_CDECL GameObjectGC(gmMachine * a_machine, gmUserObject * a_object, gmuint32 a_mark)
{
    GM_ASSERT(a_object->m_userType == GameObject::s_typeId);
    GameObject * object = (GameObject *) a_object->m_user;
    object->m_table.Destruct(a_machine);
    delete object;
}

#endif //GM_USE_INCGC

// Set string to describe object when 'AsString' is used.
static void GM_CDECL AsString(gmUserObject * a_object, char* a_buffer, int a_bufferLen)
{
    GM_ASSERT(a_object->m_userType == GameObject::s_typeId);
    GameObject * object = (GameObject *) a_object->m_user;

    char mixBuffer[128];
    sprintf(mixBuffer, "GameObject Cptr = %x", object);
    int mixLength = strlen(mixBuffer);
    int useLength = GM_MIN(mixLength, a_bufferLen-1);
    GM_ASSERT(useLength > 0);
    strncpy(a_buffer, mixBuffer, useLength);
    a_buffer[useLength] = 0;
}

// Get Dot operator for table access
void GM_CDECL GameObjectGetDot(gmThread * a_thread, gmVariable * a_operands)
{
    //O_GETDOT = 0, // object, "member" (tos is a_operands + 2)
    GM_ASSERT(a_operands[0].m_type == GameObject::s_typeId);
    gmUserObject * user = (gmUserObject *) GM_OBJECT(a_operands[0].m_value.m_ref);
    GameObject * object = (GameObject *) user->m_user;
    a_operands[0] = object->m_table->Get(a_operands[1]);
}
```

```

// Set Dot operator for table access
void __cdecl GameObjectSetDot(gmThread * a_thread, gmVariable * a_operands)
{
    //O_SETDOT,          // object, value, "member"    (tos is a_operands + 3)
    GM_ASSERT(a_operands[0].m_type == GameObject::s_typeId);
    gmUserObject * user = (gmUserObject *) GM_OBJECT(a_operands[0].m_value.m_ref);
    GameObject * object = (GameObject *) user->m_user;
    object->m_table->Set(a_thread->GetMachine(), a_operands[2], a_operands[1]);
}

// Create one of these user types from script
// NOTE: It is unusual to create a game object entirely within the script like this.
// Also note that a game may not want to use C pointers for game objects in the script variable.
// Instead a 32bit UID may be used and looked up to find and verify the game object.
int GM_CDECL CreateGameObject(gmThread * a_thread)
{
    GameObject * object = new GameObject();

    object->m_table = a_thread->GetMachine()->AllocTableObject();
    object->m_userObject = a_thread->CreateUser(object, GameObject::s_typeId);

    // Note that 'object' would probably be stored in C as well somewhere so it can be found.
    // Note 'object->m_userObject' must be stored somewhere in Script to persist.
    //      or the game must register a garbage collect callback to Mark/Trace the objects.

    return GM_OK;
}

// Get one of these user types from script
// Used to maintain a 1 to 1 relationship between game objects in C and Script. This prevents
// garbage accumulation in the VM.
int GM_CDECL GetGameObject(gmThread * a_thread)
{
    // Search through C container to find GameObject
    GameObject* foundObj = NULL;

    GM_CHECK_NUM_PARAMS(1);

    if(a_thread->ParamType(0) == GM_STRING) // Search by Name
    {
        // TODO: foundObj = FindByName(a_thread->ParamString(0));

        // If found
        a_thread->PushUser(foundObj->m_userObject);
        // If not found, return GM_NULL
        a_thread->PushNull();

        return GM_OK;
    }
    else if(a_thread->ParamType(0) == GM_INT) // Search by Id
    {
        // TODO: foundObj = FindById(a_thread->ParamInt(0))

        // If found
        a_thread->PushUser(foundObj->m_userObject);
        // If not found, return GM_NULL
        a_thread->PushNull();

        return GM_OK;
    }

    return GM_EXCEPTION;
}

// Table of functions to register
gmFunctionEntry regFuncList[] =
{
    {"GameObject", CreateGameObject},
};

// Registration code to use the user type
// Assumes 'machine' is a gmMachine created earlier

// Register new user type
GameObject::s_typeId = machine.CreateUserType("GameObject");

// Register garbage collection etc. for our new type
#if GM_USE_INCGC
    a_machine->RegisterUserCallbacks(GameObject::s_typeId,
        GameObjectTrace, GameObjectDestruct,
        AsString);
#else //GM_USE_INCGC

```

```
    a_machine->RegisterUserCallbacks(GameObject::s_typeId,
                                     GameObjectMark, GameObjectGC,
                                     AsString);
#endif //GM_USE_INCGC

// Bind Get dot operator for our type
machine.RegisterTypeOperator(GameObject::s_typeId, O_GETDOT, NULL, GameObjectGetDot);
// Bind Set dot operator for our type
machine.RegisterTypeOperator(GameObject::s_typeId, O_SETDOT, NULL, GameObjectSetDot);
// Register the function to create one of these types from script
machine.RegisterLibrary(regFuncList, sizeof(regFuncList) / sizeof(regFuncList[0]));
```

CALLSTACK

Stack growing downward...

this
calling function reference
param 0 - stack base
...
param n-1

call n where n is the number of params.