tolua++ - Reference Manual

by Waldemar Celes, Ariel Manzur.

---

tolua++ is an extended version of tolua, a tool to integrate C/C++ code with Lua. **tolua++** includes new features oriented to c++ such as:

- Support for **std::string** as a basic type (this can be turned off by a command line option).
- Support for class templates

As well as other features and bugfixes.

tolua is a tool that greatly simplifies the integration of C/C++ code with Lua. Based on a *cleaned header file* (or extracts from real header files), **tolua** automatically generates the binding code to access C/C++ features from Lua. Using Lua API and tag method facilities, **tolua** maps C/C++ constants, external variables, functions, classes, and methods to Lua.

This manual is for **tolua++** version 1.0 and is implemented upon Lua 5.0 and based on **tolua 5.0**. See Compatibility for details on switching from older versions.

The sections below describe how to use **tolua**. Please contact us with bug reports, suggestions, and comments.

How tolua works

To use **tolua**, we create a *package file*, a C/C++ cleaned header file, listing the constants, variables, functions, classes, and methods we want to export to the Lua environment. Then **tolua** parses this file and creates a C/C++ file that automatically binds the C/C++ code to Lua. If we link the created file with our application, the specified C/C++ code can be accessed from Lua.
A package file can also include regular header files, other package files, or lua files.

Let's start with some examples. If we specify as input the following C-like header file to **tolua**:

```
#define FALSE 0
#define TRUE 1

enum {
 POINT = 100,
 LINE,
 POLYGON
}

Object* createObejct (int type);
void drawObject (Object* obj, double red, double green, double blue);
int isSelected (Object* obj);
```

A C file that binds such a code to Lua is automatically generated. Therefore, in Lua code, we can access the C code, writing, for instance:

```
...
myLine = createObject(LINE)
...
if isSelected(myLine) == TRUE then
  drawObject(myLine, 1.0, 0.0, 0.0);
else
  drawObject(myLine, 1.0, 1.0, 1.0);
end
...
```

Also, consider a C++-like header file:

```
#define FALSE 0
#define TRUE 1

class Shape
{
  void draw (void);
  void draw (double red, double green, double blue);
  int isSelected (void);
};

class Line : public Shape
{
 Line (double x1, double y1, double x2, double y2);
 ~Line (void);
};
```

If this file is used as input to **tolua**, a C++ file is automatically generated proving access to such a code from Lua. Therefore, it would be valid to write Lua statements like:

```
...
myLine = Line:new (0,0,1,1)
...
if myLine:isSelected() == TRUE then
 myLine:draw(1.0,0.0,0.0)
else
 myLine:draw()
end
...
myLine:delete()
...
```

The package file (usually with extension `.pkg`) passed to **tolua** is not the real C/C++ header file, but a *cleaned* version of it. **tolua** does not implement a complete parse to interpret C/C++ code, but it understands a few declarations that are used to describe the features that are to be exported to Lua. Regular header files can be included into packages files; **tolua** will extract the code specified by the user to parse from the header (see Basic Concepts).

How to use toLua

tolua is composed by two pieces of code: an executable and a library. The executable represents the parser that reads a package file and output a C/C++ code that implements the binding to access the C/C++ features from Lua. If the package file is a C++ like code (i.e., includes class definitions), a C++ code is generated. If the cleaned header file is a C like code (i.e., without classes), a C code is generated. **tolua** accepts a set of options. Running `"tolua -h"` displays the current accepted options. For instance, to parse a file called `myfile.pkg` generating the binding code in `myfile.c`, we do:

```
tolua -o myfile.c myfile.pkg
```

The generated code must be compiled and linked with the application to provide the desired access from Lua. Each parsed file represents a package being exported to Lua. By default, the package name is the input file root name (`myfile` in the example). The user can specify a different name for the package:

```
tolua -n pkgname -o myfile.c myfile.pkg
```

The package should also be explicitly initialized. To initialize the package from our C/C++ code, we must declare and call the initialization function. The initialization function is defined as

```
int tolua_pkgname_open (lua_State*);
```

where *pkgname* represents the name of the package being bound. If we are using C++, we can opt for automatic initialization:

```
tolua -a -n pkgname -o myfile.c myfile.pkg
```

In that case, the initialization function is automatically called. However, if we are planning to use multiple Lua states, automatic initialization does not work, because the order static variables are initialized in C++ is not defined.

Optionally, the prototype of the `open` function can be outputted to a header file, which name is given by the `-H` option.

The binding code generated by **tolua** uses a set of functions defined in the **tolua** library. Thus, this library also has to be linked with the application. The file `tolua.h` is also necessary to compile the generated code.

An application can use tolua object oriented framework (see exported utility functions) without binding any package. In that case, the application must call **tolua** initialization function (this function is called by any package file initialization function):

```
int tolua_open (void);
```

Basic Concepts

The first step in using **tolua** is to create the package file. Starting with the real header files, we clean them by declaring the features we want to access from Lua in a format that **tolua** can understand. The format **tolua** understands is simple C/C++ declarations as described below.

Including files

A package file may include other package file. The general format to do that is:

```
$pfile "include_file"
```

A package file may also include regular C/C++ header files, using the `hfile` or `cfile` directive:

```
$cfile "example.h"
```

In which case, **tolua** will extract the code enclosed between `tolua_begin` and `tolua_end`, or or `tolua_export` for a single line. Consider this C++ header as example:

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

class Example { // tolua_export

private:

 string name;
 int number;

public:

 void set_number(int number);

 //tolua_begin

 string get_name();
 int get_number();
};
// tolua_end

#endif
```

In this case, the code that's not supported by **tolua** (the private part of the class), along with the function `set_number` is left outside of the package that includes this header.

Finally, lua files can be included on a package file, using `$lfile`:

```
$lfile "example.lua"
```

New on tolua++: an extra way to include source files is available since version 1.0.4 of **tolua++**, using `ifile`:

```
$ifile "filename"
```

`ifile` also takes extra optional parameters after the filename, for example:

```
$ifile "widget.h", GUI
$ifile "vector.h", math, 3d
```

`ifile`'s default behaviour is to include the whole file, untouched. However, the contents of the file and the extra parameters are put through the `include_file_hook` function before being included into the package (see Customizing tolua++ for more details).

Basic types

tolua automatically maps C/C++ basic types to Lua basic types. Thus, `char`, `int`, `float`, and `double` are mapped to the Lua type `number`; `char*` is mapped to `string`; and `void*` is mapped to `userdata`. Types may be preceded by modifiers (`unsigned`, `static`, `short`, `const`, etc.); however, be aware that **tolua** ignores the modifier `const` if applied to basic types. Thus, if we pass a constant basic type to Lua and then pass it back to C/C++ code where a non constant is expected, the constant to non constant conversion will be silently done.

Functions in C/C++ can also manipulate Lua objects explicitly. Thus `lua_Object` is also considered a basic type. In this case, any Lua value matches it.

New on tolua++: The C++ type `string` is also considered a basic type, and is passed as a value to lua (using the `c_str()` method). This feature can be turned off with the command line option `-S`.

User defined types

All other types that appear in the package file being processed are considered user defined types. These are mapped to tagged userdata type in Lua. Lua can only store pointers to user defined types; although, **tolua** automatically makes the necessary arrangement to deal with references and values. For instance, if a function or method returns a value of user defined type, **tolua** allocates a clone object when returning it to Lua and sets the garbage collection tag method to automatically free the allocated object when no longer in use by Lua.

For user defined types, `const`ness is preserved. Thus passing a non constant user defined type to a function that expects constant type generates an type mismatching error.

`NULL` and `nil`

C/C++ `NULL` or `0` pointers are mapped to Lua `nil` type; conversely, `nil` may be specified wherever a C/C++ pointer is expected. This is valid for any type: `char*`, `void*`, and pointers to user defined types.

Typedefs

tolua also accepts simple typedef*'s* inside the package files. Any occurrence of a type after its definition is mapped by **tolua** to the base type. They are useful because several packages redefine the basic C/C++ types to their own types. For instance, one can define the type `real` to represent a `double`. In that case, `real` can be used to specify the variable types inside the package file interpreted by **tolua**, but only if we include the following definition before any use of the type `real`.

```
typedef double real;
```

Otherwise, `real` would be interpreted as a user defined type and would not be mapped to Lua numbers.

Including real header files

In the package file, we must specify which are the real header files that should be included so that the generated code can access the constants, variables, functions, and classes we are binding. Any line in the package file beginning with a **$** (except $[hclp]file, $[, and $] lines) is inserted into the generated binding C/C++ code without any change, but the elimination of the **$** itself. We use this feature to include the real header files. So, our package files will usually start with a set of **$** beginning lines specifying the files that must be included, that is, the files the package file is based on.

```
/* specify the files to be included */

$#include "header1.h"                // include first header
$#include "header2.h"                // include second header
```

As illustrated, **tolua** also accepts comments, using C or C++ convention, inside the package file. Nested C-like comments can also be used.

Also note that files included with `$cfile` or `$hfile` don't need to be included using this method, this is done automatically by **tolua**.

In the following sections, we describe how to specify the C/C++ code we want to bind to Lua. The formats are simplified valid C/C++ statements.

Binding constants

To bind constants, **tolua** accepts both define's and enum's. For define's the general format is:

#define *NAME* [*VALUE*]

The value, as showed above, is optional. If such a code is inserted inside the file being processed, **tolua** generates a code that allows the use of *NAME* as a Lua global variable that has the corresponding C/C++ constant value. Only numeric constants are accepted.

New on tolua++: All other preprocessor directives are ignored.

For enum's, the general format is:

```
enum {
  NAME1 [ = VALUE1 ] ,
  NAME2 [ = VALUE2 ] ,
  ...
  NAMEn [ = VALUEn ]
};
```

Similarly, **tolua** creates a set of global variables, named *NAMEi*, with their corresponding values.

Binding external variables

Global extern variables can also be exported. In the cleaned header file they are specified as:

```
[extern] type var;
```

tolua binds such declarations to Lua global variables. Thus, in Lua, we can access the C/C++ variable naturally. If the variable is non constant, we can also assign the variable a new value from Lua. Global variables that represent arrays of value can also be bound to Lua. Arrays can be of any type. The corresponding Lua objects for arrays are Lua tables indexed with numeric values; however, be aware that index 1 in Lua is mapped to index 0 in an C/C++ array. Arrays must be pre dimensioned. For instance:

```
double v[10];
```

New on tolua++: External variables can use the `tolua_readonly` modifier (see Additional Features)

Binding functions

Functions are also specified as conventional C/C++ declarations:

```
type funcname (type1 par1[, type2 par2[,...typeN parN]]);
```

The returned type can be `void`, meaning no value is returned. A function can also have no parameter. In that case, `void` may be specified in the place of the list of parameters. The parameter types must follow the rules already posted. **tolua** creates a Lua function binding the C/C++ function. When calling a function from Lua, the parameter types must match the corresponding C/C++ types, otherwise, **tolua** generates an error and reports which parameter is wrongly specified. If a parameter name is omitted, **tolua** names it automatically, but its type should be a basic type or user type previously used.

Arrays

tolua also deals with function or method parameters that represent arrays of values. The nice thing about arrays is that the corresponding Lua tables have their values updated if the C/C++ function changes the array contents.

The arrays must be pre dimensioned. For instance:

```
void func (double a[3]);
```

is a valid function declaration for **tolua** and calling this function from Lua would be done by, for instance:

```
p = {1.0,1.5,8.6}
func (p)
```

The array dimension need not be a constant expression; the dimension can also be specified by any expression that can be evaluated in run time. For instance:

```
void func (int n, int m, double image[n*m]);
```

is also valid since the expression `n*m` is valid in the binding function scope. However, be aware that **tolua** uses dynamic allocation for binding this function, what can degrade the performance.

Despite the dimension specification, it is important to know that all arrays passed to the actual C/C++ function are in the local scope of the binding function. So, if the C/C++ function being called needs to hold the array pointer for later use, the binding code will *not* work properly.

Overloaded functions

Overloaded functions are accepted. Remember that the distinction between two functions with the same name is made based on the parameter types that are mapped to Lua. So, although

```
void func (int a);
void func (double a);
```

represent two different functions in C++, they are the same function for **tolua**, because both `int` and `double` are mapped to the same Lua type: `number`.

Another tricky situation occurs when expecting pointers. Suppose:

```
void func (char* s);
void func (void* p);
void func (Object1* ptr);
void func (Object2* prt);
```

Although these four functions represent different functions in C++, a Lua statement like:

```
func(nil)
```

matches all of them.

It is important to know that **tolua** decides which function will be called in run-time, trying to match each provided function. **tolua** first tries to call the last specified function; if it fails, **tolua** then tries the previous one. This process is repeated until one function matches the calling code or the first function is reached. For that reason, the mismatching error message, when it occurs, is based on the first function specification. When performance is important, we can specify the most used function as the last one, because it will be tried first.

tolua allows the use of overloaded functions in C, see Renaming for details.

Default parameter values

The last function parameters can have associated default values. In that case, if the function is called with fewer parameters, the default values are assumed. The format to specify the default values is the same as the one used in C++ code:

```
type funcname (..., typeN-1 parN-1 [= valueN-1], typeN parN [= valueN]);
```

toLua implements this feature without using any C++ mechanism; so, it can be used also to bind C functions.

We can also specify default values for the elements of an array (there is no way to specify a default value for the array itself, though). For instance:

```
void func (int a[5]=0);
```

sets the default element values to zero, thus the function can be called from Lua with an uninitialized table.

For Lua object types (`lua_Object`), **tolua** defines a constant that can be used to specify `nil` as default value:

```
void func (lua_Object lo = TOLUA_NIL);
```

New on tolua++: C++ class constructors are valid as default parameters. For example:

```
void set_color(const Color& color = Color(0,0,0));
```

Multiple returned values

In Lua, a function may return any number of values. **tolua** uses this feature to simulate values passed by reference. If a function parameter is specified as a pointer to or reference of a basic type or a pointer to or reference of a pointer to an user defined type, **tolua** accepts the corresponding type as input and returns, besides the conventional function returned value, if any, the updated parameter value.

For instance, consider a C function that swaps two values:

```
void swap (double* x, double* y);
```

or

```
void swap (double& x, double& y);
```

If such a function is declared in the package file, **tolua** binds it as a function receiving two numbers as input and returning two numbers. So, a valid Lua code would be:

```
x,y = swap(x,y)
```

If the input values are not used, the use of default parameter value allows calling the function from Lua without specifying them:

```
void getBox (double* xmin=0, double* xmax=0, double* ymin=0, double* ymax=0);
```

In Lua:

```
xmin, xmax, ymin, ymax = getBox()
```

With user defined types, we would have for instance:

```
void update (Point** p);
```

or

```
void update (Point*& p);
```

Binding struct fields

User defined types are nicely bound by **tolua**. For each variable or function type that does not correspond to a basic type, **tolua** automatically creates a tagged userdata to represent the C/C++ type. If the type corresponds to a struct, the struct fields can be directly accessed from Lua, indexing a variable that holds an object of such a type. In C code, these types are commonly defined

using typedef's:

```
typedef struct [name] {
   type1 fieldname1;
   type2 fieldname2;
   ...
   typeN fieldnameN;
} typename;
```

If such a code is inserted in the package file being processed, **tolua** allows any variable that holds an object of type *typename* to access any listed field indexing the variable by the field name. For instance, if `var` holds a such object, *var.fieldnamei* accesses the field named *fieldnamei*.

Fields that represent arrays of values can also be mapped:

```
typedef struct {
  int x[10];
  int y[10];
} Example;
```

Binding classes and methods

C++ class definitions are also supported by **tolua**. Actually, the **tolua** deals with single inheritance and polymorphism in a natural way. The subsections below describe what can be exported by a class definition.

Specifying inheritance

If `var` is a Lua variable that holds an object of a derived class, `var` can be used wherever its base class type is expected and `var` can access any method of its base class. For this mechanism to take effect, we must indicate that the derived class actually inherits the base class. This is done in the conventional way:

```
class classname : public basename
{
 /* class definition */
};
```

In this case, the definition of `basename` needs to appear before `classname` if the inheritance properties are to be taken advantage of from lua.

Multiple inheritance

tolua++ (starting from version 1.0.4) supports multiple inheritance by allowing you to access the extra parents 'manually'.

For example, consider the following class:

```
class Slider : public Widget, public Range {
 ...
};
```

An object of type 'Slider' will fully retain its inheritance with Widget, and will contain a 'member' of type Range, which will return the object cast to the correct base type.

For example:

```
slider = Slider:new()
slider:show() -- a Widget method

slider:set_range(0, 100) -- this won't work, because
                         -- set_range is a method from Range

slider.__Range__:set_range(0, 100) -- this is the correct way
```

This is an experimental feature.

Specifying exported members and methods

As for struct fields, class fields, static or not, can be exported. Class methods and class static methods can also be exported. Of course, they must be declared as public in the actual C++ code (the `public:` keyword may appear in the package files, it will be ignored by **tolua**).

For each bound class, **tolua** creates a Lua table and stores it at a variable which name is the name of the C++ class. This tables may contain other tables that represent other tables, the way C++ classes may contain other classes and structs. Static exported fields are accessed by indexing this table with the field names (similar to struct fields). Static methods are also called using this table, with a

colon. Non static exported fields are accessed by indexing the variable that holds the object. Class methods follow the format of the function declaration showed above. They can be accessed from Lua code using the conventional way Lua uses to call methods, applied of course to a variable that holds the appropriate object or to the class table, for static methods.

There are a few special methods that are also supported by **tolua**. Constructors are called as static methods, named `new`, `new_local` (on **tolua++**), or calling the class name directly (also on **tolua++**, see below for the difference betwheen these methods). Destructors are called as a conventional method called `delete`.

Note that **tolua** does support overload. This applies even for constructors. Also note that the `virtual` keyword has no effect in the package file.

The following code exemplifies class definitions that can be interpreted by **tolua**.

```
class Point {
   static int n;    // represents the total number of created Points
   static int get_n();    // static method

   double x;        // represents the x coordinate
   double y;        // represents the y coordinate

   static char* className (void);   // returns the name of the class

   Point (void);                          // constructor 1
   Point (double px, double py);          // constructor 2
   ~Point (void);                         // destructor

   Point add (Point& other);              // add points, returning another one
};

class ColorPoint : public Color {
   int red;      // red color component [0 - 255]
   int green;    // green color component [0 - 255]
   int blue;     // blue color component [0 - 255]

   ColorPoint (double px, double py, int r, int g, int b);
};
```

If this segment of code is processed by **tolua**, we would be able to write the following Lua statements:

```
p1 = Point:new(0.0,1.0)
p2 = ColorPoint:new(1.5,2.2,0,0,255)
print(Point.n)                      -- would print 2
print(Point:get_n())                -- would also print 2
p3 = p1:add(p2)
local p4 = ColorPoint()
print(p3.x,p3.y)                    -- would print 1.5 and 3.2
print(p2.red,p2.green,p2.blue)      -- would print 0, 0, and 255
p1:delete()                         -- call destructor
p2:delete()                         -- call destructor
```

Note that we can only explicitly delete objects that we explicitly create. In the example above, the point `p3` will be garbage-collected by **tolua** automatically; we cannot delete it.

New on tolua++: Also note that `p4` is created by calling the class name directly (`ColorPoint()`); this has the same effect as calling `new_local`, which leaves the object to be deleted by the garbaje collector, and it should not be deleted using `delete`. For each constructor on the pkg, one `new`, `new_local` and `.call` callback for the class is created.

Of course, we need to specify only the methods and members we want to access from Lua. Sometimes, it will be necessary to declare a class with no member or method just for the sake of not breaking a chain of inheritances.

Using Regular functions as class methods

tolua++ (starting from version 1.0.5) uses the keyword **tolua_outside** to specify regular functions as methods and static methods of a class or struct. For example:

```
/////////////// position.h:

typedef struct {

 int x;
 int y;
} Position;

Position* position_create();
void position_set(Position* pos, int x, int y);

/////////////// position.pkg:

struct Position {
```

```
 int x;
 int y;

 static tolua_outside Position* position_create @ create();
 tolua_outside void position_set @ set(int x, int y);
};

--------------- position.lua

local pos = Position:create()

pos:set(10, 10)
```

Note that the **position_set** method takes a pointer to **Position** as its first parameter, this is ommited on the **tolua_outside** declaration. Also note that we cannot name our methods **new** or **new_local**, or as overloaded operators (see next section), this will result in undefined behaviour.

Overloaded operators

tolua automatically binds the following binary operators:

```
     operator+   operator-   operator*   operator/
     operator<   operator>=  operator==  operator[]
```

For the relational operators, **toLua** also automatically converts a returned `0` value into `nil`, so *false* in C becomes *false* in Lua.

As an example, suppose that in the code above, instead of having:

```
   Point add (Point& other);              // add points, returning another one
```

we had:

```
   Point operator+ (Point& other);        // add points, returning another one
```

In that case, in Lua, we could simply write:

```
p3 = p1 + p2
```

The indexing operator (`operator[]`) when receiving a numeric parameter can also be exported to Lua. In this case, **tolua** accepts reference as returned value, even for basic types. Then if a reference is returned, from Lua, the programmer can either get or set the value. If the returned value is not a reference, the programmer can only get the value. An example may clarify: suppose we have a vector class and bind the following operator:

```
   double& operator[] (int index);
```

In this case, in Lua, we would be able to write: `value = myVector[i]` and also `myVector[i] = value`, which updates the C++ object. However, if the bound operator was:

```
   double operator[] (int index);
```

we would only be able to write: `value = myVector[i]`.

Free functions (i.e., not class members) that overload operators are not supported.

Cast operators

New on tolua++ (versions 1.0.90 and up): casting operators are also supported. For example:

```
/////////////// node.h

// a class that holds a value that can be of type int, double, string or Object*
class Node { // tolua_export

private:
 union {
  int int_value;
  double double_value;
  string string_value;
  Object* object_value;
 };

// tolua_begin
public:

 enum Type {
  T_INT,
  T_DOUBLE,
  T_STRING,
  T_OBJECT,
  T_MAX,
```

```
 };

 Type get_type();

 operator int();
 operator double();
 operator string();
 operator Object*();
};
// tolua_end
```

tolua++ will produce code that calls the operators by casting the object Node (using C++ `static_cast`), and register them inside the class as ".typename". For example:

```
-- node.lua

local node = list.get_node("some_node") -- returns a Node object

if node.get_type() == Node.T_STRING then

 print("node is "..node[".string"]())

elseif node.get_type() == Node.T_OBJECT then

 local object = node[".Object*"]()
 object:method()
end
```

Binding Properties

tolua++ (starting from version 1.0.6) supports declaration of class propeties, using the `tolua_property` keyword. A property will look like a 'field' of the class, but it's value will be retrieved using class methods. For example:

```
/////////////// label.h

class Label {

public:

 string get_name();
 void set_name(string p_name);

 Widget* get_parent();
};

/////////////// label.pkg
class Label {

 tolua_property string name;

 tolua_readonly tolua_property Widget* parent;
};

--------------- label.lua

local label = Label()

label.name = "hello"
print(label.name)

label.parent:show()
```

Property types

A property can have differt types, which determine how it's value will be set and retrieved. **tolua++** comes with 3 different built-in types:

- `default` will use 'get_name' and 'set_name' methods to access a property called 'name'
- `qt` will use 'name' and 'setName'
- `overload` will use 'name' and 'name' (as in 'string name(void);' to get and 'void name(string);' to set)

The property type can be appended at the end of the 'tolua_property' keyword on the declaration:

```
    tolua_property__qt string name;
```

When no type is specified, `default` will be used, but this can be changed (see below).

Changing the default property type

The default property type can be changed using the 'TOLUA_PROPERTY_TYPE' macro. This will change the default type from the point of its invocation, until the end of the block that contains it. For example:

```
TOLUA_PROPERTY_TYPE(default); // default type for the 'global' scope

namespace GUI {

 class Point {

  tolua_property int x; // will use get_x/set_x
  tolua_property int y; // will use get_y/set_y
 };

 TOLUA_PROPERTY_TYPE(qt); // changes default type to 'qt' for the rest of the 'GUI' namespace

 class Label {

  tolua_property string name; // will use name/setName
 };
};

class Sprite {

 tolua_property GUI::Point position; // will use get_position/set_position

 tolua_property__overload string name; // will use name/name
};
```

Adding custom property types

Custom property types can be added by redefining the function "get_property_methods_hook" (see Customizing tolua++ for more details). The functions takes the property type and the name, and returns the setter and getter function names. For example:

```
/////////////// custom.lua

function get_property_methods_hook(ptype, name)

 if ptype == "hungarian_string" then

  return "sGet"..name, "Set"..name
 end

 if ptype == "hungarian_int" then

  return "iGet"..name, "Set"..name
 end
 -- etc
end

/////////////// label.pkg
class Label {

 tolua_property__hungarian_string string Name; // uses 'sGetName' and 'SetName'

 tolua_property__hungarian_int string Type; // uses 'iGetType' and 'SetType'
};
```

Class Templates

One of the additional features of **tolua++** is the support for class templates, by using the `TOLUA_TEMPLATE_BIND` directive. For example:

```
class vector {

 TOLUA_TEMPLATE_BIND(T, int, string, Vector3D, double)

 void clear();
 int size() const;

 const T& operator[](int index) const;
 T& operator[](int index);
 void push_back(T val);

 vector();
 ~vector();
};
```

The `TOLUA_TEMPLATE_BIND` directive has to be the first thing on the class declaration, otherwise it will be ignored. This code will create 4 versions of the class `vector`, one for each type specified on the `TOLUA_TEMPLATE_BIND` parameters, each replacing the macro `T` (specified as the first argument of `TOLUA_TEMPLATE_BIND`). Thus, the functions `operator[]`, `&operator[]` and `push_back` will have

different signatures on each version of the object. The objects will be recognized as `vector<type>` on further declarations, and the name of the table on Lua will be `vector_type_`. Thus, the following Lua code could be used:

```
string_vector = vector_string_:new_local()
string_vector:push_back("hello")
string_vector:push_back("world")
print(string_vector[0].." "..string_vector[1])
```

Similarily, a template with more than 1 macro could be bound, and it could also inherit from another template:

```
class hash_map : public map<K,V> {

 TOLUA_TEMPLATE_BIND(K V, int string, string vector<double>)

 V get_element(K key);
 void set_element(K key, V value);

 hash_map();
 ~hash_map();
};
```

In this example, one of the objects has another template as one of its types, so it will be recognized as `hash_map<string,vector<double> >` while its constructor will be on the Lua table hash_map_string_vector_double___ (see Type Renaming for a better way to access these objects).

Note that due to the complexity in the definition of some templates, you should be careful on how you declare them. For example, if you create an object with type `hash_map<string,vector<double> >` and then declare a variable with type `hash_map<string, vector<double> >` (note the space between string and vector), the type of the variable will not be recognized. The safest way is to declare a typedef, and use that to use each type (this is also a common practice on C++ programming). For example, using the previous declaration of `vector`:

```
typedef vector VectorInt;

VectorInt variable;
```

`TOLUA_TEMPLATE_BIND` can be used with more than one parenthesis to open and close, in order to be valid as a macro inside a regular .h file. The `TOLUA_TEMPLATE_BIND` macro is declared on `tolua.h` as:

```
#define TOLUA_TEMPLATE_BIND(x)
```

Also, the parameters can have double quotes. Thus, the following uses are valid:

```
TOLUA_TEMPLATE_BIND((T, int, float)) // to be used inside a real header file
TOLUA_TEMPLATE_BIND("K V", "string string", int double)
```

Function templates are not supported on this version.

Module definition

tolua allows us to group constants, variables, and functions in a module. The module itself is mapped to a table in Lua, and its constants, variables, and functions are mapped to fields in that table. The general format to specify a module is:

```
module name
{
      ... // constant, variable, and function declarations
}
```

Thus, if we bound the following module declaration:

```
module mod
{
 #define N
 extern int var;
 int func (...):
}
```

In Lua we would be able to access such features by indexing the module: `mod.N`, `mod.var`, `mod.func`.

Renaming constants, variables and functions

When exporting constants, variable, and functions (members of a class or not), we can rename them, such that they will be bound with a different name from their C/C++ counterparts. To do that, we write the name they will be referenced in Lua after the character `@`. For instance:

```
extern int cvar @ lvar;

#define CNAME @ LNAME
```

```
enum {
  CITEM1 @ LITEM1,
  CITEM2 @ LITEM2,
  ...
};

void cfunc @ lfunc (...);

class T
{
   double cfield @ lfield;
   void cmeth @ lmeth (...);
   ...
};
```

In such a case, the global variable `cvar` would be identified in Lua by `lvar`, the constant `CNAME` by `LNAME`, and so on. Note that class cannot be renamed, because they represent types in C.

This renaming feature allows function overload in C, because we can choose to export two different C functions with a same Lua name:

```
void glVertex3d @ glVertex (double x, double y, double z=0.0);
void glVertexdv @ glVertex (double v[3]=0.0);
```

Renaming Types

Types can be renamed using the `$renaming` directive on pkg files, using the format:

```
$renaming real_name @ new_name
```

The parameters to renaming can be Lua *patterns*. For example:

```
$renaming ^_+ @
$renaming hash_map<string,vector<double> > @ StringHash
```

The first example will remove all underscores at the beginning of all types, the second will rename the template type hash_map<string,vector<double> > to StringHash. Once renamed, the Lua table for each type can be accessed only by their new name, for example: `StringHash:new()`

Storing additional fields

Finally, it is important to know that even though the variables that hold C/C++ objects are actually tagged userdata for Lua, **tolua** creates a mechanism that allows us to store any additional field attached to these objects. That is, these objects can be seen as conventional Lua tables.

```
obj = ClassName:new()

obj.myfield = 1  -- even though "myfield" does not represent a field of ClassName
```

Such a construction is possible because, if needed, **tolua** automatically creates a Lua table and associates it with the object. So that, the object can store additional fields not mapped to C/C++, but actually stored in the conjugate table. The Lua programmer accesses the C/C++ features and these additional fields in an uniform way. Note that, in fact, these additional fields overwrite C/C++ fields or methods when the names are the same.

Additional features on tolua++

Multiple variable declarations

Multiple variables of the same type can be declared at the same time, for example:

```
float x,y,z;
```

will create 3 different variables of type float. Make sure you don't leave any spaces between the commas, as that will raise a parse error.

tolua_readonly

Any variable declaration can use the `tolua_readonly` modifier, to ensure that the variable is read-only, even when its type is not `const`. Example:

```
class Widget {

 tolua_readonly string name;
};
```

This feature could be used to 'hack' the support for other unsupported things like `operator->`. Consider this example `pkg` file:

```
$hfile "node.h"
$#define __operator_arrow operator->()
$#define __get_name get_name()
```

And on the file `node.h`:

```
template class<T>
class Node { // tolua_export

private:
 string name;
 T* value;

public:

 T* operator->() {return value;};
 string get_name() {return name;};

 // tolua_begin

 #if 0
 TOLUA_TEMPLATE_BIND(T, Vector3D)

 tolua_readonly __operator_arrow @ p;
 tolua_readonly __get_name @ name;
 #endif


 Node* next;
 Node* prev;

 void set_name(string p_name) {name = p_name;};

 Node();
};
// tolua_end
```

While not a pretty thing to do to a header file, this accomplishes a number of things:

- The method `operator->()` can be used from Lua by calling the variable `p` on the object.
- The method `get_name()` can be using from Lua by calling the variable `name` on the boject.

Example lua usage:

```
node = Node_Vector3D_:new_local()
-- do something with the node here --
print("node name is "..node.name)
print("node value is ".. node.p.x ..", ".. node.p.y ..", ".. node.p.z)
```

Since **tolua++** ignores all preprocessor directives (except for #define), `node.h` remains a valid C++ header file, and also a valid source for **tolua++**, eliminating the need to maintain 2 different files, even for objects with unusual features such as these ones.

The ability to rename functions as variables might be expanded on future versions.

Defining values on command line

Starting from version 1.0.92, the command line option `-E` allows you to introduce values into to the luastate where **tolua++** runs, similar to GCC's `-D`. For example:

```
$ tolua++ -E VERSION=5.1 -E HAVE_ZLIB package.pkg > package_bind.cpp
```

This will add 2 fields to the global table `_extra_parameters`: "VERSION", with the string value "5.1", and "HAVE_ZLIB" with the boolean value `true`. For the moment, there is no way to 'use' these values, except in custom scripts defined by the user (see customizing tolua++ for details).

Using C++ typeid

Starting from version 1.0.92, the command line option `-t` is available, which generates a list of calls to the empty macro `Mtolua_typeid`, with its C++ `type_info object`, and the name used by tolua++ to identify the type. For example, if you have a package that binds 2 classes, `Foo` and `Bar`, using `-t` will produce the following output:

```
#ifndef Mtolua_typeid
#define Mtolua_typeid(L,TI,T)
#endif
 Mtolua_typeid(tolua_S,typeid(Foo), "Foo");
 Mtolua_typeid(tolua_S,typeid(Bar), "Bar");
```

The implementation of Mtolua_typename is left as an exercise to the user.

Exported utility functions

tolua uses itself to export some utility functions to Lua, including its object-oriented framework. The package file used by **tolua** is shown below:

```
module tolua
{
 char* tolua_bnd_type @ type (lua_Object lo);
 void tolua_bnd_takeownership @ takeownership (lua_Object lo);
 void tolua_bnd_releaseownership @ releaseownership (lua_Object lo);
 lua_Object tolua_bnd_cast @ cast (lua_Object lo, char* type);
 void tolua_bnd_inherit @ inherit (lua_Object table, lua_Object instance);

/* for lua 5.1 */
 void tolua_bnd_setpeer @ setpeer (lua_Object object, lua_Object peer_table);
 void tolua_bnd_getpeer @ getpeer (lua_Object object);
}
```

tolua.type (*var*)

Returns a string representing the object type. For instance, `tolua.type(tolua)` returns the string `table` and `tolua.type(tolua.type)` returns `cfunction`. Similarly, if `var` is a variable holding a user defined type `T`, `tolua.type(var)` would return `const T` or `T`, depending whether it is a constant reference.

tolua.takeownership (*var*)

Takes ownership of the object referenced *var*. This means that when all references to that object are lost, the objects itself will be deleted by lua.

tolua.releaseownership (*var*)

Releases ownership of the object referenced by *var*.

tolua.cast (*var*, *type*)

Changes the metatable of *var* in order to make it of type *type*. *type* needs to be a string with the complete C type of the object (including namespaces, etc).

tolua.inherit (*table*, *var*)

(new on **tolua++**) Causes **tolua++** to recognise *table* as an object with the same type as *var*, and to use *var* when necesary. For example, consider this method:

```
     void set_parent(Widget* p_parent);
```

A lua object could be used like this:

```
local w = Widget()
local lua_widget = {}
tolua.inherit(lua_widget, w)

set_parent(lua_widget);
```

Remember that this will only cause the table to be recognised as type 'Widget' when necesary. To be able to access Widget's methods, you'll have to implement your own object system. A simple example:

```
lua_widget.show = Widget.show

lua_widget:show() -- this will call the 'show' method from 'Widget', using the lua
                  -- table as 'self'. Since lua_widget inherits from a widget instance,
                  -- tolua++ will recognise 'self' as a 'Widget', and call the method
```

Of course a better way would be to add a __index metamethod for the lua object.

Similarily, to implement virtual functions, you'll need to create a c++ object that inherits from the desired type, implement its virtual functions, and use that to inherit from lua. The object would have a reference to the lua table, and call its methods from the c++ virtual methods.

Note: the current implementation (as of version 1.0.6) stores the C instance inside the lua table on the field ".c_instance", and looks that up when necesary. This might change in the future, so it is recommended to use an alternative way to store the C instance to use with your own object system.

tolua.setpeer (*object*, *peer_table*) (lua 5.1 only)

Sets the table as the object's *peer* table (can be `nil`). The peer table is where all the custom lua fields for the object are stored. When compiled with lua 5.1, **tolua++** stores the peer as the object's *envirnment table*, and uses uses `lua_gettable/settable` (instead of `lua_rawget/set` for lua 5.0) to retrieve and store fields on it. This allows us to implement our own object system on our table (using metatables), and use it as a way to inherit from the userdata object. Consider an alternative to the previous example:

```
-- a 'LuaWidget' class
LuaWidget = {}
LuaWidget.__index = LuaWidget

function LuaWidget:add_button(caption)

 -- add a button to our widget here. 'self' will be the userdata Widget
end


local w = Widget()
local t = {}
setmetatable(t, LuaWidget) -- make 't' an instance of LuaWidget

tolua.setpeer(w, t) -- make 't' the peer table of 'w'

set_parent(w) -- we use 'w' as the object now

w:show() -- a method from 'Widget'
w:add_button("Quit") -- a method from LuaWidget (but we still use 'w' to call it)
```

When indexing our object, the peer table (if present) will be consulted first, so we don't need to implement our own __index metamethod to call the C++ functions.

tolua.getpeer (*object*) (lua 5.1 only)

Retrieves the peer table from the object (can be `nil`).

Embedded Lua code

tolua allows us to embed Lua code in the C/C++ generated code. To do that, it compiles the specified Lua code and creates a C constant string, storing the corresponding bytecodes, in the generated code. When the package is opened, such a string is executed. The format to embed Lua code is:

```
$[

embedded Lua code
...

$]
```

As an example consider the following .pkg excerpt:

```
/* Bind a Point class */
class Point
{
 Point (int x, int y);
 ~Point ();
 void print ();
 ...
} CPoint;

$[

-- Create a Point constructor
function Point (self)
 local cobj = CPoint:new(self.x or 0, self.y or 0)
 tolua.takeownership(cobj)
 return cobj
end

$]
```

Binding such a code would allow us to write the following Lua code:

```
p = Point{ x=2, y=3 }
p:print()
...
```

Customizing tolua++

tolua++ calls empty functions at specific points of its execution. This functions can be redefined on a separate lua file (and included using the -L command line option) and be used to control the way **tolua++** behaves. This is the list of functions (taken from basic.lua on the **tolua++** source):

```
-- called right after processing the $[ichl]file directives,
-- right before processing anything else
-- takes the package object as the parameter
function preprocess_hook(p)
 -- p.code has all the input code from the pkg
end


-- called for every $ifile directive
-- takes a table with a string called 'code' inside, the filename, and any extra arguments
-- passed to $ifile. no return value
function include_file_hook(t, filename, ...)

end

-- called after processing anything that's not code (like '$renaming', comments, etc)
-- and right before parsing the actual code.
-- takes the Package object with all the code on the 'code' key. no return value
function preparse_hook(package)

end


-- called after writing all the output.
-- takes the Package object
function post_output_hook(package)

end

-- called at the beginning of the main parser function, with the code being parsed as a parameter
-- it can return nil if nothing was foind, or the contents of 'code', modified by the function
-- Usually a parser hook will search the beginning of the string for a token, and if it finds
-- anything, return the string without that token (after doing whatever it has to do with the token).
function parser_hook(code)

end


-- called from classFunction:supcode, before the call to the function is output
-- the classFunction object is passed.
function pre_call_hook(f)

end

-- called from classFunction:supcode, after the call to the function is output
-- the classFunction object is passed.
function post_call_hook(f)

end

-- called before the register code is output
function pre_register_hook(package)

end

-- called to output an error message
function output_error_hook(...)
 return string.format(...)
end
```

Handling custom types

Starting from version 1.0.93, it is possible to specify custom functions to handle certain types. There are 3 types of functions: a `'push function'`, used to push the C value onto the Lua stack, a `'to function'`, used to retrieve the value from the Lua stack, and return it as a C value, and an `'is function'`, used to check if the value on the stack is valid (or convertible to a C value by the `to function`). These functions are modelled upon `tolua_pushusertype`, `tolua_tousertype` and `tolua_isusertype`, declared in `tolua++.h`.

A number of arrays found in `basic.lua` are used to specify these functions:

```
-- for specific types
_push_functions = {}
```

```
_is_functions = {}
_to_functions = {}

-- for base types
_base_push_functions = {}
_base_is_functions = {}
_base_to_functions = {}
```

Example (using the -L command line option):

```
_is_functions['Vector3'] = 'custom_is_vector3' -- checks for a 3d vector
                                               -- (either userdata, or a table with 3 values)
_to_functions['Vector3'] = 'custom_to_vector3' -- convertes the eventual table to a Vector3

_base_push_functions['Widget'] = 'custom_push_widget' -- pushes anything that inherits from Widget
```

The `_base` tables are used to lookup functions for types that are up in the inheritance chain.

Access

Starting from version 1.0.7, all objects have a an `access` flag, which determines the object's access inside its container. Container objects also have a member called `curr_member_access`, which determines the access of each child object at the moment of its addition to the container. If the `access` flag has the value `nil` (default), `false` or `0`, the object is public. Otherwise, the object is not public, and **tolua++** will not export any code for that object (and any objects it may contain).

Another 'interesting' function is `extract_code`, defined on basic.lua, which extracs the code from files included with $cfile and $hfile (by looking for tolua_begin/end and tolua_export).

Compatibility with older versions.

tolua++ <1.0.90

Version 1.0.90 of **tolua++** introduces 2 small API changes the might be incompatible with older versions on some cases:

TEMPLATE_BIND

`TEMPLATE_BIND` is deprecated. Use `TOLUA_TEMPLATE_BIND` instead. Also, when declaring a template, the `TOLUA_TEMPLATE_BIND` statement has to be the first thing inside the class declaration, otherwise it will be ignored. This fixes a possible problem with nested template declarations.

Retrieving Objects

When passing a full userdata to a function that accepts light userdata parameters (void*), the **tolua++** library function `tolua_touserdata` will detect the full userdata and dereference the void** pointer if necesary. This is a change on the function's behaviour (which used to return the pointer as-is). This allows us to pass pointers to objects to a function that accepts void* pointers. Note that this was a problem when switching from **toLua** version 4 to version 5 (and **tolua++** versions < 1.0.90).

toLua 4

Retrieving Objects

Users switching from **tolua** v4 should know that **tolua++** stores the objects as `void**` on Lua, so when retrieving an object from the luastate using `lua_touserdata`, the pointer should be dereferenced. The library function `tolua_tousertype` should work as expected. Example:

```
lua_pushglobal(lua_state, "get_Object");
lua_call(lua_state, 0, 1);  // calling a function that returns an Object

Object *new_object = (Object*)(*lua_touserdata(lua_state, -1));
or
Object *new_object = (Object*)tolua_tousertype(lua_state, -1, NULL);
```

C++ Strings

tolua++ binds the c++ type `std::string` as a basic type, passing it to Lua as a regular string (using the method `c_str()`). This feature can be turned off with the command line option `-S`.

Operators

The list of supported operators has changed, see Binding classes and methods for more information.

toLua 5

Class destructors.

With every class constructor, **tolua++** exports a 'local' constructor, to create an instance of the class that is owned by the lua state. To implement this, **tolua++** will also export a function that will `delete` the class. There are 2 options to prevent this:

Using the **-D** command line option will turn this off completely. Destructor functions will only be exported when a destructor for the class is explicitly declared, or when there is a function or method that returns an object of the type by value (this is compatible with **tolua**'s behaviour).

Using the `TOLUA_PROTECTED_DESTRUCTOR` directive inside the class declaration, to specify that the class has a private or protected destructor. In this case, no destructor will be exported for that class.

`operator[]` index

Users switching from **tolua** v5 should know that **tolua** 5 substracts 1 from the index on operator[] functions, for compatibility with lua's method for indexing arrays (1 is the first element). This feature is turned off by default on **tolua++** (making it compatible with **tolua** 4). It can be turned back on with the command line option `-1`

C++ Strings

(see c++ strings on **tolua** 4 below)

Changes since v. 3.0

- Support for binding arrays as variables and struct/class fields;
- Support for embedding Lua code into the generated binding code;
- New utility functions: *cast* and *takeownership*;
- Option to create the corresponding header file of the binding code;
- New "close" package function;
- Fixed bug on cloning objects in C++;
- Fixed bug on enum and struct parsing;

Changes since v. 2.0

- There is a new executable parser;
- Support for multiple Lua states is provided;
- Support for module definition is provided;
- Global variables is now directly bound to Lua global variables;
- Constness of user defined types is preserved in Lua;
- Support for multiple returned values from C/C++ is provided (simulating parameters passed by reference);
- Constants, variables, and functions bound to Lua can have different names from their C/C++ counterparts;
- Object-oriented framework (and other utility functions) used in **tolua** is now exported for Lua programmers;

Incompatibilities

Lua code based on **tolua** v2.* should run with no change on **tolua** v3.0. Although, it may be necessary to change the .pkg file in order to get the same behavior. The following incompatibilities exist:

- Parameters defined as pointer to basic types are no longer converted to arrays of dimension one; they are now considered parameters passed by reference.
- Automatic initialization for C++ code must be explicitly requested when using the new parser;
- Global variables are no longer mapped to a table; the definition of a module including the global variables may be used to simulate the old behavior;
- The initialization function is no longer toLua_*package*_open but tolua_*package*_open, without the capital letter (sorry!).

Changes since v. 1.*

- The binding code should run much faster;
- The *cleaned header file* extension should now be `.pkg` instead of `.L`;
- Type modifiers is now accepted (though the current version ignores `const`'s);
- Returning object by value is accepted and memory allocation is controlled by Lua garbage collection;
- Overloaded functions/methods are accepted;
- Parameters with default values are accepted;
- Some overloaded operators are automatically bound.

Credits

Luiz Henrique de Figueiredo had the idea of creating a tool to automatically bind C code to Lua. L.H.F. wrote the very first version of such a tool (that bound C functions, variables, and constants) in *awk*. At that time, Waldemar Celes (now the main author) was only responsible for the C code that supported the generated binding code.

While working at NGD Studios, Ariel Manzur made some changes to tolua4 for their game engine. After the release of tolua5, having left NGD, enough changes were made to tolua to justify a separate release (with Waldemar's blessing :-)

Availability

tolua++ is freely available by http. The software provided hereunder is on an "as is" basis, and the author has no obligation to provide maintenance, support, updates, enhancements, or modifications.

This document was created by Waldemar Celes With modifications by Ariel Manzur/
Last update: Sept 2003