OOLua

2.0.1

Generated by Doxygen 1.8.6

Wed Aug 12 2015 21:51:28

Contents

Chapter 1

Main Page

1.1 Introduction

OOLua is cross platform, test driven, dependency free and Open Source library. It uses C++03 template meta-programming and pre-processor magic internally, which can be used to generate non intrusive proxies that provide a fast bridge for the interaction of C++ classes and functions with Lua.

The library provides multiple inheritance C++ classes without using C++'s RTTI, can be compiled either with or without exception support and is easily configurable. OOLua also provides a thin and simple abstraction layer for interfacing with the Lua stack and types in a type safe manner, whilst also supporting a means to bypass the type safety using the Lua light userdata type. The library can be use in a pick and mix fashion or in its entirety, whilst still using the Lua C API.

OOLua is developed by Liam Devine who has over ten years experience using the Lua programming language, having what he considers to be a deep understanding of both it and also C++.

This is not a fully original work and was originally based on ideas from binding classes using the famous `Lunar` and `Lua Technical Note 5`.

1.2 Lua compatibility

This version of the library is compatible with the following Lua implementations

- Rio Lua 5.1, 5.2 and 5.3 `http://www.lua.org`
- LuaJIT 1.1.8 and 2.0 `http://www.luajit.org/`

1.3 Links

- Project Home `https://oolua.org`
- Library documentation `https://docs.oolua.org/2.0.1`
- Issue tracker `https://oolua.org/issues`
- Mailing list `https://oolua.org/mailinglist`

1.4 Licence

OOLua:

Copyright

Chapter 2

Building

OOLua's source code can either be dropped into the path for a project , amalgamated to a single header and source file or compiled as a static library.

2.1 Makefiles and IDE projects

OOLua does not provide solution files instead it provides Premake4 [1] scripts. Premake is a simple [2] to use IDE project or makefile generator that can be used to help create a static library or to run Library Tests

2.1.1 Premake format

premake4 [make or IDE] [target operating system]

2.1.1.1 Makefile

```
premake4 gmake []
```

- macosx
- linux

2.1.1.2 Xcode

```
premake4 xcode[] macosx
```

- 3
- 4

  Note

  macosx is required

2.1.1.3 Visual Studio

```
premake4 vs[] windows
```

- 2005
- 2008

- 2010
- 2013

  Note

  windows is required

2.1.1.4 CodeBlocks

```
premake4 codeblocks []
```

- windows
- linux
- macosx

2.2 Library limits

The "oolua_generate" Lua module provides information about the default limits for the library. It enables the generation of boilerplate code using user defined limits or regeneration of files with the default values. The details of these configurable values being :

```
return
{
    lua_params =
    {
        desc ='Maximum amount of parameters for a call to a Lua function'
        ,value=10
    }
    ,cpp_params =
    {
        desc='Maximum number of parameters a C++ function can have'
        ,value=8
    }
    ,constructor_params =
    {
        desc='Maximum amount of parameters for a constructor of a proxied type'
        ,value=5
    }
    ,class_functions =
    {
        desc='Maximum amount of class functions that can be registered for each proxied type'
        ,value=15
    }
}
```

The most common change to these options is the number of functions which can be registered for a proxy class, this limit applies individually to constant and none constant functions, base class methods that are registered in a base class do not decrease the count for a derived class.

Using the Lua interpreter to regenerate the OOLua files increasing this option whilst using default values for the remaining options:

```
lua -e "require'build_scripts.oolua_generate'.gen({class_functions=30},'include/')"
```

For convenience you do not need a version of Lua installed on a machine to run this module, Premake the project file generator used in OOLua already contains a copy of Lua 5.1 (it has some modifications to the core libraries). To generate the files with the same options as above :

```
premake4 --class_functions=30 oolua-gen
```

The module returns a table with the following functions

```
return { gen = gen, defaults=defaults, default_details=default_details
       }
```

2.3 Library Config

See Also

Library Configuration

2.4 Build Scripts

```
[make or IDE]_build.[sh or bat]
```

When these build scripts are run from the build_scripts directory they create a "../local_install" directory into which newly compiled debug and release static libraries will be place along with the library headers in a sub directory "oolua".

2.5 Test Unit scripts

```
[make or IDE]_tests.[sh or bat]
```

The scripts test the library using exceptions and error return values in both debug and release configurations. When run from the build_scripts directory these will produce compiler and test unit output saved to disk in the directory "../build_logs", if an error occurs during a test then a message to stdout will inform of where to locate the full error message and compile log. These test scripts clean up any other files produced during their running.

[1] Premake download `http://industriousone.com/premake/download`

[2] Premake quick start `http://industriousone.com/premake-quick-start`

Chapter 3

Usage

Most if not all of the code snippets shown in this document are working pieces of code taken directly from the unit test files, as such the code is always correct although it may at times not marry up to the text which surrounds it in this documentation. If you should see such a thing please report it on the issue tracker.

3.1 First look

3.1.1 Hello Moon

```
void say(char const* input)
{
    printf("%s from a standalone function\n", input);
}

OOLUA_CFUNC(say, l_say)

    void hello_minimalist_function()
    {
        using namespace OOLUA; //NOLINT(build/namespaces)
        Script vm;
        set_global(vm, "say", l_say);
        run_chunk(vm, "say('Hello Lua')");
    }
```

3.1.2 Conventions

- DSL macros are upper case and prefixed with OOLUA_
- Minimalist DSL macro names are shorter in length than Expressive names
- Public API functions and types are directly in the OOLUA namespace
- Public API function names are lower case with words separated by underscores

3.1.3 lua_State and Script

OOLua is purposely designed not to be dependent on the Script class and therefore passes around its dependency of a lua_State instance. The Script class is only a helper and anything you can do with it can be accomplished by using a Lua_function struct, calling OOLUA namespaced functions or using the Lua C API.

Script provides :

- scoping of a lua_State pointer
- access to the lua_State pointer via a cast operator and function
- methods to register types
- binding a Lua_function instance to call functions
- member functions for a little state management
- setting up the state to work with OOLua

Note

This class is not copy constructible or assignable. To accomplish this a counted reference to the lua_State would need to be maintained.
If you do not want to or can not use this class please see setup_user_lua_state

3.1.4 OOLua and the Lua stack

Lua's C API does not force you to treat the stack as such a data structure, with operations on just one end, instead for convenience it uses indices to identify stack slots for a procedure. Given that Lua is a C library, without C++'s name mangling and overloading, it also provides a function per type for pushing to the stack. Contrary to this, OOLua is a C++ library and it tries to enforce a clean stack after operations. The library therefore provides a simpler interface to the Lua stack which consists of two functions :

- push Pushes an instance to top of the Lua stack.
- pull Pulls the top element off the stack and pops it.

Most usage of OOLua will only require these two functions to interact with the stack. However, you are free to use Lua C API calls if you take into account that pull removes the top element from the stack when it is valid.

3.1.5 Library header files

OOLua provides a kitchen sink header file called oolua.h. This is a header which pulls in all the required files for using any part of the library. The file has the benefit that it maybe a good candidate for a pre compiled header, depending on the project in which it is to be used. However, a consequence of its functionality means that files which may not be required will always be pulled into files which include the header. If this is not the behaviour your project requires then there are two further headers for the proxying and exporting of classes. These files being oolua_dsl.h and oolua_dsl_export.h which are used extensively in the libraries unit tests.

3.2 Lua Types in OOLua

OOLua provides three types to help store and interact with Lua types, these are Lua_ref, Lua_function and Table.

3.2.1 Lua_ref

The Lua_ref templated class stores a reference using Lua's reference system luaL_ref and luaL_unref, along with a lua_State. The reason this class stores the lua_State is to make it difficult to use the reference with another universe. A reference from the same Lua universe, even if it is from a different lua_State, is valid to be used in the universe.

The class takes ownership of any reference passed either to the two argument constructor or the set_ref function. On going out of scope a valid reference is guaranteed to be released, you may also force a release by passing an instance to swap for which valid returns false.

There are two special values for the reference which Lua provides, both of which OOLua will treat as an invalid reference:

- LUA_REFNIL luaL_ref return value to indicate it encountered a nil object at the location the ref was asked for
- LUA_NOREF guaranteed to be different from any reference return by luaL_ref

  Template Parameters

ID	Lua type as returned by lua_type

  Note

- Universe: A call to luaL_newstate or lua_newstate creates a Lua universe and a universe is completely independent of any other universe. lua_newthread and coroutine.create, create a lua_State in an already existing universe.

  Term first heard in a Lua mailing list post by Mark Hamburg.

For your convenience there are two predefined typedefs:

- OOLUA::Lua_func_ref

```
void pullLuaFunction_luaFunctionOnStack_functionIsValid()
{
    lua_pushcclosure(*m_lua, lua_gettop, 0);
    OOLUA::Lua_func_ref lua_func;
    OOLUA::pull(*m_lua, lua_func);
    CPPUNIT_ASSERT_EQUAL(true, lua_func.valid());
}
```

- OOLUA::Lua_table_ref

```
void pullTableRef_validTableOnStack_tableIsValid()
{
    lua_createtable(*m_lua, 0, 0);
    OOLUA::Lua_table_ref table;
    OOLUA::pull(*m_lua, table);
    CPPUNIT_ASSERT_EQUAL(true, table.valid());
}
```

3.2.2 Lua_function

Calling a Lua function, from C++ code using OOLua's API, can be achieved using a Lua_function object. This is a state bound caller, and the state in which the callee will be invoked is specified either in the constructor or via the bind_script member function.

To invoke a callee, the OOLUA::Lua_function type uses a call operator. The operator's first parameter must be the callee and it can be specified using one of the following types:

- std::string A function in the bound state's global table
- OOLUA::Lua_func_ref A reference to a function
- int A valid stack index If the callee is identified via a valid stack index, then this index will remain on the stack at same absolute location after the caller has returned.

The call operator is also overloaded to enable the passing of parameters to the callee; the maximum number of parameters is defined by the configurable value "lua_params".

3.2.2.1 Calling a Lua function

When using the Script class, a Lua_function instance is initialised in the Script's constructor and is made available as a public member using the name call. A Lua_function can also be used directly either by constructing it specifically for the call or reusing a previous instance.

Global function identified by a string name:

```
void stringFunc_callsFunctionInGlobalScope_returnsTrue()
{
    m_lua->run_chunk("_G['global_name'] = function() end");
    OOLUA::Lua_function caller(*m_lua);
    CPPUNIT_ASSERT_EQUAL(true , caller("global_name"));
}
```

Lua_func_ref from a child state that is called using the Script's public member:

```
void functionRef_functionRefIsFromAChildState_returnsTrue()
{
    OOLUA::Lua_func_ref func_from_child = create_func_ref_with_child_state();
    CPPUNIT_ASSERT_EQUAL(true, m_lua->call(func_from_child) );
}
```

Valid stack index that is a function:

```
void indexFunc_passedFunctionIndex_returnsTrue()
{
    OOLUA::Lua_function caller(*m_lua);
    m_lua->load_chunk("return");
    CPPUNIT_ASSERT_EQUAL(true, caller(1));
}
```

3.2.3 Table

Table provides a simple typed C++ interface for the Lua unordered and ordered associative container of the same name. Operations which use the Lua stack ensure that the stack is the same on exit as it was on entry, OOLua tries to force a clean stack(OOLua and the Lua stack).

Any value can be retrieved or set from the table via the use of the template member functions set, at or safe_at. If the value asked for is not the correct type located in the position an error can be reported, the type of which depends on Error Reporting and the function which was called. See individual member function documentation for details.

Note

The member function try_at is only defined when exceptions are enabled for the library.

There are two helper functions for creating a OOLUA::Table both of which are named OOLUA::new_table.

```
void setValue_valueSetInLua_cppSideRepresentationHasChange()
{
    OOLUA::Table t;
    OOLUA::new_table(*m_lua, t);

    m_lua->run_chunk("func = function(t) t['a'] = 1; end");
    m_lua->call("func", t);

    int storedValue(0);
    t.at("a", storedValue);
    CPPUNIT_ASSERT_EQUAL(1, storedValue);
}
```

3.3 Proxy

3.3.1 DSL

The domain specific language(DSL) used for generating C++ bindings to Lua.

OOLua provides a DSL for defining C++ types which are to be made available to a Lua script. The intention of this DSL is to hide the implementation details whilst providing a simple and rememberable interface to perform the required actions. For the generation of function proxies, the DSL contains two sub categories named Minimalist and Expressive.

Note

"Optional" here means that extra macro parameters are optional, up to the configuration max for a specific operation.

3.3.2 Class Proxy

For a class type, the library uses a proxy as an intermediary between the two languages of Lua and C++. A proxy contains information about an exposed type, for example its hierarchical structure and functions of interest.

Generating a proxy, using the DSL, takes place between the two DSL procedures OOLUA_PROXY and OOLUA_-PROXY_END. However, alone this does not enable the generation and usage of the type within a Lua state, to do this requires a three part process. The tasks of the process are:

- Completing a Proxy Block
- Exporting the proxy
- Registering the class with the Lua state.

3.3.2.1 Minimal Class Proxy

The following shows the usage of the DSL to proxy a very simple class, Stub1, and to use this proxy in Lua.

```
struct Stub1 {};
```

3.3.2.1.1 Proxy Block

Firstly you create a proxy block. The block starts with a OOLUA_PROXY call to which you pass the name of the C++ class to be proxied and the block ends at the next OOLUA_PROXY_END. Soon we will see how to proxy other aspects of a class in this block.

```
OOLUA_PROXY(Stub1)
OOLUA_PROXY_END
```

3.3.2.1.2 Exporting

Secondly you export the member functions which are to be made available for the type in Lua. Exporting defines which member functions will be registered with Lua when the class type is registered. Even when there are no member functions to be exported you still need to inform OOLua about this. Calling an OOLUA_EXPORT∗ procedure in a header file is an error that will fail to compile.

See Also

OOLUA_EXPORT_FUNCTIONS
OOLUA_EXPORT_FUNCTIONS_CONST
OOLUA_EXPORT_NO_FUNCTIONS

As the simple class that is being proxied does not have any member functions, the code here uses the specific DSL procedure for this.

```
OOLUA_EXPORT_NO_FUNCTIONS(Stub1)
```

3.3.2.1.3 Registering

Lastly we register the type with a lua_State after which the type can be created and used in the virtual machine.

```
void setUp()
{
    m_lua = new OOLUA::Script;
    m_lua->register_class<Stub1>();
}

void new_luaCreatesInstance_noException()
{
    CPPUNIT_ASSERT_NO_THROW(m_lua->run_chunk("Stub1.new()"));
}
```

3.3.2.2 Tags

Tags provide a method to inform the library that the type :

- has relationship and/or mathematical operators
- is an abstract class
- doesn't have a default constructor or any public constructors
- has enumerations

For an exhaustive list of the possible tags see Tags.

OOLUA_TAGS(TagList)

Parameters

TagList	Comma separated list of Tags

Note

An OOLUA_TAGS list without any Tags entries is invalid.

3.3.2.3 Constructors

3.3.2.3.1 Default Constructor

The default constructor of a proxy type is a special member function, much like C++, and it will be implicitly defined for the type unless otherwise specified. When available for a type "foo" it can be called in Lua using the following syntax.

```
foo.new()
```

See Also

Abstract, No_default_constructor and No_public_constructors

3.3.2.3.2 Non-default Constructors

OOLUA_CTORS(ConstructorEntriesList)

Parameters

Constructor-EntriesList	List of OOLUA_CTOR

To enable the construction of an instance without using the default constructor, there must be a constructor block specified for the proxy type. The constructor block, OOLUA_CTORS, is where non-default constructor entries can be specified using an OOLUA_CTOR per entry.

Constructors are the only real type of overloading which is permitted by OOLua and there is an important point which should be noted. This being that OOLua will attempt to match the number and type of parameters on the stack with the amount and types specified for each OOLUA_CTOR entry. The order in which it will attempt the matching is the same order in which they were defined. When interacting with the Lua stack certain types can not be differentiated between, these include some integral types such as float, int, double etc and types which are of a proxy class type or derived from that type. OOLua implicitly converts between classes in a hierarchy even if a reference is required. This means for example that if there are constructors such as Foo::Foo(int) and Foo::Foo(float) it will depend on which was defined first in the OOLUA_CTORS block as to which will be invoked for a call such as Foo.new(1).

See Also

No_default_constructor

Note

An OOLUA_CTORS block without any OOLUA_CTOR entries is invalid.

```
OOLUA_PROXY(ParamConstructor)
    OOLUA_TAGS(
        No_default_constructor
    )
    OOLUA_CTORS(
        OOLUA_CTOR(bool) /*NOLINT(readability/casting)*/
        OOLUA_CTOR(int) /*NOLINT(readability/casting)*/
        OOLUA_CTOR(char const*)
        OOLUA_CTOR(int, bool)
        OOLUA_CTOR(Stub1 *) /*NOLINT(readability/casting)*/
        OOLUA_CTOR(Stub1 *, Stub2*)
        OOLUA_CTOR(Stub2)
        OOLUA_CTOR(Stub3*) /*NOLINT(readability/casting)*/
        OOLUA_CTOR(Stub3 const *)
        OOLUA_CTOR(OOLUA::Lua_func_ref)
        OOLUA_CTOR(OOLUA::Table)
    )
OOLUA_PROXY_END
```

3.3.2.4 Enumerations

Class enumerations, whether weak or scoped, are specified inside the OOLUA_ENUMS block. To register the enumeration values when the class type is, the Register_class_enums tag must be present in the tags block.

OOLUA_ENUMS(EnumEntriesList)

Parameters

EnumEntriesList	List which contains OOLUA_ENUM and/or OOLUA_SCOPED_ENUM entries.

Note

An OOLUA_ENUMS block without any OOLUA_ENUM or OOLUA_SCOPED_ENUM entries is invalid.

3.3.2.4.1 Weak Enumerations

OOLUA_ENUM(EnumName)

Parameters

EnumName	The class enumeration name

```
class Enums
{
public:
    enum COLOUR{GREEN = 0, INVALID};
    Enums()
        :m_enum(INVALID)
    {}
    Enums(COLOUR e)
        :m_enum(e)
    {}

    COLOUR m_enum;
    void set_enum(COLOUR e)
    {
        m_enum = e;
    }
    COLOUR get_enum()
    {
        return m_enum;
    }
};


OOLUA_PROXY(Enums)
    OOLUA_TAGS(
        Register_class_enums
    )
    OOLUA_CTORS(
        OOLUA_CTOR(Enums::COLOUR)
    )
    OOLUA_ENUMS(
        OOLUA_ENUM(GREEN)
        OOLUA_ENUM(INVALID)
    )
    OOLUA_MFUNC(set_enum)
    OOLUA_MFUNC(get_enum)
OOLUA_PROXY_END


OOLUA_EXPORT_FUNCTIONS_CONST(Enums)
OOLUA_EXPORT_FUNCTIONS(Enums
                        , set_enum
                        , get_enum)


    void constructWithEnum_passedValueGreen_functionReturnsGreen()
    {
        m_lua->register_class<Enums>();
        m_lua->run_chunk("foo = function() "
                            "local obj = Enums.new(Enums.GREEN) "
                            "return obj:get_enum() "
                         "end");
        Enums::COLOUR result(Enums::INVALID);
        m_lua->call("foo");
        OOLUA::pull(*m_lua, result);
        CPPUNIT_ASSERT_EQUAL(Enums::GREEN, result);
    }
```

3.3.2.4.2 Scoped Enumerations

OOLUA_SCOPED_ENUM(EnumName, Entry)

Parameters

Name	The class enumeration name which will be used to access it from Lua
Entry	The class enumeration scoped qualified name (minus the class type)

See Also

OOLUA_ENUM

```
struct Has_scoped_enum
{
    enum class scoped_enum{INVALID, VALID};
    Has_scoped_enum()
        :e(scoped_enum::VALID)
    {}
    Has_scoped_enum(scoped_enum input)
        :e(input)
    {}
    scoped_enum e;
    void param(scoped_enum /*e*/){}
    scoped_enum return_enum(){return scoped_enum::VALID;}
};


OOLUA_PROXY(Has_scoped_enum)
    OOLUA_TAGS(Register_class_enums)
    OOLUA_CTORS(
        OOLUA_CTOR(Has_scoped_enum::scoped_enum)
    )
    OOLUA_ENUMS(
        OOLUA_SCOPED_ENUM(INVALID, scoped_enum::INVALID)
        OOLUA_SCOPED_ENUM(VALID, scoped_enum::VALID)
    )
    OOLUA_MGET_MSET(e)
    OOLUA_MFUNC(param)
    OOLUA_MFUNC(return_enum)
OOLUA_PROXY_END


OOLUA_EXPORT_FUNCTIONS(Has_scoped_enum
                        , param
                        , return_enum
                        , set_e)
OOLUA_EXPORT_FUNCTIONS_CONST(Has_scoped_enum
                            , get_e)


    void publicMember_inCppSetMemberToInvalidInLuaSetToValid_resultEqualsValid()
    {
        Has_scoped_enum instance;
        instance.e = Has_scoped_enum::scoped_enum::INVALID;
        m_lua->register_class<Has_scoped_enum>();
        m_lua->run_chunk("return function(obj) obj:set_e(Has_scoped_enum.VALID) end");
        m_lua->call(-1, &instance);
        CPPUNIT_ASSERT_EQUAL(static_cast<int>(Has_scoped_enum::scoped_enum::VALID), static_cast<int>(
      instance.e));
    }
```

3.3.2.5 Exposing Member Functions

Minimalist

Generates a proxy function using the only the minimal amount of information which is generally the name of the thing being proxied and possibly a new name for the proxy. If a new name is supplied then the function will be made available to Lua using it and this name must be used when Exporting the function.

This part of the DSL attempts to automatically determine the parameter types and return type for the function in question. However, if the function is overloaded then the compiler will be unable to resolve the function, due to the ambiguity, and will produce a compile time error. To help the compiler resolve this ambiguity, the user should specify more information using the corresponding, yet longer named, Expressive DSL entry.

The longer DSL name requires more information.

Note

No Traits can be expressed with this DSL group.

Expressive

Generates a function for which the user has expressed all the parameters and the return type for a function. These types may also have Traits applied to them which the Minimalist section of the DSL does not allow.

OOLUA_MFUNC(FunctionName, Optional)

Parameters

FunctionName	Name of the member function to be proxied
Optional	ProxyFunctionName. Defaults to FunctionName

See Also

cpp_params
OOLUA_MEM_FUNC
OOLUA_MEM_FUNC_RENAME

OOLUA_MFUNC_CONST(FunctionName, Optional)

Parameters

FunctionName	Name of the constant function to be proxied
Optional	ProxyFunctionName. Defaults to FunctionName

See Also

cpp_params
OOLUA_MEM_FUNC_CONST
OOLUA_MEM_FUNC_CONST_RENAME

```
//typedef the type of vector into the global namespace
//This is required as a vector has more than one template type
//and the commas in the template confuse a macro.
typedef std::vector<int> vector_int;

OOLUA_PROXY(vector_int)
    //C++11 adds an overload
    //OOLUA_MFUNC(push_back)
    OOLUA_MEM_FUNC(void, push_back, class_::const_reference)
    OOLUA_MFUNC(pop_back)
    OOLUA_MFUNC_CONST(size)
OOLUA_PROXY_END


OOLUA_EXPORT_FUNCTIONS(vector_int, push_back, pop_back)
OOLUA_EXPORT_FUNCTIONS_CONST(vector_int, size)
```

3.3.2.6 Abstract Class

Generating an abstract proxy requires that you specify the Abstract tag in the OOLUA_TAGS block. When OOLua encounters the Abstract tag it will not look for any constructors for the type and the type will not be constructable from Lua. Specifying an OOLUA_CTORS block will have no effect and such a block will be ignored.

```
class Abstract1
{
public:
    virtual ~Abstract1(){}
    virtual void func1()=0;
    virtual void virtualVoidParam3Int(int, int, int) = 0;
};


OOLUA_PROXY(Abstract1)
    OOLUA_TAGS(Abstract)
    OOLUA_MFUNC(virtualVoidParam3Int)
    OOLUA_MFUNC(func1)
OOLUA_PROXY_END


OOLUA_EXPORT_FUNCTIONS(Abstract1, func1, virtualVoidParam3Int)
OOLUA_EXPORT_FUNCTIONS_CONST(Abstract1)
```

3.3.2.7 Base Classes

Using OOLUA_PROXY's optional parameter(s) enables the specifying of base class(es) for a proxy. OOLUA_PROXY(ClassName, Optional)

Parameters

ClassName	Class to be proxied
Optional	Comma seperated list of real base classes

Precondition

Each class specified in Optional must be a real base class of ClassName

```
class Derived1Abstract1 : public Abstract1
{
public:
    virtual ~Derived1Abstract1(){}
    MOCK_METHOD0(func1, void());
    MOCK_METHOD3(virtualVoidParam3Int, void(int, int, int));
};
```

The following snippets do not proxy or expose any of the functions from the base class as they are automatically made available for the derived class. This is true for all derived proxies which have a base proxy.

```
OOLUA_PROXY(Derived1Abstract1, Abstract1)
OOLUA_PROXY_END


OOLUA_EXPORT_FUNCTIONS(Derived1Abstract1)
OOLUA_EXPORT_FUNCTIONS_CONST(Derived1Abstract1)
```

3.3.2.8 Operators

Operator Tags inform OOLua that a class exposes one or more of the operators supported:

- Less_op
- Equal_op
- Not_equal_op
- Less_equal_op
- Div_op
- Mul_op
- Sub_op
- Add_op

```
class Class_ops
{
public:
    Class_ops(int const & i):m_i(i){}
    Class_ops():m_i(0){}
    Class_ops(Class_ops const& rhs)
        :m_i(rhs.m_i)
    {}

    int const& geti()const
    {
        return m_i;
    }
    bool operator == (Class_ops const& rhs)const
    {
        return m_i == rhs.m_i;
    }
    bool operator < (Class_ops const& rhs)const
    {
        return m_i < rhs.m_i;
    }
    bool operator <= (Class_ops const& rhs)const
    {
        return m_i <= rhs.m_i;
    }
    Class_ops operator + (Class_ops const& rhs)const
    {
        return Class_ops( m_i + rhs.m_i );
    }
    Class_ops operator * (Class_ops const& rhs)const
    {
        return Class_ops(m_i * rhs.m_i);
    }
    Class_ops operator - (Class_ops const& rhs)const
    {
        return Class_ops(m_i - rhs.m_i);
    }
    Class_ops operator / (Class_ops const& rhs)const
    {
        return Class_ops(m_i / rhs.m_i);
    }
private:
    int m_i;
};


OOLUA_PROXY(Class_ops)
    OOLUA_TAGS(
        Equal_op
        , Less_op
        , Less_equal_op
        , Add_op
        , Sub_op
        , Mul_op
        , Div_op
    )
    OOLUA_MFUNC_CONST(geti)
OOLUA_PROXY_END


OOLUA_EXPORT_FUNCTIONS(Class_ops)
OOLUA_EXPORT_FUNCTIONS_CONST(Class_ops, geti)
```

3.3.2.9 Public Members

Getting or setting a public member is achieved by a function which completes the operation. These functions must be exported like all other proxy functions, so that they are available to a Lua script.

OOLUA_MGET(PublicName, Optional)

Parameters

PublicName	Name of the public variable to be proxied.
Optional	GetterName. Defaults to get_PublicName

Note

A generated getter for a pointer, or shared pointer, with a proxied pointee type, has an implicit OOLUA::maybe_null trait applied.

OOLUA_MSET(PublicName, Optional)

Parameters

PublicName	Name of the public variable to be proxied.
Optional	SetterName. Defaults to set_PublicName

OOLUA_MGET_MSET(PublicName, Optional1, Optional2)

Parameters

PublicName	Name of the public variable to be proxied.
Optional1	GetterName. Defaults to get_PublicName
Optional2	SetterName. Defaults to set_PublicName

See Also

OOLUA_MGET and OOLUA_MSET

Note

If one optional parameter is supplied then both must be given and they must use different names.

```
class Public_variables
{
public:
    Public_variables();
    ~Public_variables();
    int an_int;
    int m_int;
    int* int_ptr;
    Stub1* dummy_instance;
    Stub1 dummy_instance_none_ptr;
    Stub1& dummy_ref;
    Enums enum_instance_none_ptr;

    static const int set_value = 1;
    static const int initial_value = 0;


    Public_variables(Public_variables const&);
    Public_variables& operator = (Public_variables const&);
};


OOLUA_PROXY(Public_variables)
    OOLUA_MGET_MSET(an_int)
    OOLUA_MGET_MSET(int_ptr, get_int_ptr, set_int_ptr)
    OOLUA_MGET_MSET(dummy_instance)
    OOLUA_MGET_MSET(dummy_ref)
    OOLUA_MGET_MSET(dummy_instance_none_ptr)
    OOLUA_MGET(m_int, get_int)
    OOLUA_MGET(m_int)
    OOLUA_MSET(m_int, set_int)
    OOLUA_MSET(m_int)
    OOLUA_MSET(enum_instance_none_ptr)
OOLUA_PROXY_END


OOLUA_EXPORT_FUNCTIONS(Public_variables
                                , set_an_int
                                , set_int_ptr
                                , set_dummy_instance
                                , set_dummy_ref
                                , set_m_int
                                , set_int
                                , set_dummy_instance_none_ptr
                                , set_enum_instance_none_ptr)

OOLUA_EXPORT_FUNCTIONS_CONST(Public_variables
                            , get_an_int
                            , get_int_ptr
                            , get_dummy_instance
                            , get_dummy_ref
                            , get_dummy_instance_none_ptr
                            , get_int
                            , get_m_int)
```

Public member access in Lua is via a member function

```
    void getAnInt_publicVariablesClassPassedToLua_returnsSetValue()
    {
        m_class_with_public_vars->an_int = Public_variables::set_value;
        m_lua->run_chunk("func = function(obj) return obj:get_an_int() end");
        m_lua->call("func", m_class_with_public_vars);
        int result;
        OOLUA::pull(*m_lua, result);
        CPPUNIT_ASSERT_EQUAL(Public_variables::set_value, result);
    }
```

3.3.2.10 Static Functions

OOLUA_SFUNC(FunctionName, Optional)

Parameters

FunctionName	Name of the static function to be proxied
Optional	ProxyFunctionName. Defaults to FunctionName

Note

This function will not be exported and needs to be registered with OOLua see OOLUA::register_class_static

See Also

cpp_params

```
class ClassHasStaticFunction
{
public:
    static void static_function(){}
    static void static_function(int /*DontCare*/){}
    static int returns_input(int t){return t;}
};


OOLUA_PROXY(ClassHasStaticFunction)
    OOLUA_TAGS(No_public_constructors)
    OOLUA_SFUNC(returns_input)
OOLUA_PROXY_END


OOLUA_EXPORT_NO_FUNCTIONS(ClassHasStaticFunction)
```

When registering a static function that was exposed with OOLUA_SFUNC, the second parameter to the OOLUA-::register_class_static function is the address of the proxy function. The parameter therefore needs to be a fully qualified static function for the specialised Proxy_class.

```
m_lua->register_class_static<ClassHasStaticFunction>("returns_input",
                                                     &
      OOLUA::Proxy_class<ClassHasStaticFunction>::returns_input
      );

m_lua->run_chunk("foo = function(obj, input) "
                   "return obj.returns_input(input) "
               "end ");
```

3.3.3 C Functions

3.3.3.1 Minimalist

We have already seen the Minimalist version in the Hello Moon example.

Deduce and generate a proxy for a C function.

OOLUA_CFUNC(FunctionName, ProxyFunctionName)

Parameters

FunctionName	Name of the C function to be proxied
ProxyFunction-Name	Name of the function to generate which will proxy FunctionName

See Also

cpp_params
OOLUA_C_FUNCTION

```
void say(char const* input)
{
    printf("%s from a standalone function\n", input);
}
```

```
OOLUA_CFUNC(say, l_say)


    void hello_minimalist_function()
    {
        using namespace OOLUA; //NOLINT(build/namespaces)
        Script vm;
        set_global(vm, "say", l_say);
        run_chunk(vm, "say('Hello Lua')");
    }
```

3.3.3.2 Expressive

Generates a block which will call the C function FunctionName.

OOLUA_C_FUNCTION(FunctionReturnType,FunctionName, Optional)

Parameters

FunctionReturn-Type	
FunctionName	
Optional	Comma separated list of function parameter types

See Also

cpp_params

Precondition

The function in which this macro is contained must declare a lua_State pointer which can be identified by the name "vm"

```
extern void foo(int);
int l_foo(lua_State* vm)
{
    OOLUA_C_FUNCTION(void,foo,int)
}
```

Note

This macro should ideally be used as the last operation of a function body as control will return to the caller. Notice there is no return statement in l_foo

In the following example we have a C function which is overloaded, we can use the Expressive DSL here in which we supply the return and parameter types. The function will then be resolved to the correct overload.

```
void expressive_say(char const* input)
{
    printf("%s from a expressive function\n", input);
}
void expressive_say(int input)
{
    printf("Huh %d\n", input);
    CPPUNIT_ASSERT(0);
}


int expressive_lsay(lua_State* vm)
{
    OOLUA_C_FUNCTION(void, expressive_say, char const*)
}


    void hello_expressive_function()
    {
        using namespace OOLUA; //NOLINT(build/namespaces)
        Script vm;
        set_global(vm, "say", expressive_lsay);
        vm.run_chunk("say('Hello Lua')");
    }
```

3.3.3.3 Overloaded Minimalist

You may have noticed that we did not apply any Traits for the Expressive C version, so maybe it would be nice if we could do it another way; well that all depends on what you consider nice! The function can not be resolved unless we give the compiler more information, but in this case it does not mean we have to use the Expressive DSL. We can instead cast the function pointer, note that a stand alone function name is a function pointer, to the wanted type and therefore resolve to the correct function overload whilst still using the Minimalist DSL

```
void expressive_say(char const* input)
{
    printf("%s from a expressive function\n", input);
}
void expressive_say(int input)
{
    printf("Huh %d\n", input);
    CPPUNIT_ASSERT(0);
}

OOLUA_CFUNC( (( void(*)(char const*))expressive_say), cast_expressive_say)


    void hello_cast_minimalist_function()
    {
        using namespace OOLUA; //NOLINT(build/namespaces)
        Script vm;
        set_global(vm, "say", cast_expressive_say);
        vm.run_chunk("say('Hello Lua, we are a cast function not')");
    }
```

3.3.4 Traits

Provides direction and/or ownership information.

The general naming conventions for traits are :

- Parameter Traits : end in "_p"
- Function Return Traits : end in "_return" or "_null"
- Stack Traits : end in "_ptr".

3.3.4.1 Parameter Traits

DSL Traits for function parameter types.

Traits which allow control of ownership include in their name either "lua" or "cpp"; directional traits contain "in", "out" or a combination.

3.3.4.1.1 in_p

The calling Lua procedure supplies the parameter to the proxied function. No change of ownership occurs.

Note

This is the default trait used for function parameters when no trait is supplied.

Member Function:

```
    virtual void refPtrConst(ParamType const* & instance) = 0;
```

Proxy Function:

```
    OOLUA_MFUNC(refPtrConst)
```

Usage:

```
void inTraitConst_refPtrConst_calledOnceWithCorrectValue()
{
    InHelper helper(m_lua);
    EXPECT_CALL(helper.mock, refPtrConst(::testing::Eq(helper.inputParam_ptrConst))).Times(1);
    helper.run_method();
    m_lua->call(1, helper.object, "refPtrConst", helper.inputParam_ptrConst);
}
```

3.3.4.1.2 out_p

The calling Lua procedure does not pass the parameter to the proxied function, instead one is created using the default constructor and passed to the proxied function. The result after the proxied call with be returned to the calling procedure. If this is a type which has a proxy then it will cause a heap allocation of the type, which Lua will own.

Member Function:

```
virtual void refPtr(ParamType*& instance) = 0;
```

Proxy Function:

```
OOLUA_MEM_FUNC_RENAME(outTraitRefPtr, void, refPtr, out_p<HasIntMember*&>)
```

Usage:

```
void OutTraitRefPtr_luaPassesNoParam_topOfStackIsOwnedByLua()
{
    ::testing::NiceMock<OutParamUserDataMock> stub;
    m_lua->run_chunk("return function(obj) return obj:outTraitRefPtr() end");
    m_lua->call(1, static_cast<OutParamUserData*>(&stub));
    OOLUA::INTERNAL::Lua_ud * ud = static_cast<OOLUA::INTERNAL::Lua_ud *>(lua_touserdata(*m_lua, -1));
    CPPUNIT_ASSERT_EQUAL(true, OOLUA::INTERNAL::userdata_is_to_be_gced(ud));
}
```

3.3.4.1.3 in_out_p

The calling Lua procedure supplies the parameter to the proxied function, the value of the parameter after the proxied call will be passed back to the calling procedure as a return value. No change of ownership occurs.

Member Function:

```
virtual void ref(ParamType& instance)=0;
```

Proxy Function:

```
OOLUA_MEM_FUNC(void, ref, in_out_p<int&>)
```

Usage:

```
void inOutTraitRef_luaPassesIntCppAssignsNewValue_returnIsNewlyAssignedValue()
{
    InOutParamHelper helper(m_lua);
    EXPECT_CALL(helper.mock, ref(::testing::_)).Times(1).WillOnce(::testing::SetArgReferee<0>(helper.
  expected));
    m_lua->run_chunk("return function(object) return object:ref(1) end");
    m_lua->call(1, helper.object);
    assert_top_of_stack_is_expected_value(helper.expected);
}
```

3.3.4.1.4 lua_out_p

Lua code does not pass an instance to the C++ function, yet the pushed back value after the function call will be owned by Lua. This is meaningful only if called with a type which has a proxy and it is by reference, otherwise undefined.

Member Function:

```
virtual void refPtr(ParamType*& instance) = 0;
```

Proxy Function:

```
OOLUA_MEM_FUNC_RENAME(lua_takes_ownership_of_ref_2_ptr
                    , void, refPtr, lua_out_p<Stub1*&>)
```

Usage:

```
    m_lua->register_class<OwnershipParamUserData>();
    m_lua->run_chunk("return function(object) return object:lua_takes_ownership_of_ref_2_ptr()
   end");
    m_lua->call(1, object);
    //there is now a proxy type on top of the stack which Lua owns
```

3.3.4.1.5 cpp_in_p

Parameter supplied via Lua changes ownership to C++.

Member Function:

```
virtual void ptr(ParamType* instance) = 0;
```

Proxy Function:

```
OOLUA_MEM_FUNC_RENAME(cpp_takes_ownership_of_ptr_param
                    , void, ptr, cpp_in_p<Stub1*>)
```

Usage:

```
void cppInP_ptr2UserDataType_passingPtrThatLuaOwns_topOfStackGcIsFalse()
{
    bool result = returnGarbageCollectValueAfterCppTakingOwnership(
                     "cpp_takes_ownership_of_ptr_param");
    CPPUNIT_ASSERT_EQUAL(false, result);
}
```

3.3.4.1.6 light_p

The calling Lua procedure supplies a LUA_TLIGHTUSERDATA which will be cast to the requested T type. If T is not the correct type for the light userdata then the casting is undefined. A light userdata is never owned by Lua

Member Function:

```
void value(void* void_ptr);
```

Proxy Function:

```
OOLUA_MEM_FUNC(void, value, light_p<void*>)
```

Usage:

```
void functionParam_functionWhichTakesVoidPointer_functionIsCalledWithTheCorrectValue()
{
    LightParamUserDataMock mock;
    LightParamUserData* object = &mock;
    m_lua->register_class<LightParamUserData>();
    int i(0);
    void* input_ud = &i;
    EXPECT_CALL(mock, value(::testing::Eq(input_ud))).Times(1);
    m_lua->run_chunk("return function(object,param) return object:value(param) end");
    m_lua->call(1, object, input_ud);
}
```

or

Member Function:

```
void ptr(InvalidStub* data);
```

Proxy Function:

```
    OOLUA_MEM_FUNC(void, ptr, light_p<InvalidStub*>)
```

Usage:

```
    void functionParam_functionWhichTakesNoneVoidPointer_functionIsCalledWithTheCorrectValue()
    {
        LightNoneVoidParamUserDataMock mock;
        LightNoneVoidParamUserData* object = &mock;
        m_lua->register_class<LightNoneVoidParamUserData>();
        InvalidStub lightud;
        void* lightud_ptr = &lightud;
        EXPECT_CALL(mock, ptr(::testing::Eq(lightud_ptr))).Times(1);
        m_lua->run_chunk("return function(object,param) return object:ptr(param) end");
        m_lua->call(1, object, lightud_ptr);
    }
```

3.3.4.1.7 calling_lua_state

This is different from all other traits as it does not take a type, yet is a type. It informs OOLua that the calling state is a parameter for a function

Member Function:

```
    virtual void value(ParamType instance) = 0;
```

Proxy Function:

```
    OOLUA_MEM_FUNC(void, value, calling_lua_state)
```

Usage:

```
    void callingLuaState_luaPassesNoParameterYetFunctionWantsALuaInstance_calledOnceWithCorrectInstance()
    {
        LuaStateParamMock mock;
        lua_State* vm = *m_lua;
        EXPECT_CALL(mock, value(::testing::Eq(vm))).Times(1);

        m_lua->register_class<LuaStateParam>();
        m_lua->run_chunk("return function(object) object:value() end");
        m_lua->call(1, static_cast<LuaStateParam*>(&mock));
    }
```

3.3.4.2 Function Return Traits

DSL traits for function return types.

Some of the these traits allow for NULL pointers to be returned from functions, which was something commonly requested for the library. When such a trait is used and the runtime value is NULL, Lua's value of nil will be pushed to the stack.

3.3.4.2.1 lua_return

The type returned from the function is a heap allocated instance whose ownership will be controlled by Lua. This is only valid for function return types.

Member Function:

```
    virtual ReturnType* ptr() = 0;
```

Proxy Function:

```
OOLUA_MEM_FUNC(lua_return<Stub1*>, ptr)
```

Usage:

```
void luaReturnTrait_callsMethodPtr_returnValueIsToBeGarbageCollected()
{
    ReturnTraitHelper helper(m_lua);
    EXPECT_CALL(helper.mock, ptr()).Times(1).WillOnce(::testing::Return(&helper.return_stub));
    helper.call_object_method("ptr");
    assert_that_tops_gc_flag_is(true);
    set_tops_gc_flag_to(false);
}
```

3.3.4.2.2 maybe_null

The type returned from the function is a pointer instance whose runtime value maybe NULL. If it is NULL then lua_pushnil will be called else the pointer will be pushed as normal. No change of ownership will occur for the type. This is only valid for function return types.

Note

> To be consistent in naming this should really be called maybe_null_return, however I feel this would be too long a name for the trait so "return" has been dropped.

Member Function:

```
virtual ReturnType * const constPtr() = 0;
```

Proxy Function:

```
OOLUA_MEM_FUNC(maybe_null<Stub1*>, ptr)
```

Usage:

```
void maybeNullTrait_callsMethodConstPtrWhichReturnsNull_stackTopIsNil()
{
    MaybeNullTraitHelper helper(m_lua);
    EXPECT_CALL(helper.mock, constPtr()).Times(1).WillOnce(::testing::Return(static_cast<Stub1 *const>(
  NULL)));
    helper.call_object_method("constPtr");
    CPPUNIT_ASSERT_EQUAL(LUA_TNIL, lua_type(*m_lua, -1));
}
```

3.3.4.2.3 maybe_null and lua_return

The maybe_null and lua_return traits can be combined for a function return type. If the instance is non NULL then this combination provides the behaviour of the lua_return trait. On the other hand, when the instance is NULL it will provide the behaviour of the maybe_null trait.

Member Function:

```
virtual ReturnType* ptr() = 0;
```

Proxy Function:

```
OOLUA_MEM_FUNC(maybe_null<lua_return<Stub1*> >, ptr)
```

Usage:

```
void luaMaybeNullTrait_callsMethodPtrWhichReturnsValidPtr_stackTopGcValueIsTrue()
{
    LuaMaybeNullTraitHelper helper(m_lua);
    EXPECT_CALL(helper.mock, ptr()).Times(1).WillOnce(::testing::Return(&helper.return_stub));
    helper.call_object_method("ptr");
    assert_that_tops_gc_flag_is(true);
    set_tops_gc_flag_to(false);
}
```

3.3.4.2.4 light_return

The type returned from the function is either a void pointer or a pointer to another type. When the function returns, it will push a LUA_TLIGHTUSERDATA to the stack even when the pointer is NULL; therefore a NULL pointer using this traits is never converted to a Lua nil value. A light userdata is also never owned by Lua and OOLua does not store any type information for the it; light_return is a black box which when used incorrectly will invoke undefined behaviour.

This is only valid for function return types.

Void pointer:

```
OOLUA_MEM_FUNC(light_return<void*>, value)
```

Non void pointer:

```
OOLUA_MEM_FUNC(light_return<InvalidStub*>, ptr)
```

3.3.4.3 Stack Traits

Public API traits which control a change of ownership.

Valid to usage for the Public API which interact with the Lua stack.

3.3.4.3.1 cpp_acquire_ptr

Informs the library that C++ will take control of the pointer being used and call delete on it when appropriate. This is only valid for public API functions which OOLUA::pull from the stack.

```
    m_lua->run_chunk("return Stub1.new()");
    OOLUA::cpp_acquire_ptr<Stub1*> res;
    OOLUA::pull(*m_lua, res);
    CPPUNIT_ASSERT_EQUAL(true, res.m_ptr != 0);
    delete res.m_ptr;
```

3.3.4.3.2 lua_acquire_ptr

Informs the library that Lua will take control of the pointer being used and call delete on it when appropriate. This is only valid for public API functions which OOLUA::push to the stack.

```
void callFunction_passingPointerUsingLuaAcquirePtr_topOfStackGcIsTrue()
{
    Stub1 stub;
    m_lua->run_chunk("foo = function(param) return param end");
    m_lua->call("foo", OOLUA::lua_acquire_ptr<Stub1*>(&stub));
    OOLUA::INTERNAL::Lua_ud * ud = get_ud_helper();
    bool gc_value = OOLUA::INTERNAL::userdata_is_to_be_gced(ud);
    OOLUA::INTERNAL::userdata_gc_value(ud, false);
    CPPUNIT_ASSERT_EQUAL(true, gc_value);
}
```

Note

Here we use the public API function OOLUA::Script::call which uses OOLUA::push

3.3.4.4 Return Order

Lua supports multiple return values for functions (return = [explist]). The order of returns in the stack is shown in the following example, simply the first will be pushed to the top of the stack, then the second to the top. This continues until all returns have been pushed on to the stack and the final return is located at the top.

```
void luaReturnOrder_luaFunctionWhichReturnsMultipleValuesToCpp_orderFromTopOfStackIsInput2Input1()
{
    m_lua->run_chunk("return function(input1, input2) return input1, input2 end ");
```

```
        int input1(1);
        int input2(2);

        m_lua->call(1, input1, input2);
        /*
          ========
         | input2 | <-- stack top
          ========
         | input1 |
          ========
         |   ...  |
         */
        int topOfStack, nextSlot;
        OOLUA::pull(*m_lua, topOfStack);
        OOLUA::pull(*m_lua, nextSlot);

        CPPUNIT_ASSERT_EQUAL(input2, topOfStack);
        CPPUNIT_ASSERT_EQUAL(input1, nextSlot);
    }
```

C++ in a way also supports multiple returns via references. Here we have a C++ member function which returns an int, the function also assigns a new value to the parameter which is taken by reference.

```
struct ReturnOrder
{
    enum {returnValue=-1, paramValue};
    int foo(int& bar)
    {
        bar = paramValue;
        return returnValue;
    }
};
```

In effect this function has two return values so one way we could proxy the function and detail that information would be using the Expressive DSL macro OOLUA_MEM_FUNC and applying an in_out_p trait to the parameter.

```
OOLUA_PROXY(ReturnOrder)
    OOLUA_MEM_FUNC(int, foo, in_out_p<int&>)
OOLUA_PROXY_END
```

After calling this function there will be two returned values; the return of the C++ function and the value of the parameter after the call. The top of stack will contain the furthest right handside parameter which had an out trait, which in this case there was only one, below this will be proceeding parameters which had out traits and then the return value in that order.

```
    void ordering_functionWhichReturnsValueAndTwoInOutParams_orderFromTopOfStackIsParam2Param1Return()
    {
        int input1(OutParamsTest::Dummy);
        int input2(OutParamsTest::Dummy);
        run_chunk_function_push_two_ints("return_int_and_2_int_refs", input1, input2, true);
        ::testing::NiceMock<MockOutParamsTest> stub;
        m_lua->call("func", static_cast<OutParamsTest*>(&stub));

        int r1, r2, r3;
        OOLUA::pull(*m_lua, r1);//top of stack
        OOLUA::pull(*m_lua, r2);
        OOLUA::pull(*m_lua, r3);
        CPPUNIT_ASSERT_EQUAL(static_cast<int>(OutParamsTest::Param2), r1);
        CPPUNIT_ASSERT_EQUAL(static_cast<int>(OutParamsTest::Param1), r2);
        CPPUNIT_ASSERT_EQUAL(static_cast<int>(OutParamsTest::Return), r3);
    }
```

Are you a bottom up kind of person?

The return value is on the bottom of the stack (Lua stack index 1) with parameter one at index 2.

Chapter 4

Library Tests

OOLua is a test driven library which uses two cross platform external libraries for test verification, CppUnit 1.12.1 [1] is used for state based verification and GoogleMock 1.6 [2] for behaviour verification. For anybody who is not fimilar with these libraries and would like to know more then I would recommed an IBM article [3] for CppUnit whilst for GoogleMock a recorded presentation by the author [4] additionaly the library cheat sheet [5].

4.1 Directory Layout

Library test code is situated in a directory named unit_tests in the root of the repository [6] or the root of a released source package [7]. This directory has three main sub directories into which the test code is seperated.

- cpp_classes Classes which will be proxied in tests.
- bind_classes The OOLua bindings for the cpp_classes.
- test_classes Test suites using CppUnit and GoogleMock.

4.2 Test Scripts

See Also

Test Unit scripts

[1] CppUnit home page `http://sourceforge.net/projects/cppunit/`

[2] GoogleMock home page `http://code.google.com/p/googlemock/`

[3] Open source C/C++ unit testing tools, Part 2: Get to know CppUnit `http://www.ibm.com/developerworks/aix/libra`
`_cppunit/index.html`

[4] C++ Mocks Made Easy - An Introduction to gMock `http://www.youtube.com/watch?v=sYpCyL-I47rM`

[5] Google C++ Mocking Framework Cheat Sheet `http://code.google.com/p/googlemock/wiki/-CheatSheet`

[6] Repository unit test directory `http://oolua.org/browse/unit_tests`

[7] Source package downloads `http://oolua.org/downloads.html`

Chapter 5

Change Log

5.1 2.0.1

- Updated detection to include mscv 14 (Josh Hayashida) Bitbucket issue #16

5.2 2.0.0

- Pretty much a new DSL which is not backwards compatible
- Calling static functions in Lua now requires the dot notation
- Calling new in Lua now requires the dot notation
- New Lua module which generates boilerplate OOLua C++ files, removes old console application
- Added HTML docs and improved inline documentation for DSL, makes online wiki invalid
- Added a new Lua module for comparisons and updated C++ code, now compares with LuaBind,LuaBridge,S-LB3 and SWIG
- Renamed push2lua and pull2cpp to OOLUA::push and OOLUA::pull
- Added OOLua version macros OOLUA_VERSION_MAJ OOLUA_VERSION_MIN and OOLUA_VERSION_-PATCH
- Base checking no longer touches the Lua stack
- C string traits no longer use a std::string temporary
- Script helper class now has OOLUA::Script::push and OOLUA::Script::pull methods
- Bug fix. If an abstract class had a base class which was not abstract, then it was possible to call new on the type.
- Renamed Table::set_value to OOLUA::Table::set
- Renamed Table::remove_value to OOLUA::Table::remove
- New Lua simplified class format, which improves self call performance
- Extra parameters to bound functions are now ignored. Does not include constructors
- Renamed Script::get_ptr to OOLUA::Script::state for consistency
- Added a base class exception type OOLUA::Exception
- Added OOLUA::lua_return which is a specific trait for return types which will be owned by Lua.

- Added OOLUA::maybe_null which allows C functions and member functions to return NULL
- Added OOLUA::lua_maybe_null which allows C functions and member functions to return a runtime value of NULL, if it is not NULL then the instance will be owned by Lua
- Changed OOLUA_C_FUNCTION, it now requires a lua_State pointer instance identified as "vm" instead of 'l'
- Added OOLUA::light_p This pulls a light userdata from the stack and casts to the requested type
- Added OOLUA::light_return This is a function return type which pushes a light userdata onto the stack
- Removed ability for OOLUA::lua_acquire_ptr to be used on function returns, use OOLUA::lua_return instead
- Removed ability for OOLUA::cpp_acquire_ptr to be used for function parameters, use OOLUA::cpp_in_p instead
- Modified OOLUA_MGET, OOLUA_MSET and OOLUA_MGET_MSET to use optional parameters.
- Added oolua_dsl.h and oolua_dsl_export.h which reduces the include graph when using the DSL
- Added oolua_string.h/.c to make it easier to enable other string types as an integral type. OOLUA::STRING
- Bug fix. Prevent exceptions escaping from stand alone functions.
- Bug fix. Incorrect function dispatcher being set on a cached base constant method.
- Removed OOLUA::register_class_and_bases, OOLUA::register_class now does this.
- Added OOLUA::idxs_equal to compare stack indices, may take metamethods into consideration, compatible with Lua 5.1 and 5.2
- Added assignment operator for OOLUA::Lua_ref
- Added equality operator for OOLUA::Lua_ref
- Added assignment operator for OOLUA::Table
- Bug Pointer to first member of a class, without an offset from the class instance, was being incorrectly handled (Juan Batovi) Bitbucket issue #2
- Added OOLUA_USE_SHARED_PTR Enables support for a shared pointer type
- Added OOLUA_SHARED_HEADER Specifies the header for the shared pointer type
- Added OOLUA_SHARED_TYPE The shared pointer type which is supported
- Added OOLUA_SHARED_CONST_CAST Template function to cast away shared type constness
- Added OOLUA_NEW_POINTER_DEFAULT_IS_SHARED_TYPE Defines the default type to use when a new instance is created
- Added OOLUA::Shared When the default is to use raw pointers then this tag overrides it for a type
- Added OOLUA::No_shared When the default is to use a shared pointer this tag overrides it for a type
- Added OOLUA::shared_return trait which creates a shared pointer from raw
- Bug fix. Every OuterClass<InnerClass> is treated as shared pointer (Renan Inácio) Bitbucket issue #1.
- Added support for C++11 scoped enums OOLUA_SCOPED_ENUM
- Added ability for public members, which are proxies or shared_ptrs, to push nil when the value is NULL.
- Added Lua function new_table which takes array and hash size hints.
- Added a Lua module that amalgamates the library files.
- Bug fix. Incorrect handling of an error string which contained an embedded NULL. (Mauricio)
- Silenced g++ warnings about local typedefs which are unused i.e. static asserts
- Bug fix. Incorrectly attempting to handle proxy value types which are public members (Oscar Zhao) Bitbucket issue #8
- Fix - Typo in include guard of oolua_dsl_export.h (Chris Schade) Bitbucket issues #15

5.3 1.4.0

- Added OOLUA_DEDUCE_FUNC(_CONST) for when there is no ambiguity for a function
- Added OOLUA_TYPEDEFS_END which is an alias for OOLUA_END_TYPES to match the naming of other macros
- Type comparison now uses the address of a template typed function
- Removed OOLUA_SAFE_ID_COMPARE
- Added config option OOLUA_CHECK_EVERY_USERDATA_IS_CREATED_BY_OOLUA
- Added config option OOLUA_USERDATA_OPTIMISATION
- Moved base checking function from the metatable it is now store in Lua_ud
- Added new trait OOLUA::calling_lua_state which passes allows passing the calling Lua state as a parameter
- Added friendlier registering of class enums
- Added function return traits for a reference to constant std::string
- Bug fix Issue 28: Proxy checker typedefs in the default scope instead of public. (Sakamoto)
- Bug fix Issue 29: Lua 5.2 calls __gc method with a table. (Ilia Pavlovets)
- Prevent invalid Lua stack indexes when pulling a Lua_ref or Proxy_class. Indexes Zero (lua_gettop result) or -1 with an empty stack.
- Bug fix Issue 30: Table traverse function incorrectly assumes the stack is empty (Steve Nichols)
- Added oolua_ipairs and oolua_ipairs_end macros for iterating over arrays
- Added oolua_pairs and oolua_pairs_end macros for iterating over tables
- Removed the lua_State parameter from for_each_key_value function
- Added bool OOLUA::can_xmove(lua_State∗vm0,lua_State∗vm1)
- Lua_ref can safely be moved between related Lua states.
- Added OOLUA::load_chunk, OOLUA::run_chunk, OOLUA::run_file and OOLUA::load_file
- Bug fix Issue 25 : Enums being classed as a class type for member functions (Harley Laue)
- Added the ability to pass a stack index as the function to call with Lua_function
- Fixed on error Lua_function now resets the stack to the same as before entry.
- Added OOLua module
- Updated **VA_ARGS** macro for VS11

5.4 1.3.2

- Bug fix Issue 19 : Variadic macros which rename a function
- Added ability to typedef classes inside the OOLUA namespace see: `http://groups.google.-com/group/oolua-user/browse_thread/thread/688ddac870fb76d5`
- Bug fix Issue 22 : Remove return statements which generate warning with gcc (Tim Mensch)
- Refactored so that anything which is not meant to be called by a user, is now in the OOLUA::INTERNAL namespace,

- Added compile time constraints to traits
- OOLUA::cpp_acquire_ptr and OOLUA::lua_acquire_ptr - Type supplied to template is now the real type <foo∗> or <foo const∗>
- Bug fix : Converter was not taking the parameter by reference when it needed to.
- Removed the restriction of using classes only in the thread they were created under all conditions

5.5 1.3.1

- Work around for Visual Studio as reported by Tom on the mailing list

5.6 1.3.0

- Support for limited constructors
- Added a file generator to the generator solution for constructor parameters
- Added the types OOLUA::No_default_constructor,OOLUA::No_public_constructors and OOLUA::No_public_destructor to oolua_typedefs.h
- Added OOLUA_ONLY_DEFAULT_CONSTRUCTOR
- Broke ABI removing default constructor being forced
- Added OOLUA::table_set_value which does not retrieve the table from the registry yet uses a stack index
- Added convenience function OOLUA::new_table
- Added copy constructor to Lua_ref and Lua_table
- Added param traits for Lua_ref
- Added push member to lua_ref
- Enabled a constructor to take a Lua_func_ref
- Bug fix Issue 10 : fixed user type return on the stack (Tomm)
- Enabled a constructor to take a Lua_table
- Added method to pull a table reference from the stack
- Enabled a constructor to take a Lua_table_ref
- Added a conversion constructor to Lua_table from Lua_ref_table, introduced a friend hack!!
- Moved the Lua_table member function get_table to the private interface
- Refactored the pulling of a registry type (Lua_ref<T> and Lua_table)
- Added pulling a registry type when nil is on the stack, frees the registry ref and sets it to invalid
- Refactored Lua_table removing the lua_State instead using the reference's state member
- Added a default implementation of Proxy_class which creates a typedef that identifies it as a none proxy type
- Bug fix : Public members retrieved with get_?, now push by reference if the type has a proxy type and it is by value
- Visual Studio work around for when taking the address of a function
- Added quotation marks to TargetPath as a post build event in visual studio. Directories with a space caused a problem.

- Added check to make sure a user data type was created by OOLUA when pulling a class from the stack
- Bypassed checking the user data type when calling a member function on that instance
- Changed the internal registration key of the function which checks a class bases
- Added support for building and running unit tests with vs2010 and gmock 1.5
- Updated generator project to include C function wrappers
- Added C function wrappers
- Moved build scripts to "build_scripts" directory
- Added oolua_config.h
- Added config option OOLUA_RUNTIME_CHECKS_ENABLED
- Added config option OOLUA_STD_STRING_IS_INTEGRAL
- How errors are reported now depend on which language called the function and the settings in oolua_config.h
- OOLUA::push2lua now returns a boolean which is the result of the operation, if exceptions are enabled it throws on error
- OOLUA::Lua_function now adds a trackback (copied from Lua code) which is enabled with OOLUA_DEBU-G_CHECKS
- Operator functions now use the OOLUA::LUA_CALLED::pull2cpp functions which act differently to OOLUA-::pull2cpp on an error
- OOLUA::Lua_ref has two extra functions to be used via Lua code, lua_pull and lua_push
- Bug fix : OOLUA::Lua_table's safe_at now does the correct thing when exceptions are enabled and does not let an exception escape.
- Added definition of OOLUA::get_last_error even if store last error is not enabled, in this instance it is a no op.
- Exceptions now can pop the error of the Lua stack and Runtime_error can be initialsed with a string
- oolua_member_function.h 's proxy calling functions now wrap code in a try block if exceptions are enabled.
- Removed LUA_GLOBALSINDEX define from lua_includes when using Lua 5.2 instead lua_getglobal and lua_setglobal are used throughout
- Added support for std::string to have embedded nulls as suggested by Tomm on the mailing list
- Moved C++ classes used in tests to cpp_classes directory
- Moved OOLua proxy classes used in tests to bind_classes directory
- Moved all unit tests to the unit_tests directory
- Added string is integral unit test
- Table::pull_from_stack now returns a bool to indicate the result if called by C++ code and not using exceptions
- Added unit_test_config(root,name) to premake helper file
- Added support for **VA_ARGS** macros with one or more arguments
- Added support for **VA_ARGS** macros with zero arguments using compiler extensions
- Added helper function OOLUA::get_global
- Added helper function OOLUA::set_global
- Added helper function OOLUA::set_global_to_nil
- Bug fix : Calling a static function on a derived instance when the function was registered with a base class

5.7 1.2.2

- Converted Premake scripts to Premake4
- Optimised the checking of a type against a requested type
- Userdata name now changes when it's constant status in set_type_top_to_none_const
- Added Xcode support to Premake scripts
- Added xcode test unit bash build script.
- Build logs directed to there own directory
- Added new test project "tests_may_fail" for issue 7
- Updated bash build scripts to run the tests_may_fail aswell as unit.tests
- Added a readme.txt with details of library as many download locations are now available
- Bug fix Issue 8 : Passing a c style string to a member function bug as reported by (airbash)
- Bug fix Issue 8 : A corresponding bug of a member function which returns a c style string.
- Added define in lua_includes.h to support Lua 5.2 and 5.1.4 simultaneously
- Renamed platform test scripts
- Added build scripts to create a local install

5.8 1.2.1

- Was actually 1.2.0 yet due to a packaging error had to be incremented.

5.9 1.2.0

- Added fields to Lua_ud which are used for comparison removing the metatable raw_equals.
- Added name_size to proxy classes and updated the generation file to reflect changes.
- Changed headers that used old licence.
- Added a function to register a type and all it's bases.
- Added a couple of profile tests in the directory profile.

5.10 1.1.0

- Removed the dynamic allocation of Proxy classes to use stack versions.

5.11 1.0.0

- First public release

Chapter 6

Library Comparisons

6.1 Introduction

The intention of the comparison is to give both you and I some ball park costs and were orginally based on a Gem [1]; an excellent side effect from the libraries compared, other than SWIG, is that they have seen an optimisation improvement as a result.

Previous versions of these comparsions were perceived by some difficult to fully understand what a number meant in relation to others, without also understanding some of the differences between libraries; additionly there was a concern that the cost of the method look up should not be part of the comparison.

6.1.1 Userdata verification

Although the comparisons ran the same code when being timed, it was not simply a case of a one to one mapping between the different libraries.Most concerning to some was the fact that as a library feature LuaBind verified a userdata was created by itself whilst SWIG and originally OOLua did not perform such a check, thus OOLua and SWIG benefited whilst LuaBind was penalised.

Depending on your requirements SWIG, OOLua and LuaBind can all be compiled so that they do not perform these userdata checks, the potential problem this introduces can be shown with the following Lua 5.1 snippets:

```
--Calling a member function passing a none library userdata
local cached_func = obj.func
cached_func( newproxy() )
```

or

```
--Passing a none library userdata when one is needed
obj:func( newproxy() )
```

When an incorrect userdata is encountered which maybe from an external module or from a Lua script such as in the examples; then best case scenario is the library will detect it, yet in the process could cause undefined behaviour, and worst case maybe a segfault or your toaster runs off with the next door neighbour's.

To compile OOLua and LuaBind to use the same behaviour as SWIG

- OOLua: define OOLUA_CHECK_EVERY_USERDATA_IS_CREATED_BY_OOLUA 0
- LuaBind: define LUABIND_DISABLE_UD_CHECK and add the following macro guard to object_rep.cpp

```
    LUABIND_API object_rep* get_instance(lua_State* L, int index)
    {
        object_rep* result = static_cast<object_rep*>(lua_touserdata(L, index));

#ifndef LUABIND_DISABLE_UD_CHECK
        if (!result || !lua_getmetatable(L, index))
```

```
                return 0;

            lua_rawgeti(L, -1, 1);

            if (lua_tocfunction(L, -1) != &get_instance_value)
                result = 0;

            lua_pop(L, 2);
    #endif
            return result;
        }
```

For this reason the comparisons are performed for libraries with this feature enabled and disabled where possible, otherwise the category a library falls into by default.

Note

It is my belief that a determined party could possibly craft malicous code that will pass most library userdata checks, as essentially they all boil down to doing a check and if it passes then casting a void pointer to a type, some actually perform an undefined cast before any such check passes.[2]

6.1.2 Function caching

A Lua self call self:func() is functionally the same as self.func(self), it is also normal and recommeded usage in certain situations to cache values to locals. The comparison code is run in such a formentioned situation with tight loops, so if it were normal user code you would generally cache the member function as shown in the following example. Otherwise it would repeatively pay for the function look up when the object types are the same, whilst that is a valid concern my observed usage of C++ binding libraries is via an object call hence OOLua.

```
,mfunc_cached = function(object)
    local ave = 0
    local func = object.get
    for i = 0, N do
        local t0 = clock()
        for i=1,times do
            func(object)
        end
        local dt = clock()-t0
        if i~=0 then
        ave = ave + dt
        end
    end
    return (ave/N)/times
end
```

For this reason the comparisons are performed for libraries both with caching function and self calls.

- Comparison code
- Comparison results
- Comparison overview

[1] GPG6 Celes, W., Figueiredo, L.H. and Ierusalimschy, R., "Binding C/C++ Objects to Lua." Game Programming Gems 6, Charles River Media, 2006.

[2] Programming languages C++, ISO/IEC 14882:2003, "5.2.9 static_cast", American National Standards Institute, 2003

6.2 Comparison code

6.2.1 C++

The comparisons are performed using library bindings to the following C++ classes

```
class Set_get
{
public:
    Set_get():_i(0.0){}
    void set(double i)
    {
        _i = i;
    }
    double get()const
    {
        return _i;
    }
private:
    double _i;
};


class ProfileBase
{
public:
    ProfileBase():_i(0){}
    virtual ~ProfileBase(){}
    void increment_a_base(ProfileBase* base)
    {
        ++base->_i;
    }
    virtual void virtual_func()
    {
        ++_i;
    }
    virtual void pure_virtual_func() = 0;
private:
    int _i;
};

class ProfileAnotherBase
{
public:
    virtual ~ProfileAnotherBase(){}
};

class ProfileDerived : public ProfileBase
{
public:
    virtual ~ProfileDerived(){}
    virtual void pure_virtual_func()
    {
        ++_i;
    }
private:
    int _i;
};

class ProfileMultiBases : public ProfileDerived, public ProfileAnotherBase
{
public:
    void virtual_func()
    {
        ++_i;
    }
private:
    int _i;
};
```

6.2.2 Lua

The different types of function calls are ran using the following Lua module.

```
]]
local clock = os.clock
local N = 10
local times = 1000000

return
{
    vfunc_self = function(object)
        local ave = 0
        if not object.virtual_func then return -1 end
        for i = 0, N do
            local t0 = clock()
            for i=1,times do
                object:virtual_func()
            end
            local dt = clock()-t0
            if i~=0 then
                ave = ave + dt
            end
        end
        return (ave/N)/times
    end

    ,vfunc_cached = function(object)
        local ave = 0
        if not object.virtual_func then return -1 end
        for i = 0, N do
            local cached_vfunc = object.virtual_func
            local t0 = clock()
            for i=1,times do
                cached_vfunc(object)
            end
            local dt = clock()-t0
            if i~=0 then
             ave = ave + dt
            end
        end
        return (ave/N)/times
    end

    ,mfunc_self = function(object)
        local ave = 0
        for i = 0, N do
            local t0 = clock()
            for i=1,times do
                object:get()
            end
            local dt = clock()-t0
            if i~=0 then
             ave = ave + dt
            end
        end
        return (ave/N)/times
    end
    --
    ,mfunc_cached = function(object)
        local ave = 0
        local func = object.get
        for i = 0, N do
            local t0 = clock()
            for i=1,times do
                func(object)
            end
            local dt = clock()-t0
            if i~=0 then
            ave = ave + dt
            end
        end
        return (ave/N)/times
    end
    --
    ,increment_a_base_self = function(object,param)
        local ave = 0
        for i = 0, N do
            local t0 = clock()
            for i=1,times do
                object:increment_a_base(param)
            end
            local dt = clock()-t0
            if i~=0 then
             ave = ave + dt
            end
        end
        return (ave/N)/times
    end

    ,increment_a_base_cached = function(object,param)
        local ave = 0
        local func = object.increment_a_base
        for i = 0, N do
            local t0 = clock()
            for i=1,times do
                func(object,param)
            end
            local dt = clock()-t0
            if i~=0 then
            ave = ave + dt
            end
        end
        return (ave/N)/times
    end
}

--[[
```

6.3 Comparison results

Wed Aug 12 21:03:51 BST 2015 Intel(R) Core(TM) i3 CPU M 370 @ 2.40GHz

6.3.1 Lua 5.1.5 : Userdata checks

Library test	cached call	self call
LuaBind mfunc	1.007763e-07	1.673694e-07
LuaBind vfunc	1.067601e-07	1.76468e-07
LuaBind class param	1.85963e-07	2.301886e-07
LuaBridge mfunc	1.76075e-07	2.504913e-07
LuaBridge vfunc	unavailable	unavailable
LuaBridge class param	4.668377e-07	7.439228e-07
OOLua mfunc	6.41734e-08	9.685e-08
OOLua vfunc	6.39864e-08	8.78577e-08
OOLua class param	1.062235e-07	1.083249e-07

6.3.2 Lua 5.1.5 : No userdata checks

Library test	cached call	self call
LuaBind mfunc	7.60837e-08	1.451553e-07
LuaBind vfunc	8.61303e-08	1.550592e-07
LuaBind class param	1.399331e-07	1.882814e-07
OOLua mfunc	5.50472e-08	7.86604e-08
OOLua vfunc	5.48013e-08	8.65963e-08
OOLua class param	8.6675e-08	8.8671e-08
SWIG mfunc	6.39963e-08	2.467633e-07
SWIG vfunc	6.31579e-08	2.458278e-07
SWIG class param	1.135185e-07	2.726672e-07

6.3.3 Lua 5.2.4 : Userdata checks

Library test	cached call	self call
LuaBind mfunc	9.87227e-08	1.726097e-07
LuaBind vfunc	1.111242e-07	1.834055e-07
LuaBind class param	1.893031e-07	2.409434e-07
LuaBridge mfunc	2.153461e-07	2.693592e-07

LuaBridge vfunc	unavailable	unavailable
LuaBridge class param	5.786645e-07	8.499132e-07
OOLua mfunc	6.5286099999999e-08	8.65781e-08
OOLua vfunc	6.5739599999998e-08	8.7746900000001e-08
OOLua class param	1.060817e-07	1.075109e-07

6.3.4 Lua 5.2.4 : No userdata checks

Library test	cached call	self call
LuaBind mfunc	7.53741e-08	1.499324e-07
LuaBind vfunc	8.87473e-08	1.599992e-07
LuaBind class param	1.487226e-07	2.006113e-07
OOLua mfunc	5.7768e-08	7.99339e-08
OOLua vfunc	5.86034e-08	8.11962e-08
OOLua class param	8.9948499999999e-08	8.97089e-08
SLB3 mfunc	6.7081e-08	8.93789e-08
SLB3 vfunc	1.555159e-07	1.754534e-07
SLB3 class param	1.819193e-07	1.820886e-07
SWIG mfunc	6.50532e-08	2.608859e-07
SWIG vfunc	6.6952400000001e-08	2.587621e-07
SWIG class param	1.147327e-07	2.872808e-07

6.3.5 Lua 5.3.0 : Userdata checks

Library test	cached call	self call
LuaBind mfunc	1.012876e-07	1.807787e-07
LuaBind vfunc	1.117346e-07	1.824406e-07
LuaBind class param	1.934463e-07	2.430269e-07
LuaBridge mfunc	2.014037e-07	2.913269e-07
LuaBridge vfunc	unavailable	unavailable
LuaBridge class param	5.575555e-07	8.809483e-07
OOLua mfunc	6.92391e-08	8.83054e-08
OOLua vfunc	6.66639e-08	9.1454000000001e-08
OOLua class param	1.213471e-07	1.165102e-07

6.3.6 Lua 5.3.0 : No userdata checks

Library test	cached call	self call
LuaBind mfunc	7.57741e-08	1.47208e-07
LuaBind vfunc	8.70102e-08	1.576735e-07
LuaBind class param	1.50597e-07	2.0071e-07
OOLua mfunc	5.68803e-08	7.95171e-08

OOLua vfunc	5.87445e-08	7.98216e-08
OOLua class param	8.82771e-08	8.87009e-08
SLB3 mfunc	7.7821e-08	1.00388e-07
SLB3 vfunc	1.166812e-07	1.359227e-07
SLB3 class param	1.916863e-07	1.943615e-07
SWIG mfunc	6.44358e-08	2.997889e-07
SWIG vfunc	6.4826099999999e-08	2.976194e-07
SWIG class param	1.108507e-07	3.213004e-07

6.3.7 LuaJIT 2.0.3 : Userdata checks

Library test	cached call	self call
LuaBind mfunc	8.27365e-08	2.114417e-07
LuaBind vfunc	1.03834e-07	1.621875e-07
LuaBind class param	1.578896e-07	2.301369e-07
LuaBridge mfunc	1.617671e-07	2.337921e-07
LuaBridge vfunc	unavailable	unavailable
LuaBridge class param	3.994024e-07	6.173423e-07
OOLua mfunc	5.39881e-08	8.49012e-08
OOLua vfunc	5.01838e-08	7.65322e-08
OOLua class param	6.80857e-08	9.2842600000001e-08

6.3.8 LuaJIT 2.0.3 : No userdata checks

Library test	cached call	self call
OOLua mfunc	4.18646e-08	6.8805e-08
OOLua vfunc	3.93808e-08	6.6519e-08
OOLua class param	4.78118e-08	7.08243e-08
SWIG mfunc	4.39282e-08	2.067105e-07
SWIG vfunc	4.30934e-08	2.047546e-07
SWIG class param	6.90115e-08	2.320766e-07

6.4 Comparison overview

6.4.1 Userdata checks

Lua imp		mfunc			vfunc			param	
		cached	self		cached	self		cached	self
Lua 5.1.5		OOLua 6.-41734e-08	OOLua 9.685e-08		OOLua 6.-39864e-08	OOLua 8.-78577e-08		OOLua 1.-062235e-07	OOLua 1.-083249e-07

Lua 5.2.4		OOLua 6.-528609999[illegible] 08	OOLua 8.-[illegible]e- 08		OOLua 6.-57395999[illegible]00000001e- 08	OOLua 8.-[illegible] 08		OOLua 1.-060817e-07	OOLua 1.-075109e-07
Lua 5.3.0		OOLua 6.-92391e-08	OOLua 8.-83054e-08		OOLua 6.-66639e-08	OOLua 9.-1454000000001e-08		OOLua 1.-213471e-07	OOLua 1.-165102e-07
LuaJIT 2.0.3		OOLua 5.-39881e-08	OOLua 8.-49012e-08		OOLua 5.-01838e-08	OOLua 7.-65322e-08		OOLua 6.-80857e-08	OOLua 9.-28426000000001e-08

6.4.2 No userdata checks

Lua imp		**mfunc**			**vfunc**			**param**	
		cached	self		cached	self		cached	self
Lua 5.1.5		OOLua 5.-50472e-08	OOLua 7.-86604e-08		OOLua 5.-48013e-08	OOLua 8.-65963e-08		OOLua 8.-6675e-08	OOLua 8.-8671e-08
Lua 5.2.4		OOLua 5.-7768e-08	OOLua 7.-99339e-08		OOLua 5.-86034e-08	OOLua 8.-11962e-08		OOLua 8.-994849999[illegible] 08	OOLua 8.-[illegible]e- 08
Lua 5.3.0		OOLua 5.-68803e-08	OOLua 7.-95171e-08		OOLua 5.-87445e-08	OOLua 7.-98216e-08		OOLua 8.-82771e-08	OOLua 8.-87009e-08
LuaJIT 2.0.3		OOLua 4.-18646e-08	OOLua 6.-8805e-08		OOLua 3.-93808e-08	OOLua 6.-6519e-08		OOLua 4.-78118e-08	OOLua 7.-08243e-08

Chapter 7

Module Index

7.1 Modules

Here is a list of all modules:

Chapter 8

Namespace Index

8.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 9

Hierarchical Index

9.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 10

Class Index

10.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 11

File Index

11.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 12

Module Documentation

12.1 Library Configuration

Modules

- File Generation

  Lua module for generating configurable OOLua boilerplate code.
- File amalgamation

  Header and source file amalgamation.
- String Configuration

  Enables a string type to be treated as an integral.
- Error Reporting

  Defines how any errors are reported.
- Error Checking

  Defines the type of checks which will be performed.
- Shared Pointer

  Enable and configure library support for a shared pointer type.

12.1.1 Detailed Description

–[[

OOLua is configurable in a number of ways. You may change library limits using the file generation module and then regenerate core library files. Edit the oolua_config.h file to adjust Error Checking, Error Reporting and enabling support for Shared Pointer. Additionally, you may adjust which, if any, type is treated as a string integral via String Configuration.

12.2 File Generation

Lua module for generating configurable OOLua boilerplate code.

Functions

- function gen (options, path)

  Generate boilerplate header files.
- function default_details ()

  Returns the library defaults and details.
- function defaults ()

  Gets the default options as key(string) and value(number) entries in a table.

12.2.1 Detailed Description

Lua module for generating configurable OOLua boilerplate code. The "oolua_generate" Lua module provides information about the default limits for the library. It enables the generation of boilerplate code using user defined limits or regeneration of files with the default values. The details of these configurable values being :

```
return
{
    lua_params =
    {
        desc ='Maximum amount of parameters for a call to a Lua function'
        ,value=10
    }
    ,cpp_params =
    {
        desc='Maximum number of parameters a C++ function can have'
        ,value=8
    }
    ,constructor_params =
    {
        desc='Maximum amount of parameters for a constructor of a proxied type'
        ,value=5
    }
    ,class_functions =
    {
        desc='Maximum amount of class functions that can be registered for each proxied type'
        ,value=15
    }
}
```

The most common change to these options is the number of functions which can be registered for a proxy class, this limit applies individually to constant and none constant functions, base class methods that are registered in a base class do not decrease the count for a derived class.

Using the Lua interpreter to regenerate the OOLua files increasing this option whilst using default values for the remaining options:

```
lua -e "require'build_scripts.oolua_generate'.gen({class_functions=30},'include/')"
```

For convenience you do not need a version of Lua installed on a machine to run this module, Premake the project file generator used in OOLua already contains a copy of Lua 5.1 (it has some modifications to the core libraries). To generate the files with the same options as above :

```
premake4 --class_functions=30 oolua-gen
```

The module returns a table with the following functions

```
return { gen = gen, defaults=defaults, default_details=default_details
      }
```

12.2.2 Function Documentation

12.2.2.1 function default_details ( )

Returns the library defaults and details.

–]]

Returns a table detailing the library defaults and descriptions

```
return
{
    lua_params =
    {
        desc ='Maximum amount of parameters for a call to a Lua function'
        ,value=10
    }
    ,cpp_params =
    {
        desc='Maximum number of parameters a C++ function can have'
        ,value=8
    }
    ,constructor_params =
    {
        desc='Maximum amount of parameters for a constructor of a proxied type'
        ,value=5
    }
    ,class_functions =
    {
        desc='Maximum amount of class functions that can be registered for each proxied type'
        ,value=15
    }
}
```

Returns

Table of the format { config_option ={desc='blurb',value=0} }

12.2.2.2 function defaults ( )

Gets the default options as key(string) and value(number) entries in a table.

Modifies the table returned by default_details so the it is formatted correctly for any functions it will be passed to.

Returns

Table of the format { config_option = 0 }

See Also

default_details

12.2.2.3 function gen (options , path)

Generate boilerplate header files.

Parameters

options	[optional] Defaults to the library defaults

path	[optional] Defaults to the current working directory

Generates boilerplate C++ files code required for OOLua using the passed options or if an option is not present then the default is used. If Path is not nil then it is required to be a string which is slash postfixed.

12.3 File amalgamation

Header and source file amalgamation.

Functions

- function amalgamate (include_dir, src_dir, output_dir)

  Generates an amalgamated header and source file for the library.

12.3.1 Detailed Description

Header and source file amalgamation. The module "oolua_amalgamation" returns a table containing a single function, amalgamate.

```
return {
    amalgamate = amalgamate
}
```

This function concatenates all of the library's header files to produce the file oolua.h, and similarly the source files to produce the file oolua.cpp. The two files will contain all the functionality of the library and could quickly be integrated into your project

You can produce the files using either the module with Lua or premake4.

Lua module:

```
lua -e "require('build_scripts.oolua_amalgamation').amalgamate('./include/', './src/', './')"
```

Premake4:

Generating the amalgamated files with premake4, will create the files oolua.h and oolua.cpp that will be located in the directory "amal".

```
premake4 oolua-amalgam
```

12.3.2 Function Documentation

12.3.2.1 function amalgamate (include_dir , src_dir , output_dir)

Generates an amalgamated header and source file for the library.

–]]

Parameters

include_dir	Directory containing the header files
src_dir	Directory containing the source files
output_dir	Output directory for the amalgamated files

Concatenates all the header files from include_dir and separately the source files from src_dir, producing the outputs oolua.h and oolua.cpp respectively. These two files, located in output_dir, contain all the functionality of the library.

12.4 Known limitations

12.4.1 Incorrect creation of userdata

OOLua incorrectly creates a new userdata when it should reuse one which has already been created.

See Also

http://code.google.com/p/oolua/issues/detail?id=5

```
void differentRootsOfaTree_twoRootsPassedToLua_luaUdComparesEqual()
{
    OOLUA::register_class<DerivedFromTwoAbstractBasesAndAbstract3>(*m_lua);
    DerivedFromTwoAbstractBasesAndAbstract3 derived;
    Abstract2* a2 = &derived;
    Abstract3* a3 = &derived;
    OOLUA::push(*m_lua, a2);
    OOLUA::push(*m_lua, a3);
    OOLUA::INTERNAL::Lua_ud* ud_a2 = static_cast<OOLUA::INTERNAL::Lua_ud*>(lua_touserdata(*m_lua,-2));
    OOLUA::INTERNAL::Lua_ud* ud_a3 = static_cast<OOLUA::INTERNAL::Lua_ud*>(lua_touserdata(*m_lua,-1));
    CPPUNIT_ASSERT_EQUAL(true, ud_a2 == ud_a3);
}
```

12.5 String Configuration

Enables a string type to be treated as an integral.

Namespaces

- OOLUA::STRING

  Defines which type of string classes can be pulled and pushed from the stack with the public API and the DSL.

Classes

- struct OOLUA::STRING::only_std_string_conforming_with_c_str_method

  Defines the structure which checks for the method "c_str" which conforms to the std::string signature.
- struct OOLUA::STRING::is_integral_string_class

  Preforms the check on the type without including the string header.

Macros

- #define OOLUA_STD_STRING_IS_INTEGRAL

  Default: *Enabled*
- #define OOLUA_CLASS_OR_BASE_CONTAINS_METHOD(StructName, MethodSignature, MethodName)

  Creates a structure that enables checking a class type for a specific function signature that has a specific name.

Functions

- template<typename StringType >
  bool OOLUA::STRING::push (lua_State ∗const vm, StringType const &value)

  Function to which public API calls resolve to.
- template<typename StringType >
  bool OOLUA::STRING::pull (lua_State ∗const vm, StringType &value)

  Function to which public API calls resolve to.
- template<typename StringType >
  void OOLUA::STRING::get (lua_State ∗const vm, int idx, StringType &value)

  Internal function used by the DSL to retrieve a string from the stack.

12.5.1 Detailed Description

Enables a string type to be treated as an integral. String configuration enables a type to be treated as an alias and integral type for a Lua string, LUA_TSTRING. When a type is considered an integral it means that a heap allocation, for the object itself, does not occur and the instance is a value rather than a reference. The alternative to this, is for the string type to be proxied using the library's DSL.

Identification of the type is tested using the compile time value contained in is_integral_string_class. A provided macro, OOLUA_CLASS_OR_BASE_CONTAINS_METHOD, helps to create a structure that identifies the type and can be used in is_integral_string_class. The library file oolua_string.h contains examples of using this macro to identify a number of string types; such as std::string, C++03 and C++11 strings(std::string, std::wstring, std::u16string and std::u32string), wxString and QString.

Once a type can be identified in is_integral_string_class as an integral string alias, then a user should provide implementations for the three specific templated functions, (OOLUA::STRING::push, OOLUA::STRING::pull and OOLUA::STRING::get) in oolua_string.cpp.

12.5.2 Macro Definition Documentation

12.5.2.1 #define OOLUA_CLASS_OR_BASE_CONTAINS_METHOD(*StructName, MethodSignature, MethodName*)

Creates a structure that enables checking a class type for a specific function signature that has a specific name.

Parameters

StructName	The name of the structure this macro will create.
Method-Signature	The signature the checker will look for.
MethodName	The function's name which has the MethodSignature

12.5.2.2 #define OOLUA_STD_STRING_IS_INTEGRAL

Default: Enabled

Allows std::string to be a parameter or a return type for a function.

Note

This is always by value

Parameters

0	Disabled
1	Enabled

12.5.3 Function Documentation

12.5.3.1 template<typename StringType > void OOLUA::STRING::get (lua_State ∗const *vm,* int *idx,* StringType & *value*)

Internal function used by the DSL to retrieve a string from the stack.

The are a couple of differences between this function and pull. Firstly, pull retrieves the stack's top entry and pops it where as this function uses a stack index to identify the stack slot and the function does not pop the entry. Secondly, there is a difference in how errors are reported. Pull will either store and error or throw an exception, where as this function will eventually call lua_error.

12.6 DSL

The domain specific language(DSL) used for generating C++ bindings to Lua.

Modules

- Expressive

  Generates a proxy function where a user has expressed all the details.
- Minimalist

  Generates code using only the minimal amount of information.
- Exporting

  Exports member functions.

Macros

- #define OOLUA_PROXY(...)

  Starts the generation a proxy class.
- #define OOLUA_TAGS(...)

  Allows more information to be specified about the proxy class.
- #define OOLUA_PROXY_END

  Ends the generation of the proxy class.
- #define OOLUA_SCOPED_ENUM(Name, Entry)

  Creates a entry into a OOLUA_ENUMS block for a C++11 scoped enum.
- #define OOLUA_ENUM(EnumName)

  Creates a entry into a OOLUA_ENUMS block.
- #define OOLUA_ENUMS(EnumEntriesList)

  Creates a block into which enumerators can be defined with OOLUA_ENUM or OOLUA_SCOPED_ENUM.
- #define OOLUA_CTOR(...)

  Generates a constructor in a constructor block.
- #define OOLUA_CTORS(ConstructorEntriesList)

  Creates a block into which none default constructors can be defined using OOLUA_CTOR.
- #define OOLUA_MGET(...)

  Generates a getter, which is a constant function, to retrieve a public instance.
- #define OOLUA_MSET(...)

  Generates a setter, which is a none constant function, to set a public instance.
- #define OOLUA_MGET_MSET(...)

  Generates both a getter and a setter for a public instance.

12.6.1 Detailed Description

The domain specific language(DSL) used for generating C++ bindings to Lua. OOLua provides a DSL for defining C++ types which are to be made available to a Lua script. The intention of this DSL is to hide the implementation details whilst providing a simple and rememberable interface to perform the required actions. For the generation of function proxies, the DSL contains two sub categories named Minimalist and Expressive.

Note

"Optional" here means that extra macro parameters are optional, up to the configuration max for a specific operation.

12.6.2 Macro Definition Documentation

12.6.2.1 #define OOLUA_CTOR(...)

Generates a constructor in a constructor block.

See Also

OOLUA_CTORS

OOLUA_CTOR(ConstructorParameterList)

Parameters

Constructor-ParameterList	Comma separated list of parameters

Precondition

Size of ConstructorParameterList >0 and <= "constructor_params"

See Also

constructor_params

12.6.2.2 #define OOLUA_CTORS(*ConstructorEntriesList*)

Creates a block into which none default constructors can be defined using OOLUA_CTOR.

OOLUA_CTORS(ConstructorEntriesList)

Parameters

Constructor-EntriesList	List of OOLUA_CTOR

To enable the construction of an instance without using the default constructor, there must be a constructor block specified for the proxy type. The constructor block, OOLUA_CTORS, is where non-default constructor entries can be specified using an OOLUA_CTOR per entry.

Constructors are the only real type of overloading which is permitted by OOLua and there is an important point which should be noted. This being that OOLua will attempt to match the number and type of parameters on the stack with the amount and types specified for each OOLUA_CTOR entry. The order in which it will attempt the matching is the same order in which they were defined. When interacting with the Lua stack certain types can not be differentiated between, these include some integral types such as float, int, double etc and types which are of a proxy class type or derived from that type. OOLua implicitly converts between classes in a hierarchy even if a reference is required. This means for example that if there are constructors such as Foo::Foo(int) and Foo::Foo(float) it will depend on which was defined first in the OOLUA_CTORS block as to which will be invoked for a call such as Foo.new(1).

See Also

No_default_constructor

Note

An OOLUA_CTORS block without any OOLUA_CTOR entries is invalid.

12.6.2.3 #define OOLUA_ENUM(*EnumName*)

Creates a entry into a OOLUA_ENUMS block.

OOLUA_ENUM(EnumName)

Parameters

EnumName	The class enumeration name

12.6.2.4 #define OOLUA_ENUMS(*EnumEntriesList*)

Creates a block into which enumerators can be defined with OOLUA_ENUM or OOLUA_SCOPED_ENUM.

OOLUA_ENUMS(EnumEntriesList)

Parameters

EnumEntriesList	List which contains OOLUA_ENUM and/or OOLUA_SCOPED_ENUM entries.

Note

An OOLUA_ENUMS block without any OOLUA_ENUM or OOLUA_SCOPED_ENUM entries is invalid.

12.6.2.5 #define OOLUA_MGET(...)

Generates a getter, which is a constant function, to retrieve a public instance.

OOLUA_MGET(PublicName, Optional)

Parameters

PublicName	Name of the public variable to be proxied.
Optional	GetterName. Defaults to get_PublicName

Note

A generated getter for a pointer, or shared pointer, with a proxied pointee type, has an implicit OOLUA::maybe-_null trait applied.

12.6.2.6 #define OOLUA_MGET_MSET(...)

Generates both a getter and a setter for a public instance.

OOLUA_MGET_MSET(PublicName, Optional1, Optional2)

Parameters

PublicName	Name of the public variable to be proxied.
Optional1	GetterName. Defaults to get_PublicName
Optional2	SetterName. Defaults to set_PublicName

See Also

OOLUA_MGET and OOLUA_MSET

Note

If one optional parameter is supplied then both must be given and they must use different names.

12.6.2.7 #define OOLUA_MSET(...)

Generates a setter, which is a none constant function, to set a public instance.

OOLUA_MSET(PublicName, Optional)

Parameters

PublicName	Name of the public variable to be proxied.
Optional	SetterName. Defaults to set_PublicName

12.6.2.8 #define OOLUA_PROXY(...)

Starts the generation a proxy class.

OOLUA_PROXY(ClassName, Optional)

Parameters

ClassName	Class to be proxied
Optional	Comma seperated list of real base classes

Precondition

Each class specified in Optional must be a real base class of ClassName

12.6.2.9 #define OOLUA_SCOPED_ENUM(*Name, Entry*)

Creates a entry into a OOLUA_ENUMS block for a C++11 scoped enum.

OOLUA_SCOPED_ENUM(EnumName, Entry)

Parameters

Name	The class enumeration name which will be used to access it from Lua
Entry	The class enumeration scoped qualified name (minus the class type)

See Also

OOLUA_ENUM

12.6.2.10 #define OOLUA_TAGS(...)

Allows more information to be specified about the proxy class.

Tags provide a method to inform the library that the type :

- has relationship and/or mathematical operators
- is an abstract class
- doesn't have a default constructor or any public constructors
- has enumerations

For an exhaustive list of the possible tags see Tags.

OOLUA_TAGS(TagList)

Parameters

TagList	Comma separated list of Tags

Note

An OOLUA_TAGS list without any Tags entries is invalid.

12.7 Expressive

Generates a proxy function where a user has expressed all the details.

Macros

- #define OOLUA_MEM_FUNC(...)

  Generates a member function proxy which will also be the named FunctionName.
- #define OOLUA_MEM_FUNC_RENAME(...)

  Generates a member function proxy which will be the named ProxyFunctionName.
- #define OOLUA_MEM_FUNC_CONST(...)

  Generates a constant member function proxy which will also be the named FunctionName.
- #define OOLUA_MEM_FUNC_CONST_RENAME(...)

  Generates a constant member function which will be named ProxyFunctionName.
- #define OOLUA_C_FUNCTION(...)

  Generates a block which will call the C function FunctionName.

12.7.1 Detailed Description

Generates a proxy function where a user has expressed all the details. Generates a function for which the user has expressed all the parameters and the return type for a function. These types may also have Traits applied to them which the Minimalist section of the DSL does not allow.

12.7.2 Macro Definition Documentation

12.7.2.1 #define OOLUA_C_FUNCTION(...)

Generates a block which will call the C function FunctionName.

OOLUA_C_FUNCTION(FunctionReturnType,FunctionName, Optional)

Parameters

FunctionReturn-Type	
FunctionName	
Optional	Comma separated list of function parameter types

See Also

cpp_params

Precondition

The function in which this macro is contained must declare a lua_State pointer which can be identified by the name "vm"

```
extern void foo(int);
int l_foo(lua_State* vm)
{
    OOLUA_C_FUNCTION(void,foo,int)
}
```

Note

This macro should ideally be used as the last operation of a function body as control will return to the caller. Notice there is no return statement in l_foo

12.7.2.2 #define OOLUA_MEM_FUNC(...)

Generates a member function proxy which will also be the named FunctionName.

OOLUA_MEM_FUNC(FunctionReturnType, FunctionName, Optional)

Parameters

FunctionReturn-Type	
FunctionName	
Optional	: Comma separated list of function parameter types

See Also

cpp_params

12.7.2.3 #define OOLUA_MEM_FUNC_CONST(...)

Generates a constant member function proxy which will also be the named FunctionName.

OOLUA_MEM_FUNC_CONST(FunctionReturnType,FunctionName,Optional)

Parameters

FunctionReturn-Type	
FunctionName	
Optional	Comma separated list of function parameter types

See Also

cpp_params

12.7.2.4 #define OOLUA_MEM_FUNC_CONST_RENAME(...)

Generates a constant member function which will be named ProxyFunctionName.

OOLUA_MEM_FUNC_CONST_RENAME(ProxyFunctionName, FunctionReturnType, FunctionName,Optional)

Parameters

ProxyFunction-Name	
FunctionReturn-Type	
FunctionName	
Optional	Comma separated list of function parameter types

See Also

cpp_params

12.7.2.5 #define OOLUA_MEM_FUNC_RENAME(...)

Generates a member function proxy which will be the named ProxyFunctionName.

OOLUA_MEM_FUNC_RENAME(ProxyFunctionName, FunctionReturnType,FunctionName, Optional)

Parameters

ProxyFunction-Name	
FunctionReturn-Type	
FunctionName	
Optional	: Comma separated list of function parameter types

See Also

cpp_params

12.8 Minimalist

Generates code using only the minimal amount of information.

Macros

- #define OOLUA_MFUNC(...)

  Deduce and generate a proxy for a member function.
- #define OOLUA_MFUNC_CONST(...)

  Deduce and generate a proxy for a constant member function.
- #define OOLUA_CFUNC(...)

  Deduce and generate a proxy for a C function.
- #define OOLUA_SFUNC(...)

  Deduce and generate a proxy for a class static function.

12.8.1 Detailed Description

Generates code using only the minimal amount of information. Generates a proxy function using the only the minimal amount of information which is generally the name of the thing being proxied and possibly a new name for the proxy. If a new name is supplied then the function will be made available to Lua using it and this name must be used when Exporting the function.

This part of the DSL attempts to automatically determine the parameter types and return type for the function in question. However, if the function is overloaded then the compiler will be unable to resolve the function, due to the ambiguity, and will produce a compile time error. To help the compiler resolve this ambiguity, the user should specify more information using the corresponding, yet longer named, Expressive DSL entry.

The longer DSL name requires more information.

Note

No Traits can be expressed with this DSL group.

12.8.2 Macro Definition Documentation

12.8.2.1 #define OOLUA_CFUNC(...)

Deduce and generate a proxy for a C function.

OOLUA_CFUNC(FunctionName, ProxyFunctionName)

Parameters

FunctionName	Name of the C function to be proxied
ProxyFunction-Name	Name of the function to generate which will proxy FunctionName

See Also

cpp_params
OOLUA_C_FUNCTION

12.8.2.2 #define OOLUA_MFUNC(...)

Deduce and generate a proxy for a member function.

OOLUA_MFUNC(FunctionName, Optional)

Parameters

FunctionName	Name of the member function to be proxied
Optional	ProxyFunctionName. Defaults to FunctionName

See Also

cpp_params
OOLUA_MEM_FUNC
OOLUA_MEM_FUNC_RENAME

12.8.2.3 #define OOLUA_MFUNC_CONST(...)

Deduce and generate a proxy for a constant member function.

OOLUA_MFUNC_CONST(FunctionName, Optional)

Parameters

FunctionName	Name of the constant function to be proxied
Optional	ProxyFunctionName. Defaults to FunctionName

See Also

cpp_params
OOLUA_MEM_FUNC_CONST
OOLUA_MEM_FUNC_CONST_RENAME

12.8.2.4 #define OOLUA_SFUNC(...)

Deduce and generate a proxy for a class static function.

OOLUA_SFUNC(FunctionName, Optional)

Parameters

FunctionName	Name of the static function to be proxied
Optional	ProxyFunctionName. Defaults to FunctionName

Note

This function will not be exported and needs to be registered with OOLua see OOLUA::register_class_static

See Also

cpp_params

12.9 Exporting

Exports member functions.

Macros

- #define OOLUA_EXPORT_FUNCTIONS(...)

  Exports zero or more member functions which will be registered with Lua.
- #define OOLUA_EXPORT_FUNCTIONS_CONST(...)

  Exports zero or more const member functions which will be registered with Lua.
- #define OOLUA_EXPORT_NO_FUNCTIONS(Class)

  Inform that there are no functions of interest.

12.9.1 Detailed Description

Exports member functions. Exporting defines which member functions will be registered with Lua when the class type is registered. Even when there are no member functions to be exported you still need to inform OOLua about this. Calling an OOLUA_EXPORT∗ procedure in a header file is an error that will fail to compile.

See Also

OOLUA_EXPORT_FUNCTIONS
OOLUA_EXPORT_FUNCTIONS_CONST
OOLUA_EXPORT_NO_FUNCTIONS

12.9.2 Macro Definition Documentation

12.9.2.1 #define OOLUA_EXPORT_FUNCTIONS(...)

Exports zero or more member functions which will be registered with Lua.

OOLUA_EXPORT_FUNCTIONS(ClassName,Optional)

Parameters

ClassName	Name of class to which the function belong to
Optional	Comma separated list of member function names

See Also

class_functions

12.9.2.2 #define OOLUA_EXPORT_FUNCTIONS_CONST(...)

Exports zero or more const member functions which will be registered with Lua.

OOLUA_EXPORT_FUNCTIONS_CONST(ClassName,Optional)

Parameters

ClassName	Name of class to which the function belong to

Optional	Comma separated list of constant member function names

See Also

class_functions

12.9.2.3 #define OOLUA_EXPORT_NO_FUNCTIONS(*Class*)

Value:

```
EXPORT_OOLUA_FUNCTIONS_0_NON_CONST(Class)\
    EXPORT_OOLUA_FUNCTIONS_0_CONST(Class)
```

Inform that there are no functions of interest.

Parameters

Class	

12.10 Error Reporting

Defines how any errors are reported.

Modules

- Exception classes

Macros

- #define OOLUA_USE_EXCEPTIONS

  Default: *Disabled*
- #define OOLUA_STORE_LAST_ERROR

  Default: *Enabled*

Functions

- void OOLUA::reset_error_value (lua_State ∗vm)

  Reset the error state such that a call to OOLUA::get_last_error will return an empty string.
- std::string OOLUA::get_last_error (lua_State ∗vm)

  Returns the last stored error.

12.10.1 Detailed Description

Defines how any errors are reported. Errors can be reported either by using exceptions or storing a retrievable error string. Only one of these methods is allowed and this condition is enforced, yet also neither are required. If both options are disabled then it depends on OOLUA_DEBUG_CHECKS as to whether any error will be reported.

12.10.2 Macro Definition Documentation

12.10.2.1 #define OOLUA_STORE_LAST_ERROR

Default: Enabled

Stores an error message in the registry which overwrites any previously stored error. The last error to have occurred is retrievable via OOLUA::get_last_error.

See Also

OOLUA::get_last_error
OOLUA::reset_error_value

Parameters

0	Disabled
1	Enabled

12.10.2.2 #define OOLUA_USE_EXCEPTIONS

Default: Disabled

Throws exceptions from C++ code. This could be the return of a pcall, or from pulling an incorrect type off the stack when OOLUA_RUNTIME_CHECKS_ENABLED is enabled. It also prevents exceptions escaping from functions proxied by the library, enabling calls to such functions to be caught with pcall in Lua code.

Parameters

0	Disabled
1	Enabled

12.10.3 Function Documentation

12.10.3.1 std::string OOLUA::get_last_error (lua_State ∗ *vm*)

Returns the last stored error.

Returns

empty string if there is not an error else the error message

See Also

Error Reporting

Note

This function is a nop when OOLUA_STORE_LAST_ERROR is not enabled

12.10.3.2 void OOLUA::reset_error_value (lua_State ∗ *vm*)

Reset the error state such that a call to OOLUA::get_last_error will return an empty string.

See Also

Error Reporting

Note

This function is a nop when OOLUA_STORE_LAST_ERROR is not enabled

12.11 Error Checking

Defines the type of checks which will be performed.

Macros

- #define OOLUA_RUNTIME_CHECKS_ENABLED

  Default: *Enabled*
- #define OOLUA_CHECK_EVERY_USERDATA_IS_CREATED_BY_OOLUA

  Default: *Enabled*
- #define OOLUA_USERDATA_OPTIMISATION

  Default: *Enabled*
- #define OOLUA_DEBUG_CHECKS

  Default: *Enabled when DEBUG or _DEBUG is defined*
- #define OOLUA_SANDBOX

  Default: *Disabled*

12.11.1 Detailed Description

Defines the type of checks which will be performed.

12.11.2 Macro Definition Documentation

12.11.2.1 #define OOLUA_CHECK_EVERY_USERDATA_IS_CREATED_BY_OOLUA

Default: Enabled

Does what it says on the tin, only valid when OOLUA_RUNTIME_CHECKS_ENABLED is enabled

Parameters

0	Disabled
1	Enabled

12.11.2.2 #define OOLUA_DEBUG_CHECKS

Default: Enabled when DEBUG or _DEBUG is defined

Provides the following

- asserts on NULL pointers
- adds a stack trace to errors reported by pcall
- asserts on errors if both OOLUA_USE_EXCEPTIONS and OOLUA_STORE_LAST_ERROR are both disabled

Parameters

0	Disabled

1	Enabled

12.11.2.3 #define OOLUA_RUNTIME_CHECKS_ENABLED

Default: Enabled

Checks that a type being pulled off the stack is of the correct type, if this is a proxy type, it also checks the userdata on the stack was created by OOLua

Parameters

0	Disabled
1	Enabled

12.11.2.4 #define OOLUA_SANDBOX

Default: Disabled

check everything

Parameters

0	Disabled
1	Enabled

12.11.2.5 #define OOLUA_USERDATA_OPTIMISATION

Default: Enabled

Userdata optimisation which checks for a magic cookie to try and ensure it was created by OOLua, by default this is on when userdata checking is on. Turning this off by setting it to zero will use a slower yet correct (as correct as can be) method.

Only meaningful when OOLUA_CHECK_EVERY_USERDATA_IS_CREATED_BY_OOLUA is enabled

Parameters

0	Disabled
1	Enabled

12.12 Shared Pointer

Enable and configure library support for a shared pointer type.

Macros

- #define OOLUA_USE_SHARED_PTR

  Default: *Disabled*
- #define OOLUA_SHARED_HEADER

  Default: *MSC: <memory> other compilers: <tr1/memory>*
- #define OOLUA_SHARED_TYPE

  Default: *std::tr1::shared_ptr*
- #define OOLUA_SHARED_CONST_CAST

  Default: *std::tr1::const_pointer_cast*
- #define OOLUA_NEW_POINTER_DEFAULT_IS_SHARED_TYPE

  Default: *Disabled*

12.12.1 Detailed Description

Enable and configure library support for a shared pointer type. Requirements for the shared pointer type.

- The type must be non intrusive to the underlying type
- Have a "get" member function which returns a raw pointer
- Be constructable from a more derived shared_ptr
- Have a constructor which enables construction of shared<foo const> from shared<foo>
- Be of uniform size
- Have a const cast template function

Defaults for the configuration options in this category only apply when shared pointer support is enabled.

12.12.2 Macro Definition Documentation

12.12.2.1 #define OOLUA_NEW_POINTER_DEFAULT_IS_SHARED_TYPE

Default: Disabled

When compiled with support for a shared pointer type (OOLUA_USE_SHARED_PTR) and in a situation that requires the allocation of a proxy type, then how the situation is handled depends upon this configuration value and possibly Shared Tags defined for the proxy. The resulting pointer can either be a 'Raw' pointer or a 'Shared' pointer that retains shared ownership . Allocation of a proxy occurs for functions and operators that return a non-integral on the C stack and constructors.

Configuration value	Has Shared tag	Has No_shared tag	Pointer type
Disabled	No	X	Raw
Disabled	Yes	X	Shared
Enabled	X	No	Shared
Enabled	X	Yes	Raw

Parameters

0	Disabled
1	Enabled

See Also

OOLUA_USE_SHARED_PTR
OOLUA::Shared
OOLUA::No_shared

12.12.2.2 #define OOLUA_SHARED_CONST_CAST

Default: std::tr1::const_pointer_cast

Templated function which casts away constness for the shared pointer type.

12.12.2.3 #define OOLUA_SHARED_HEADER

Default: MSC: <memory> other compilers: <tr1/memory>

Header file for the shared pointer type, library code will include the header using :

```
#include OOLUA_SHARED_HEADER
```

12.12.2.4 #define OOLUA_SHARED_TYPE

Default: std::tr1::shared_ptr

The templated shared pointer type.

12.12.2.5 #define OOLUA_USE_SHARED_PTR

Default: Disabled

Configuration option to enable or disable the support of a shared pointer type for OOLua proxies.

When enabled the library supports:

- pushing a shared pointer to the stack
- pulling a shared pointer from the stack(only defined when it is a shared pointer)
- pulling a raw pointer from a stack that contains a shared pointer(It is up to the user of the library to ensure the type will not be garbage collected)
- functions that return a shared pointer
- functions that return a shared pointer and have the OOLUA::maybe_null trait
- functions that take a shared pointer as a parameter
- decaying of a shared pointer to a raw pointer for functions parameters(The raw pointer is defined to be valid for the duration of the call)

  Note

  The OOLUA::Shared and OOLUA::No_shared tags maybe ignored, as they are dependent on the value of OOLUA_NEW_POINTER_DEFAULT_IS_SHARED_TYPE

Parameters

0	Disabled
1	Enabled

12.13 Exception classes

Classes

- struct OOLUA::Exception

  Base class for OOLua exceptions.
- struct OOLUA::Syntax_error

  Reports LUA_ERRSYNTAX.
- struct OOLUA::Runtime_error

  Reports LUA_ERRRUN.
- struct OOLUA::Memory_error

  Reports LUA_ERRMEM.
- struct OOLUA::File_error

  Reports LUA_ERRFILE.
- struct OOLUA::Type_error

  Reports that a type pulled from the stack was not the type that was asked for.

12.13.1 Detailed Description

12.14 Traits

Provides direction and/or ownership information.

Modules

- Parameter Traits

    DSL Traits for function parameter types.
- Function Return Traits

    DSL traits for function return types.
- Stack Traits

    Public API traits which control a change of ownership.

12.14.1 Detailed Description

Provides direction and/or ownership information. The general naming conventions for traits are :

- Parameter Traits : end in "_p"
- Function Return Traits : end in "_return" or "_null"
- Stack Traits : end in "_ptr".

12.15 Parameter Traits

DSL Traits for function parameter types.

Classes

- struct OOLUA::in_p< T >

  Input parameter trait.
- struct OOLUA::out_p< T >

  Output parameter trait.
- struct OOLUA::in_out_p< T >

  Input and output parameter trait.
- struct OOLUA::cpp_in_p< T >

  Input parameter trait which will be owned by C++.
- struct OOLUA::lua_out_p< T >

  Output parameter trait which will be owned by Lua.
- struct OOLUA::light_p< T >

  Input parameter trait.
- struct OOLUA::calling_lua_state

  Special parameter type.

12.15.1 Detailed Description

DSL Traits for function parameter types. Traits which allow control of ownership include in their name either "lua" or "cpp"; directional traits contain "in", "out" or a combination.

12.16 Function Return Traits

DSL traits for function return types.

Classes

- struct OOLUA::light_return< T >

  Return trait for a light userdata type.
- struct OOLUA::lua_return< T >

  Return trait for a type which will be owned by Lua.
- struct OOLUA::maybe_null< T >

  Return trait for a pointer which at runtime maybe NULL.
- struct OOLUA::shared_return< T >

  Converts a raw pointer return type to the supported shared pointer type.

12.16.1 Detailed Description

DSL traits for function return types. Some of the these traits allow for NULL pointers to be returned from functions, which was something commonly requested for the library. When such a trait is used and the runtime value is NULL, Lua's value of nil will be pushed to the stack.

12.17 Stack Traits

Public API traits which control a change of ownership.

Classes

- struct OOLUA::cpp_acquire_ptr< T >

  Change of ownership to C++.
- struct OOLUA::lua_acquire_ptr< T >

  Change of ownership to Lua.

12.17.1 Detailed Description

Public API traits which control a change of ownership. Valid to usage for the Public API which interact with the Lua stack.

12.18 Tags

Possible members for OOLUA_TAGS which help express more information about a class which is to be proxied.

Modules

- Operator Tags

  Informs that a class has an operator exposed.
- Shared Tags

  Tags to override the default behaviour the library was compiled with.

Namespaces

- OOLUA

  This is the root namespace of the Library.

Classes

- struct OOLUA::Abstract

  The class being mirrored is an abstract class.
- struct OOLUA::No_default_constructor

  There is not a default constructor in the public interface yet there are other constructors.
- struct OOLUA::No_public_constructors

  There are no constructors in the public interface.
- struct OOLUA::No_public_destructor

  There is not a destructor in the public interface and OOLua will not attempt to delete an instance of this type.
- struct OOLUA::Register_class_enums

  The class has enums to register.

12.18.1 Detailed Description

Possible members for OOLUA_TAGS which help express more information about a class which is to be proxied.

12.19 Operator Tags

Informs that a class has an operator exposed.

Classes

- struct OOLUA::Less_op

  Less than operator is defined for the type.
- struct OOLUA::Equal_op

  Equal operator is defined for the type.
- struct OOLUA::Not_equal_op

  Not equal operator is defined for the type.
- struct OOLUA::Less_equal_op

  Less than or equal operator is defined for the type.
- struct OOLUA::Div_op

  Division operator is defined for the type.
- struct OOLUA::Mul_op

  Multiplication operator is defined for the type.
- struct OOLUA::Sub_op

  Subtraction operator is defined for the type.
- struct OOLUA::Add_op

  Addition operator is defined for the type.

12.19.1 Detailed Description

Informs that a class has an operator exposed. Operator Tags inform OOLua that a class exposes one or more of the operators supported:

- Less_op
- Equal_op
- Not_equal_op
- Less_equal_op
- Div_op
- Mul_op
- Sub_op
- Add_op

12.20 Shared Tags

Tags to override the default behaviour the library was compiled with.

Classes

- struct OOLUA::Shared

  Overrides the configuration behaviour when creating proxied types.
- struct OOLUA::No_shared

  Overrides the configuration behaviour when creating proxied types.

12.20.1 Detailed Description

Tags to override the default behaviour the library was compiled with.

Chapter 13

Namespace Documentation

13.1 OOLUA Namespace Reference

This is the root namespace of the Library.

Namespaces

- STRING

    Defines which type of string classes can be pulled and pushed from the stack with the public API and the DSL.

Classes

- struct Lua_function

    Structure which is used to call a Lua function.
- class Proxy_class

    A template wrapper for class objects of type T used by the script binding.
- struct Lua_ref

    A typed wrapper for a Lua reference.
- class Script

    OOLua helper class.
- class Table

    Wrapper around a table in Lua which allows easy usage.
- struct in_p

    Input parameter trait.
- struct out_p

    Output parameter trait.
- struct in_out_p

    Input and output parameter trait.
- struct cpp_in_p

    Input parameter trait which will be owned by C++.
- struct lua_out_p

    Output parameter trait which will be owned by Lua.
- struct light_p

    Input parameter trait.
- struct light_return

    Return trait for a light userdata type.
- struct lua_return

Return trait for a type which will be owned by Lua.

- struct maybe_null

  Return trait for a pointer which at runtime maybe NULL.
- struct shared_return

  Converts a raw pointer return type to the supported shared pointer type.
- struct cpp_acquire_ptr

  Change of ownership to C++.
- struct lua_acquire_ptr

  Change of ownership to Lua.
- struct calling_lua_state

  Special parameter type.
- struct in_p$<$ char $*$ $>$

  Specialisation for C style strings.
- struct Abstract

  The class being mirrored is an abstract class.
- struct Less_op

  Less than operator is defined for the type.
- struct Equal_op

  Equal operator is defined for the type.
- struct Not_equal_op

  Not equal operator is defined for the type.
- struct Less_equal_op

  Less than or equal operator is defined for the type.
- struct Div_op

  Division operator is defined for the type.
- struct Mul_op

  Multiplication operator is defined for the type.
- struct Sub_op

  Subtraction operator is defined for the type.
- struct Add_op

  Addition operator is defined for the type.
- struct No_default_constructor

  There is not a default constructor in the public interface yet there are other constructors.
- struct No_public_constructors

  There are no constructors in the public interface.
- struct No_public_destructor

  There is not a destructor in the public interface and OOLua will not attempt to delete an instance of this type.
- struct Register_class_enums

  The class has enums to register.
- struct Shared

  Overrides the configuration behaviour when creating proxied types.
- struct No_shared

  Overrides the configuration behaviour when creating proxied types.
- struct Exception

  Base class for OOLua exceptions.
- struct Syntax_error

  Reports LUA_ERRSYNTAX.
- struct Runtime_error

  Reports LUA_ERRRUN.
- struct Memory_error

  Reports LUA_ERRMEM.

- struct File_error

  Reports LUA_ERRFILE.
- struct Type_error

  Reports that a type pulled from the stack was not the type that was asked for.

Typedefs

- typedef Lua_ref< LUA_TTABLE > Lua_table_ref

  Typedef helper for a LUA_TTABLE registry reference.
- typedef Lua_ref< LUA_TFUNCTION > Lua_func_ref

  Typedef helper for a LUA_TFUNCTION registry reference.

Enumerations

- enum Owner { No_change, Cpp, Lua }

Functions

- template<typename T >
  bool set_global (lua_State ∗vm, char const ∗name, T &instance)

  Helper function to set a Lua global variable.
- bool set_global (lua_State ∗vm, char const ∗name, lua_CFunction instance)

  None template version.
- void set_global_to_nil (lua_State ∗vm, char const ∗name)

  Helper function to set a Lua global variable to nil.
- template<typename T >
  bool get_global (lua_State ∗vm, char const ∗name, T &instance)

  Helper function to set a Lua global variable.
- bool load_chunk (lua_State ∗vm, std::string const &chunk)

  Loads a chunk leaving the resulting function on the stack.
- bool run_chunk (lua_State ∗vm, std::string const &chunk)

  Loads and runs a chunk of code.
- bool load_file (lua_State ∗vm, std::string const &filename)

  Loads a file leaving the resulting function on the stack.
- bool run_file (lua_State ∗vm, std::string const &filename)

  Loads and runs the file.
- void reset_error_value (lua_State ∗vm)

  Reset the error state such that a call to OOLUA::get_last_error will return an empty string.
- std::string get_last_error (lua_State ∗vm)

  Returns the last stored error.
- bool idxs_equal (lua_State ∗vm, int idx0, int idx1)
- bool can_xmove (lua_State ∗vm0, lua_State ∗vm1)

  Uses the Lua C API to check if it is valid to move data between the states.
- void setup_user_lua_state (lua_State ∗vm)

  Sets up a lua_State to work with OOLua.
- template<typename T >
  void register_class (lua_State ∗vm)

  Registers the class type T and it's bases with an instance of lua_State.
- template<typename T , typename K , typename V >
  void register_class_static (lua_State ∗const vm, K const &key, V const &value)

Registers a key K and value V entry into class T.

- template<typename T , typename T1 >
  void table_set_value (lua_State ∗vm, int table_index, T const &key, T1 const &value)

  The table is at table_index which can be either absolute or pseudo in the stack table is left at the index.
- template<typename T , typename T1 >
  bool table_at (lua_State ∗vm, int const table_index, T const &key, T1 &value)

  The table is at table_index which can be either absolute or pseudo in the stack table is left at the index.
- void new_table (lua_State ∗vm, OOLUA::Table &t)

  Creates a new valid OOLUA::Table.
- OOLUA::Table new_table (lua_State ∗vm)

  Creates a new valid Table.

- template<typename T >
  bool pull (lua_State ∗const vm, T &value)

  Pulls the top element off the stack and pops it.
- template<typename T >
  bool pull (lua_State ∗const vm, T ∗&value)

  Pulls the top element off the stack and pops it.
- bool pull (lua_State ∗const vm, void ∗&lightud)

  Pulls the top element off the stack and pops it.
- bool pull (lua_State ∗const vm, bool &value)

  Pulls the top element off the stack and pops it.
- bool pull (lua_State ∗const vm, double &value)

  Pulls the top element off the stack and pops it.
- bool pull (lua_State ∗const vm, float &value)

  Pulls the top element off the stack and pops it.
- bool pull (lua_State ∗const vm, oolua_CFunction &value)

  Pulls the top element off the stack and pops it.
- bool pull (lua_State ∗const vm, Table &value)

  Pulls the top element off the stack and pops it.
- template<typename T >
  bool pull (lua_State ∗const vm, cpp_acquire_ptr< T > const &value)

  Pulls the top element off the stack and pops it.

- template<typename T >
  bool push (lua_State ∗const vm, T const &value)

  Pushes an instance to top of the Lua stack.
- template<typename T >
  bool push (lua_State ∗const vm, OOLUA::lua_acquire_ptr< T > const &value)

  Pushes an instance to top of the Lua stack.
- template<typename T >
  bool push (lua_State ∗const vm, T ∗const &value)

  Pushes an instance to top of the Lua stack.
- bool push (lua_State ∗const vm, void ∗lightud)

  Pushes an instance to top of the Lua stack.
- bool push (lua_State ∗const vm, bool const &value)

  Pushes an instance to top of the Lua stack.
- bool push (lua_State ∗const vm, char ∗const &value)

  Pushes an instance to top of the Lua stack.
- bool push (lua_State ∗const vm, char const ∗const &value)

  Pushes an instance to top of the Lua stack.
- bool push (lua_State ∗const vm, double const &value)

Pushes an instance to top of the Lua stack.
- bool push (lua_State ∗const vm, float const &value)

  Pushes an instance to top of the Lua stack.
- bool push (lua_State ∗const vm, oolua_CFunction const &value)

  Pushes an instance to top of the Lua stack.
- bool push (lua_State ∗const vm, Table const &value)

  Pushes an instance to top of the Lua stack.

Variables

- static const char version_str [ ] = OOLUA_STRINGISE(OOLUA_VERSION_MAJ) "." OOLUA_STRINGISE(O-OLUA_VERSION_MIN) "." OOLUA_STRINGISE(OOLUA_VERSION_PATCH)

  OOLua version string.
- static const int version_number = 2∗10000+0∗1000+1

  OOLua version int.

13.1.1 Detailed Description

This is the root namespace of the Library. There are sub namespaces contained in OOLUA yet mostly these are not meant for general usage, instead this namespace contains all Public API functions, structures etc.

13.1.2 Enumeration Type Documentation

13.1.2.1 enum **OOLUA::Owner**

Enumerator

No_change	No change of ownership
Cpp	Change in ownership, C++ will now own the instance
Lua	Change in ownership, Lua will now own the instance

13.1.3 Function Documentation

13.1.3.1 bool OOLUA::can_xmove (**lua_State** ∗ *vm0,* **lua_State** ∗ *vm1*)

Uses the Lua C API to check if it is valid to move data between the states.

lua_xmove returns without doing any work if the two pointers are the same and fails when using LUA_USE_APIC-HECK and the states do not share the same global_State.

It may be fine to move numbers between different unrelated states when Lua was not compiled with LUA_USE_A-PICHECK but this function would still return false for that scenario.

Parameters

in	*vm0*	
in	*vm1*	

Returns

true if vm0 and vm1 are different yet none NULL related states, else false

13.1.3.2 template< typename T > bool OOLUA::get_global (**lua_State** ∗ *vm,* char const ∗ *name,* T & *instance*)

Helper function to set a Lua global variable.

Template Parameters

T	Type for instance

Parameters

`in`	*vm*	lua_State
`in`	*name*	Global name to query
`out`	*instance*	Any variable which is valid to pull from the stack

Returns

Boolean indicating if the operation was successful

See Also

Error Reporting

13.1.3.3 bool OOLUA::idxs_equal (lua_State ∗ *vm,* int *idx0,* int *idx1*)

Compares two valid indices on the stack of vm.

Compares the two indicies which will take into consideration metamethods when present for the types.

Parameters

`in`	*vm*	The lua_State in which to preform the operation
`in`	*idx0*	Valid stack index
`in`	*idx1*	Valid stack index

Returns

bool Result of the comparison

13.1.3.4 bool OOLUA::load_chunk (lua_State ∗ *vm,* std::string const & *chunk*)

Loads a chunk leaving the resulting function on the stack.

Parameters

`in`	*vm*	Lua virtual machine. Taken from Lua manual : An opaque structure that points to a thread and indirectly (through the thread) to the whole state of a Lua interpreter. The Lua library is fully reentrant: it has no global variables. All information about a state is accessible through this structure.
`in`	*chunk*	

13.1.3.5 bool OOLUA::load_file (lua_State ∗ *vm,* std::string const & *filename*)

Loads a file leaving the resulting function on the stack.

Parameters

`in`	*vm*	Lua virtual machine. Taken from Lua manual : An opaque structure that points to a thread and indirectly (through the thread) to the whole state of a Lua interpreter. The Lua library is fully reentrant: it has no global variables. All information about a state is accessible through this structure.
`in`	*filename*	

13.1.3.6 void OOLUA::new_table (lua_State ∗ *vm*, OOLUA::Table & *t*)

Creates a new valid OOLUA::Table.

Parameters

`in`	*vm*	The lua_State in which to create the table.
`in,out`	*t*	Table which will hold the newly created valid table.

Postcondition

stack is the same on exit as entry and t will be an instance on which valid returns true.

13.1.3.7 OOLUA::Table OOLUA::new_table (lua_State ∗ *vm*)

Creates a new valid Table.

Postcondition

stack is the same on exit as entry

Parameters

`in`	*vm*	The lua_State in which to create the table

Returns

A newly constructed Table on which valid will return true.

13.1.3.8 bool OOLUA::pull (lua_State ∗const *vm,* void ∗& *lightud*)

Pulls the top element off the stack and pops it.

In stack terms this is a top followed by pop.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by throwing an exception.

See Also

Error Reporting
Exception classes

13.1.3.9 bool OOLUA::pull (lua_State ∗const *vm,* bool & *value*)

Pulls the top element off the stack and pops it.

In stack terms this is a top followed by pop.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by throwing an exception.

See Also

Error Reporting
Exception classes

13.1.3.10 bool OOLUA::pull (lua_State ∗const *vm*, double & *value*)

Pulls the top element off the stack and pops it.

In stack terms this is a top followed by pop.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by throwing an exception.

See Also

Error Reporting
Exception classes

13.1.3.11 bool OOLUA::pull (lua_State ∗const *vm*, float & *value*)

Pulls the top element off the stack and pops it.

In stack terms this is a top followed by pop.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by throwing an exception.

See Also

Error Reporting
Exception classes

13.1.3.12 bool OOLUA::pull (lua_State ∗const *vm*, oolua_CFunction & *value*)

Pulls the top element off the stack and pops it.

In stack terms this is a top followed by pop.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by throwing an exception.

See Also

Error Reporting
Exception classes

13.1.3.13 bool OOLUA::pull (lua_State ∗const *vm*, Table & *value*)

Pulls the top element off the stack and pops it.

In stack terms this is a top followed by pop.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by throwing an exception.

See Also

Error Reporting
Exception classes

13.1.3.14 template< typename T > bool OOLUA::pull (lua_State ∗const *vm*, cpp_acquire_ptr< T > const & *value*)

Pulls the top element off the stack and pops it.

In stack terms this is a top followed by pop.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by throwing an exception.

See Also

Error Reporting
Exception classes

13.1.3.15 template< typename T > bool OOLUA::pull (lua_State ∗const *vm*, T & *value*) `[inline]`

Pulls the top element off the stack and pops it.

In stack terms this is a top followed by pop.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by throwing an exception.

See Also

Error Reporting
Exception classes

13.1.3.16 template< typename T > bool OOLUA::pull (lua_State ∗const *vm*, T ∗& *value*) `[inline]`

Pulls the top element off the stack and pops it.

In stack terms this is a top followed by pop.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by throwing an exception.

See Also

Error Reporting
Exception classes

13.1.3.17 bool OOLUA::push (lua_State ∗const *vm*, void ∗ *lightud*)

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.18 bool OOLUA::push (lua_State ∗const *vm*, bool const & *value*)

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.19 bool OOLUA::push (lua_State ∗const *vm*, char ∗const & *value*)

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.20 bool OOLUA::push (lua_State *const *vm,* char const *const & *value*)

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.21 bool OOLUA::push (lua_State *const *vm,* double const & *value*)

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.22 bool OOLUA::push (lua_State *const *vm,* float const & *value*)

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.23 bool OOLUA::push (lua_State *const *vm,* oolua_CFunction const & *value*)

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.24 bool OOLUA::push (lua_State *const *vm,* Table const & *value*)

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.25 template<typename T > bool OOLUA::push (lua_State *const *vm,* T const & *value*) `[inline]`

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.26 template<typename T > bool OOLUA::push (lua_State *const *vm*, OOLUA::lua_acquire_ptr< T > const & *value*) [inline]

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.27 template<typename T > bool OOLUA::push (lua_State *const *vm*, T *const & *value*) [inline]

Pushes an instance to top of the Lua stack.

Returns

If OOLUA_STORE_LAST_ERROR is set to one then the the return value will indicate success or failure, if OOLUA_USE_EXCEPTIONS is set to one then failure will always be reported by the throwing of an exception.

Note

Although all push methods return a boolean, most simply return true. The only versions which can return false are functions which operate on full userdata and values which are associated with a Lua universe.

See Also

OOLUA::can_xmove
Error Reporting
Exception classes

13.1.3.28 template<typename T > void OOLUA::register_class (lua_State * *vm*) [inline]

Registers the class type T and it's bases with an instance of lua_State.

Template Parameters

T	Class type to register with OOLua

Registers a class type T for which there is a Proxy_class and also registers it's bases, if it has any. The function preforms a check to see if the type has already been registered with the instance and is safe to be called multiple times with a Lua universe.

Parameters

in	*vm*	Universe to register the class with.

13.1.3.29 template<typename T , typename K , typename V > void OOLUA::register_class_static (lua_State *const *vm,* K const & *key,* V const & *value*) [inline]

Registers a key K and value V entry into class T.

Template Parameters

T	Class type to register the static for
K	Key
V	Value

Parameters

in	*vm*	lua_State
in	*key*	Key to register in type T,
in	*value*	The data to associate with key in the class type T.

13.1.3.30 bool OOLUA::run_chunk (lua_State * *vm,* std::string const & *chunk*)

Loads and runs a chunk of code.

Parameters

in	*vm*	Lua virtual machine. Taken from Lua manual : An opaque structure that points to a thread and indirectly (through the thread) to the whole state of a Lua interpreter. The Lua library is fully reentrant: it has no global variables. All information about a state is accessible through this structure.
in	*chunk*	

13.1.3.31 bool OOLUA::run_file (lua_State * *vm,* std::string const & *filename*)

Loads and runs the file.

Parameters

in	*vm*	Lua virtual machine. Taken from Lua manual : An opaque structure that points to a thread and indirectly (through the thread) to the whole state of a Lua interpreter. The Lua library is fully reentrant: it has no global variables. All information about a state is accessible through this structure.
in	*filename*	

13.1.3.32 template<typename T > bool OOLUA::set_global (lua_State * *vm,* char const * *name,* T & *instance*)

Helper function to set a Lua global variable.

Template Parameters

T	Type for instance

Parameters

in	*vm*	lua_State
in	*name*	Global name to set
in	*instance*	Any variable which is valid to push to the stack

Returns

Boolean indicating if the operation was successful

See Also

Error Reporting

13.1.3.33 bool OOLUA::set_global (lua_State ∗ *vm,* char const ∗ *name,* lua_CFunction *instance*)

None template version.

Enables setting a global with a value of lua_CFunction without requiring you make a reference to the function.

Parameters

in	*vm*	The lua_State to work on
in	*name*	String which is used for the global name
in	*instance*	The lua_CFuntion which will be set at the global value for name

13.1.3.34 void OOLUA::set_global_to_nil (lua_State ∗ *vm,* char const ∗ *name*)

Helper function to set a Lua global variable to nil.

Parameters

in	*vm*	lua_State
in	*name*	Global name to set

13.1.3.35 void OOLUA::setup_user_lua_state (lua_State ∗ *vm*)

Sets up a lua_State to work with OOLua.

If you want to use OOLua with a lua_State you already have active or supplied by some third party, then calling this function adds the necessary tables and globals for it to work with OOLua.

Parameters

in	*vm*	lua_State to be initialise by OOLua

13.2 OOLUA::STRING Namespace Reference

Defines which type of string classes can be pulled and pushed from the stack with the public API and the DSL.

Classes

- struct only_std_string_conforming_with_c_str_method
  Defines the structure which checks for the method "c_str" which conforms to the std::string signature.
- struct is_integral_string_class
  Preforms the check on the type without including the string header.

Functions

- template<typename StringType >
  bool push (lua_State *const vm, StringType const &value)

  Function to which public API calls resolve to.
- template<typename StringType >
  bool pull (lua_State *const vm, StringType &value)

  Function to which public API calls resolve to.
- template<typename StringType >
  void get (lua_State *const vm, int idx, StringType &value)

  Internal function used by the DSL to retrieve a string from the stack.

13.2.1 Detailed Description

Defines which type of string classes can be pulled and pushed from the stack with the public API and the DSL. I would really like to be able to forward declare string types in a cross platform way. For example when using G-CC we could, but really shouldn't, use bits/stringfwd.h. However this is just not possible. Instead this file and its accompanying source file, oolua_string.cpp, provide a way to not include the string header through out the library DSL headers, yet still be able to use a string type when needed. It also facilitates the addition, and therefore interaction, of other string types with Lua. To allow a string type to be compatible with OOLua requires that three functions are defined for the type(push, pull and get) and is_integral_string_class::value must be a compile time value that returns one for the type.

Chapter 14

Class Documentation

14.1 OOLUA::Abstract Struct Reference

The class being mirrored is an abstract class.

`#include <proxy_tags.h>`

14.1.1 Detailed Description

The class being mirrored is an abstract class.

When OOLua encounters the Abstract tag it will not look for any constructors for the type and the type will not be constructable from Lua. Specifying an OOLUA_CTORS block will have no effect and such a block will be ignored.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.2 OOLUA::Add_op Struct Reference

Addition operator is defined for the type.

`#include <proxy_tags.h>`

14.2.1 Detailed Description

Addition operator is defined for the type.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.3 OOLUA::calling_lua_state Struct Reference

Special parameter type.

`#include <oolua_traits.h>`

14.3.1 Detailed Description

Special parameter type.

This is different from all other traits as it does not take a type, yet is a type. It informs OOLua that the calling state is a parameter for a function

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.4 OOLUA::cpp_acquire_ptr< T > Struct Template Reference

Change of ownership to C++.

`#include <oolua_traits.h>`

14.4.1 Detailed Description

template<typename T>struct OOLUA::cpp_acquire_ptr< T >

Change of ownership to C++.

Informs the library that C++ will take control of the pointer being used and call delete on it when appropriate. This is only valid for public API functions which OOLUA::pull from the stack.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.5 OOLUA::cpp_in_p< T > Struct Template Reference

Input parameter trait which will be owned by C++.

`#include <oolua_traits.h>`

14.5.1 Detailed Description

template<typename T>struct OOLUA::cpp_in_p< T >

Input parameter trait which will be owned by C++.

Parameter supplied via Lua changes ownership to C++.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.6 OOLUA::Div_op Struct Reference

Division operator is defined for the type.

`#include <proxy_tags.h>`

14.6.1 Detailed Description

Division operator is defined for the type.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.7 OOLUA::Equal_op Struct Reference

Equal operator is defined for the type.

`#include <proxy_tags.h>`

14.7.1 Detailed Description

Equal operator is defined for the type.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.8 OOLUA::Exception Struct Reference

Base class for OOLua exceptions.

`#include <oolua_exception.h>`

Inherits std::exception.

Inherited by OOLUA::File_error, OOLUA::Memory_error, OOLUA::Runtime_error, OOLUA::Syntax_error, and OO-LUA::Type_error.

14.8.1 Detailed Description

Base class for OOLua exceptions.

See Also

Error Reporting

The documentation for this struct was generated from the following file:

- oolua_exception.h

14.9 OOLUA::File_error Struct Reference

Reports LUA_ERRFILE.

`#include <oolua_exception.h>`

Inherits OOLUA::Exception.

14.9.1 Detailed Description

Reports LUA_ERRFILE.

See Also

Error Reporting

The documentation for this struct was generated from the following file:

- oolua_exception.h

14.10 HasIntMember Struct Reference

```
#include <cpp_userdata_function_params.h>
```

14.10.1 Detailed Description

[CppOutParamsUserData]

The documentation for this struct was generated from the following file:

- cpp_userdata_function_params.h

14.11 Hello_moon Class Reference

Inherits TestFixture.

Public Member Functions

- void hello_minimalist_function ()
- void hello_expressive_function ()
- void hello_cast_minimalist_function ()
- void hello_function_no_registration ()
- void hello_class_function ()

14.11.1 Detailed Description

[HelloMoonClass]

14.11.2 Member Function Documentation

14.11.2.1 void Hello_moon::hello_cast_minimalist_function ( ) `[inline]`

[HelloMoonCFuncExpressiveUsage] [HelloMoonCFuncCastUsage]

14.11.2.2 void Hello_moon::hello_class_function ( ) `[inline]`

[HelloMoonCFuncAndProxyUsageLua]

14.11.2.3 void Hello_moon::hello_expressive_function ( ) [inline]

[HelloMoonCFuncMinimalistUsage] [HelloMoonCFuncExpressiveUsage]

14.11.2.4 void Hello_moon::hello_function_no_registration ( ) [inline]

[HelloMoonCFuncCastUsage] [HelloMoonCFuncAndProxyUsageLua]

14.11.2.5 void Hello_moon::hello_minimalist_function ( ) [inline]

[HelloMoonCFuncMinimalistUsage]

The documentation for this class was generated from the following file:

- hello_moon.cpp

14.12 OOLUA::in_out_p< T > Struct Template Reference

Input and output parameter trait.

`#include <oolua_traits.h>`

14.12.1 Detailed Description

template<typename T>struct OOLUA::in_out_p< T >

Input and output parameter trait.

The calling Lua procedure supplies the parameter to the proxied function, the value of the parameter after the proxied call will be passed back to the calling procedure as a return value. No change of ownership occurs.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.13 OOLUA::in_p< T > Struct Template Reference

Input parameter trait.

`#include <oolua_traits.h>`

14.13.1 Detailed Description

template<typename T>struct OOLUA::in_p< T >

Input parameter trait.

The calling Lua procedure supplies the parameter to the proxied function. No change of ownership occurs.

Note

This is the default trait used for function parameters when no trait is supplied.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.14 OOLUA::in_p< char * > Struct Template Reference

Specialisation for C style strings.

`#include <oolua_traits.h>`

14.14.1 Detailed Description

template<>struct OOLUA::in_p< char * >

Specialisation for C style strings.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.15 OOLUA::STRING::is_integral_string_class Struct Reference

Preforms the check on the type without including the string header.

`#include <oolua_string.h>`

14.15.1 Detailed Description

Preforms the check on the type without including the string header.

To add a different string class type, see the commented out macros in oolua_string.h.

See Also

OOLUA_STD_STRING_IS_INTEGRAL

The documentation for this struct was generated from the following file:

- oolua_string.h

14.16 OOLUA::Less_equal_op Struct Reference

Less than or equal operator is defined for the type.

`#include <proxy_tags.h>`

14.16.1 Detailed Description

Less than or equal operator is defined for the type.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.17 OOLUA::Less_op Struct Reference

Less than operator is defined for the type.

`#include <proxy_tags.h>`

14.17.1 Detailed Description

Less than operator is defined for the type.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.18 OOLUA::light_p< T > Struct Template Reference

Input parameter trait.

`#include <oolua_traits.h>`

14.18.1 Detailed Description

template<typename T>struct OOLUA::light_p< T >

Input parameter trait.

The calling Lua procedure supplies a LUA_TLIGHTUSERDATA which will be cast to the requested T type. If T is not the correct type for the light userdata then the casting is undefined. A light userdata is never owned by Lua

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.19 OOLUA::light_return< T > Struct Template Reference

Return trait for a light userdata type.

`#include <oolua_traits.h>`

14.19.1 Detailed Description

template<typename T>struct OOLUA::light_return< T >

Return trait for a light userdata type.

The type returned from the function is either a void pointer or a pointer to another type. When the function returns, it will push a LUA_TLIGHTUSERDATA to the stack even when the pointer is NULL; therefore a NULL pointer using this traits is never converted to a Lua nil value. A light userdata is also never owned by Lua and OOLua does not store any type information for the it; light_return is a black box which when used incorrectly will invoke undefined behaviour.

This is only valid for function return types.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.20 OOLUA::lua_acquire_ptr< T > Struct Template Reference

Change of ownership to Lua.

`#include <oolua_traits.h>`

14.20.1 Detailed Description

template<typename T>struct OOLUA::lua_acquire_ptr< T >

Change of ownership to Lua.

Informs the library that Lua will take control of the pointer being used and call delete on it when appropriate. This is only valid for public API functions which OOLUA::push to the stack.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.21 OOLUA::Lua_function Struct Reference

Structure which is used to call a Lua function.

`#include <oolua_function.h>`

Public Member Functions

- Lua_function ()

    Default constructor initialises the object.
- Lua_function (lua_State ∗vm)

    Binds the state vm to this instance.
- void bind_script (lua_State ∗const vm)

    Sets the state in which functions will be called.

- template<typename FUNC_TYPE >
  bool operator() (FUNC_TYPE const &func)

    Function call operator.
- template<typename FUNC_TYPE , typename P1 >
  bool operator() (FUNC_TYPE const &func, P1 p1)

    Function call operator.
- template<typename FUNC_TYPE , typename P1 , typename P2 >
  bool operator() (FUNC_TYPE const &func, P1 p1, P2 p2)

    Function call operator.
- template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 >
  bool operator() (FUNC_TYPE const &func, P1 p1, P2 p2, P3 p3)

    Function call operator.
- template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 >
  bool operator() (FUNC_TYPE const &func, P1 p1, P2 p2, P3 p3, P4 p4)

    Function call operator.
- template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 >
  bool operator() (FUNC_TYPE const &func, P1 p1, P2 p2, P3 p3, P4 p4, P5 p5)

    Function call operator.
- template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 >
  bool operator() (FUNC_TYPE const &func, P1 p1, P2 p2, P3 p3, P4 p4, P5 p5, P6 p6)

    Function call operator.
- template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 , typename P7 >
  bool operator() (FUNC_TYPE const &func, P1 p1, P2 p2, P3 p3, P4 p4, P5 p5, P6 p6, P7 p7)

    Function call operator.

- template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 , typename P7 , typename P8 >
  bool operator() (FUNC_TYPE const &func, P1 p1, P2 p2, P3 p3, P4 p4, P5 p5, P6 p6, P7 p7, P8 p8)

  Function call operator.
- template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 , typename P7 , typename P8 , typename P9 >
  bool operator() (FUNC_TYPE const &func, P1 p1, P2 p2, P3 p3, P4 p4, P5 p5, P6 p6, P7 p7, P8 p8, P9 p9)

  Function call operator.
- template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 , typename P7 , typename P8 , typename P9 , typename P10 >
  bool operator() (FUNC_TYPE const &func, P1 p1, P2 p2, P3 p3, P4 p4, P5 p5, P6 p6, P7 p7, P8 p8, P9 p9, P10 p10)

  Function call operator.

14.21.1 Detailed Description

Structure which is used to call a Lua function.

Calling a Lua function, from C++ code using OOLua's API, can be achieved using a Lua_function object. This is a state bound caller, and the state in which the callee will be invoked is specified either in the constructor or via the bind_script member function.

To invoke a callee, the OOLUA::Lua_function type uses a call operator. The operator's first parameter must be the callee and it can be specified using one of the following types:

- std::string A function in the bound state's global table
- OOLUA::Lua_func_ref A reference to a function
- int A valid stack index If the callee is identified via a valid stack index, then this index will remain on the stack at same absolute location after the caller has returned.

The call operator is also overloaded to enable the passing of parameters to the callee; the maximum number of parameters is defined by the configurable value "lua_params".

14.21.2 Constructor & Destructor Documentation

14.21.2.1 OOLUA::Lua_function::Lua_function ( )

Default constructor initialises the object.

Postcondition

Any call to a function call operator will cause an error until a lua_State is bound via bind_script

14.21.3 Member Function Documentation

14.21.3.1 void OOLUA::Lua_function::bind_script (lua_State *const *vm*)

Sets the state in which functions will be called.

Parameters

`in`	*vm*	The state to bind to the instance.

14.21.3.2 template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1,* P2 *p2,* P3 *p3,* P4 *p4,* P5 *p5*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.3 template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 , typename P7 , typename P8 , typename P9 , typename P10 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1,* P2 *p2,* P3 *p3,* P4 *p4,* P5 *p5,* P6 *p6,* P7 *p7,* P8 *p8,* P9 *p9,* P10 *p10*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.4 template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 , typename P7 , typename P8 , typename P9 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1,* P2 *p2,* P3 *p3,* P4 *p4,* P5 *p5,* P6 *p6,* P7 *p7,* P8 *p8,* P9 *p9*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.5 template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 , typename P7 , typename P8 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1,* P2 *p2,* P3 *p3,* P4 *p4,* P5 *p5,* P6 *p6,* P7 *p7,* P8 *p8*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.6 template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 , typename P7 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1,* P2 *p2,* P3 *p3,* P4 *p4,* P5 *p5,* P6 *p6,* P7 *p7*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.7 template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 , typename P5 , typename P6 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1,* P2 *p2,* P3 *p3,* P4 *p4,* P5 *p5,* P6 *p6*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.8 template<typename FUNC_TYPE , typename P1 , typename P2 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1,* P2 *p2*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.9 template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 , typename P4 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1,* P2 *p2,* P3 *p3,* P4 *p4*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.10 template<typename FUNC_TYPE , typename P1 , typename P2 , typename P3 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1,* P2 *p2,* P3 *p3*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.11 template<typename FUNC_TYPE > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

14.21.3.12 template<typename FUNC_TYPE , typename P1 > bool OOLUA::Lua_function::operator() (FUNC_TYPE const & *func,* P1 *p1*)

Function call operator.

Returns

Result indicating success

Template Parameters

FUNC_TYPE	

See Also

Error Reporting

The documentation for this struct was generated from the following file:

- oolua_function.h

14.22 OOLUA::lua_out_p< T > Struct Template Reference

Output parameter trait which will be owned by Lua.

```
#include <oolua_traits.h>
```

14.22.1 Detailed Description

template<typename T>struct OOLUA::lua_out_p< T >

Output parameter trait which will be owned by Lua.

Lua code does not pass an instance to the C++ function, yet the pushed back value after the function call will be owned by Lua. This is meaningful only if called with a type which has a proxy and it is by reference, otherwise undefined.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.23 OOLUA::Lua_ref< ID > Struct Template Reference

A typed wrapper for a Lua reference.

```
#include <oolua_ref.h>
```

Public Member Functions

- Lua_ref ()

  Initialises the instance so that a call to valid will return false.
- Lua_ref (lua_State ∗const vm, int const &ref)

  Sets the lua_State and reference for the instance.
- Lua_ref (lua_State ∗const vm)

  Sets the lua_State for the instance and initialises the instance so that a call to valid will return false.
- Lua_ref (Lua_ref const &rhs) OOLUA_DEFAULT

  Creates a copy of rhs.
- ∼Lua_ref () OOLUA_DEFAULT

  Destructor which releases a valid reference, removing the value from the registry.
- bool valid () const

  Returns true if both the Lua instance is not NULL and the registry reference is not invalid.
- lua_State ∗ state () const

  Returns the lua_State associated with the Lua reference.
- int const & ref () const

  Returns the integer Lua registry reference value.
- void set_ref (lua_State ∗const vm, int const &ref) OOLUA_DEFAULT

  Sets the stored reference and state.

- bool operator== (Lua_ref const &rhs) const

  Compares this instance reference with the right hand side operand using lua_rawequal.
- Lua_ref & operator= (Lua_ref const &rhs)

  Makes this instance a copy of rhs.
- void swap (Lua_ref &rhs)

  Swaps the Lua instance and the registry reference with rhs.

14.23.1 Detailed Description

template<int ID>struct OOLUA::Lua_ref< ID >

A typed wrapper for a Lua reference.

The Lua_ref templated class stores a reference using Lua's reference system luaL_ref and luaL_unref, along with a lua_State. The reason this class stores the lua_State is to make it difficult to use the reference with another universe. A reference from the same Lua universe, even if it is from a different lua_State, is valid to be used in the universe.

The class takes ownership of any reference passed either to the two argument constructor or the set_ref function. On going out of scope a valid reference is guaranteed to be released, you may also force a release by passing an instance to swap for which valid returns false.

There are two special values for the reference which Lua provides, both of which OOLua will treat as an invalid reference:

- LUA_REFNIL luaL_ref return value to indicate it encountered a nil object at the location the ref was asked for
- LUA_NOREF guaranteed to be different from any reference return by luaL_ref

  Template Parameters

ID	Lua type as returned by lua_type

Note

- Universe: A call to luaL_newstate or lua_newstate creates a Lua universe and a universe is completely independent of any other universe. lua_newthread and coroutine.create, create a lua_State in an already existing universe.

  Term first heard in a Lua mailing list post by Mark Hamburg.

14.23.2 Constructor & Destructor Documentation

14.23.2.1 template<int ID> OOLUA::Lua_ref< ID >::Lua_ref (lua_State ∗const *vm*, int const & *ref*)

Sets the lua_State and reference for the instance.

Note

This does not preform any validation on the parameters and it is perfectly acceptable to pass parameters such that a call to valid will return false.

Parameters

`in`	*vm*	lua_State for which the ref is coupled with.
`in`	*ref*	Registry reference or registry special value for this instance.

14.23.2.2 template<int ID> OOLUA::Lua_ref< ID >::Lua_ref (lua_State ∗const *vm*) `[explicit]`

Sets the lua_State for the instance and initialises the instance so that a call to valid will return false.

Parameters

`in`	*vm*	lua_State for which this instance is coupled with.

14.23.2.3 template<int ID> OOLUA::Lua_ref< ID >::Lua_ref (Lua_ref< ID > const & *rhs*)

Creates a copy of rhs.

If rhs is valid then creates a new Lua reference to the value which rhs refers to, otherwise it initialises this instance so that a Lua_ref::valid call returns false.

Parameters

`in`	*rhs*	Reference for which this instance will initialise its internal state from.

14.23.3 Member Function Documentation

14.23.3.1 template<int ID> Lua_ref& OOLUA::Lua_ref< ID >::operator= (Lua_ref< ID > const & *rhs*)

Makes this instance a copy of rhs.

Parameters

in	*rhs*	The instance to make a copy of

Note

Even self assignment makes a copy, yet it will refer to the same actual Lua instance. It doesn't seem correct for every assignment to pay for a branch just to keep the internal reference id the same.

14.23.3.2 template<int ID> bool **OOLUA::Lua_ref**< ID >::operator== (**Lua_ref**< ID > const & *rhs*) const

Compares this instance reference with the right hand side operand using lua_rawequal.

Parameters

in	*rhs*	Right hand side operand for the operator.

An invalid reference compares equal with any other invalid reference regardless of the lua_State members. This operator can produce different results for Lua versions 5.1 and 5.2. In the latter pushing the same C function twice to the stack using lua_pushcclosure and then comparing them will return true, yet in 5.1 this will return false.

Returns

bool Result of the comparison.

14.23.3.3 template<int ID> void **OOLUA::Lua_ref**< ID >::set_ref (**lua_State** ∗const *vm,* int const & *ref*)

Sets the stored reference and state.

Releases any currently stored reference and takes ownership of the passed reference.

Parameters

in	*vm*	lua_State to associated the reference with.
in	*ref*	Registry reference id for which the instance takes ownership of.

14.23.3.4 template<int ID> void **OOLUA::Lua_ref**< ID >::swap (**Lua_ref**< ID > & *rhs*)

Swaps the Lua instance and the registry reference with rhs.

Swaps the lua_State and reference with rhs, this is a simple swap and does not call luaL_ref therefore it will not create any new references.

Parameters

in,out	*rhs*	Reference which will re-initialise this instance's state and which will receive the internal state of this instance as it was before the swap.

The documentation for this struct was generated from the following file:

- oolua_ref.h

14.24 OOLUA::lua_return< T > Struct Template Reference

Return trait for a type which will be owned by Lua.

```
#include <oolua_traits.h>
```

14.24.1 Detailed Description

template<typename T>struct OOLUA::lua_return< T >

Return trait for a type which will be owned by Lua.

The type returned from the function is a heap allocated instance whose ownership will be controlled by Lua. This is only valid for function return types.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.25 lua_State Struct Reference

Lua virtual machine.

14.25.1 Detailed Description

Lua virtual machine.

Taken from Lua manual : An opaque structure that points to a thread and indirectly (through the thread) to the whole state of a Lua interpreter. The Lua library is fully reentrant: it has no global variables. All information about a state is accessible through this structure.

The documentation for this struct was generated from the following file:

- oolua.dox

14.26 OOLUA::maybe_null< T > Struct Template Reference

Return trait for a pointer which at runtime maybe NULL.

```
#include <oolua_traits.h>
```

14.26.1 Detailed Description

template<typename T>struct OOLUA::maybe_null< T >

Return trait for a pointer which at runtime maybe NULL.

The type returned from the function is a pointer instance whose runtime value maybe NULL. If it is NULL then lua_pushnil will be called else the pointer will be pushed as normal. No change of ownership will occur for the type. This is only valid for function return types.

Note

> To be consistent in naming this should really be called maybe_null_return, however I feel this would be too long a name for the trait so "return" has been dropped.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.27 OOLUA::Memory_error Struct Reference

Reports LUA_ERRMEM.

```
#include <oolua_exception.h>
```

Inherits OOLUA::Exception.

14.27.1 Detailed Description

Reports LUA_ERRMEM.

See Also

Error Reporting

The documentation for this struct was generated from the following file:

- oolua_exception.h

14.28 MockOutParamsUserData Class Reference

```
#include <cpp_out_params.h>
```

Inherits OutParamsUserData.

14.28.1 Detailed Description

[CppOutParamsUserData]

The documentation for this class was generated from the following file:

- cpp_out_params.h

14.29 OOLUA::Mul_op Struct Reference

Multiplication operator is defined for the type.

```
#include <proxy_tags.h>
```

14.29.1 Detailed Description

Multiplication operator is defined for the type.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.30 OOLUA::No_default_constructor Struct Reference

There is not a default constructor in the public interface yet there are other constructors.

```
#include <proxy_tags.h>
```

14.30.1 Detailed Description

There is not a default constructor in the public interface yet there are other constructors.

There is not a public default constructor or you do not wish to expose such a constructor, yet there are other constructors which will be specified by OOLUA_CTOR entries inside a OOLUA_CTOR block.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.31 OOLUA::No_public_constructors Struct Reference

There are no constructors in the public interface.

`#include <proxy_tags.h>`

14.31.1 Detailed Description

There are no constructors in the public interface.

When OOLua encounters this tag it will not look for any constructors for the type and the type will not be constructable from Lua. Specifying an OOLUA_CTORS block will have no effect and such a block will be ignored.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.32 OOLUA::No_public_destructor Struct Reference

There is not a destructor in the public interface and OOLua will not attempt to delete an instance of this type.

`#include <proxy_tags.h>`

14.32.1 Detailed Description

There is not a destructor in the public interface and OOLua will not attempt to delete an instance of this type.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.33 OOLUA::No_shared Struct Reference

Overrides the configuration behaviour when creating proxied types.

`#include <proxy_tags.h>`

14.33.1 Detailed Description

Overrides the configuration behaviour when creating proxied types.

When the library is compiled with shared pointer support and it is also configured to create shared pointers by default, then this tag overrides that behaviour for the proxy type.

Note

If the library is configured to not use the shared pointer type by default then this tag is ignored when present in a proxy tag block.

See Also

OOLUA_USE_SHARED_PTR
OOLUA_NEW_POINTER_DEFAULT_IS_SHARED_TYPE

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.34 OOLUA::Not_equal_op Struct Reference

Not equal operator is defined for the type.

`#include <proxy_tags.h>`

14.34.1 Detailed Description

Not equal operator is defined for the type.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.35 OOLUA::STRING::only_std_string_conforming_with_c_str_method Struct Reference

Defines the structure which checks for the method "c_str" which conforms to the std::string signature.

`#include <oolua_string.h>`

14.35.1 Detailed Description

Defines the structure which checks for the method "c_str" which conforms to the std::string signature.

The documentation for this struct was generated from the following file:

- oolua_string.h

14.36 OOLUA::out_p< T > Struct Template Reference

Output parameter trait.

`#include <oolua_traits.h>`

14.36.1 Detailed Description

template<typename T>struct OOLUA::out_p< T >

Output parameter trait.

The calling Lua procedure does not pass the parameter to the proxied function, instead one is created using the default constructor and passed to the proxied function. The result after the proxied call with be returned to the calling procedure. If this is a type which has a proxy then it will cause a heap allocation of the type, which Lua will own.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.37 OutParamsUserData Class Reference

`#include <cpp_out_params.h>`

Inherited by MockOutParamsUserData.

14.37.1 Detailed Description

[CppOutParamsUserData]

The documentation for this class was generated from the following file:

- cpp_out_params.h

14.38 OOLUA::Proxy_class< T > Class Template Reference

A template wrapper for class objects of type T used by the script binding.

`#include <proxy_class.h>`

14.38.1 Detailed Description

template<typename T>class OOLUA::Proxy_class< T >

A template wrapper for class objects of type T used by the script binding.

Template Parameters

T	Type that is being proxied

See Also

DSL for the macros which are used to define a proxy class.

The documentation for this class was generated from the following file:

- oolua_pull.h

14.39 OOLUA::Register_class_enums Struct Reference

The class has enums to register.

`#include <proxy_tags.h>`

14.39.1 Detailed Description

The class has enums to register.

The class has enums which are specified inside the OOLUA_ENUMS block, these entries will be registered with a lua_State when the proxy type is.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.40 ReturnOrder Struct Reference

14.40.1 Detailed Description

[CppTraitReturnOrderOneParam]

The documentation for this struct was generated from the following file:

- return_order.cpp

14.41 OOLUA::Runtime_error Struct Reference

Reports LUA_ERRRUN.

`#include <oolua_exception.h>`

Inherits OOLUA::Exception.

14.41.1 Detailed Description

Reports LUA_ERRRUN.

See Also

Error Reporting

The documentation for this struct was generated from the following file:

- oolua_exception.h

14.42 Say Struct Reference

14.42.1 Detailed Description

[HelloMoonCFuncExpressiveProxy] [HelloMoonClass]

The documentation for this struct was generated from the following file:

- hello_moon.cpp

14.43 OOLUA::Script Class Reference

OOLua helper class.

`#include <oolua_script.h>`

Public Member Functions

- Script ()

    Initialises the instance.
- ∼Script ()

    Releases the bound lua_State if it is not NULL.

- int stack_count ()

    Returns the stack count from the lua_State.
- operator lua_State ∗ () const

    Conversion operator so that a Script instance can be passed in place of a lua_State pointer.
- lua_State ∗const & state () const

    Sometimes you may want to be explicit.
- void gc ()

    Performs a garbage collection on the state.

- template<typename T >
  void register_class ()

    Helper function.
- template<typename T >
  void register_class (T ∗)

    Helper function.
- template<typename T , typename K , typename V >
  void register_class_static (K const &k, V const &v)

    Helper function.

- bool run_file (std::string const &filename)

    Helper function.
- bool load_file (std::string const &filename)

    Helper function.
- bool load_chunk (std::string const &chunk)

    Helper function.
- bool run_chunk (std::string const &chunk)

    Helper function.

- template<typename T >
  bool pull (T &t)

    Helper function.
- template<typename T >
  bool push (T const &t)

    Helper function.

Public Attributes

- Lua_function call

14.43.1 Detailed Description

OOLua helper class.

OOLua is purposely designed not to be dependent on the Script class and therefore passes around its dependency of a lua_State instance. The Script class is only a helper and anything you can do with it can be accomplished by using a Lua_function struct, calling OOLUA namespaced functions or using the Lua C API.

Script provides :

- scoping of a lua_State pointer
- access to the lua_State pointer via a cast operator and function
- methods to register types
- binding a Lua_function instance to call functions
- member functions for a little state management
- setting up the state to work with OOLua

Note

This class is not copy constructible or assignable. To accomplish this a counted reference to the lua_State would need to be maintained.
If you do not want to or can not use this class please see setup_user_lua_state

14.43.2 Constructor & Destructor Documentation

14.43.2.1 OOLUA::Script::Script ( )

Initialises the instance.

- Creates a new Lua universe
- Binds the public member call with the lua_State
- Sets the lua_State up so that it will work with OOLUA::Proxy_class.

14.43.3 Member Function Documentation

14.43.3.1 bool OOLUA::Script::load_chunk (std::string const & *chunk*)

Helper function.

See Also

OOLUA::load_chunk

14.43.3.2 bool OOLUA::Script::load_file (std::string const & *filename*)

Helper function.

See Also

OOLUA::load_file

14.43.3.3 template< typename T > bool OOLUA::Script::pull (T & *t*) [inline]

Helper function.

See Also

OOLUA::pull

14.43.3.4 template< typename T > bool OOLUA::Script::push (T const & *t*) [inline]

Helper function.

See Also

OOLUA::push

14.43.3.5 template< typename T > void OOLUA::Script::register_class ( ) [inline]

Helper function.

See Also

OOLUA::register_class

14.43.3.6 template< typename T > void OOLUA::Script::register_class (T *) [inline]

Helper function.

See Also

OOLUA::register_class

14.43.3.7 template< typename T , typename K , typename V > void OOLUA::Script::register_class_static (K const & *k*, V const & *v*) [inline]

Helper function.

See Also

OOLUA::register_class_static

14.43.3.8 bool OOLUA::Script::run_chunk (std::string const & *chunk*)

Helper function.

See Also

OOLUA::run_chunk

14.43.3.9 bool OOLUA::Script::run_file (std::string const & *filename*)

Helper function.

See Also

OOLUA::run_file

14.43.3.10 lua_State∗ const& OOLUA::Script::state ( ) const [inline]

Sometimes you may want to be explicit.

See Also

Script::operator()

14.43.4 Member Data Documentation

14.43.4.1 Lua_function OOLUA::Script::call

Function object instance which can be used to call Lua functions

The documentation for this class was generated from the following file:

- oolua_script.h

14.44 OOLUA::Shared Struct Reference

Overrides the configuration behaviour when creating proxied types.

```
#include <proxy_tags.h>
```

14.44.1 Detailed Description

Overrides the configuration behaviour when creating proxied types.

When the library is compiled with shared pointer support and it is not configured to create shared pointers by default, then this tag overrides that behaviour for the proxy type.

Note

If the library is configured to use the shared pointer type by default then this tag is ignored when present in a proxy tag block.

See Also

OOLUA_USE_SHARED_PTR
OOLUA_NEW_POINTER_DEFAULT_IS_SHARED_TYPE

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.45 OOLUA::shared_return< T > Struct Template Reference

Converts a raw pointer return type to the supported shared pointer type.

`#include <oolua_traits.h>`

14.45.1 Detailed Description

template<typename T>struct OOLUA::shared_return< T >

Converts a raw pointer return type to the supported shared pointer type.

A shared_return does not define that a function returns a shared_ptr instead it informs the proxy to create a new shared object for the returned pointer. This trait therefore requires that there is not a reference to the pointer already known to the library.

The documentation for this struct was generated from the following file:

- oolua_traits.h

14.46 Stub1 Struct Reference

`#include <cpp_stub_classes.h>`

14.46.1 Detailed Description

[UsedAsMinimalClass]

The documentation for this struct was generated from the following file:

- cpp_stub_classes.h

14.47 Stub2 Struct Reference

`#include <cpp_stub_classes.h>`

14.47.1 Detailed Description

[UsedAsMinimalClass]

The documentation for this struct was generated from the following file:

- cpp_stub_classes.h

14.48 OOLUA::Sub_op Struct Reference

Subtraction operator is defined for the type.

`#include <proxy_tags.h>`

14.48.1 Detailed Description

Subtraction operator is defined for the type.

The documentation for this struct was generated from the following file:

- proxy_tags.h

14.49 OOLUA::Syntax_error Struct Reference

Reports LUA_ERRSYNTAX.

`#include <oolua_exception.h>`

Inherits OOLUA::Exception.

14.49.1 Detailed Description

Reports LUA_ERRSYNTAX.

See Also

Error Reporting

The documentation for this struct was generated from the following file:

- oolua_exception.h

14.50 OOLUA::Table Class Reference

Wrapper around a table in Lua which allows easy usage.

`#include <oolua_table.h>`

Public Member Functions

- bool valid () const

  Returns a boolean which is the result of checking the state of the internal Lua_func_ref.
- void traverse (traverse_do_function do_)

  Traverses the table using oolua_pairs.
- lua_State ∗ state () const

  Provides access to the associated lua_State.

- Table ()

  Default creates an object on which a call to valid returns false.
- Table (Lua_table_ref const &ref)

  Initialises the reference to be an instance of the same registry reference or an invalid table if ref.valid() == false.
- Table (lua_State ∗const vm, std::string const &name)

  Sets the lua_State and calls Lua_table::set_table.
- Table (Table const &rhs)

  Default creates an object on which a call to valid returns false.
- Table & operator= (Table const &rhs)

  Assigns a copy of rhs's internal state to this instance.

- void bind_script (lua_State ∗const vm)

  Associates the instance with the lua_State vm.
- void set_table (std::string const &name)

  Order of trying to initialise :
- void set_ref (lua_State ∗const vm, int const &ref)

  Initailises the internal Lua_func_ref to the id ref.
- void swap (Table &rhs)

  Swaps the internal Lua_func_ref and rhs.m_table_ref.

- template<typename T , typename T1 >
  void try_at (T const &key, T1 &value)

  Function which throws on an error.
- template<typename T , typename T1 >
  bool safe_at (T const &key, T1 &value)

  A safe version of at, which will always return a boolean indicating the success of the function call.
- template<typename T , typename T1 >
  T1 & at (T const &key, T1 &value)

- template<typename T , typename T1 >
  void set (T const &key, T1 const &value)

  Inserts the key value pair into the table if key is not present else it updates the table's key entry.
- template<typename T >
  void remove (T const &key)

  Removes the key from the table by setting it's value to nil.

14.50.1 Detailed Description

Wrapper around a table in Lua which allows easy usage.

Table provides a simple typed C++ interface for the Lua unordered and ordered associative container of the same name. Operations which use the Lua stack ensure that the stack is the same on exit as it was on entry, OOLua tries to force a clean stack(OOLua and the Lua stack).

Any value can be retrieved or set from the table via the use of the template member functions set, at or safe_at. If the value asked for is not the correct type located in the position an error can be reported, the type of which depends on Error Reporting and the function which was called. See individual member function documentation for details.

Note

The member function try_at is only defined when exceptions are enabled for the library.

14.50.2 Member Function Documentation

14.50.2.1 template<typename T , typename T1 > T1 & Table::at (T const & *key*, T1 & *value*) `[inline]`

Template Parameters

T	Key type
T1	Value type

Parameters

in	*key*	
out	*value*	zreturn The same instance as value

Note

No error checking.
It is undefined to call this function when:

- table or the key are invalid
- table does not contain the key
- value is not the correct type

See Also

Lua_table::safe_at
Lua_table::try_at

14.50.2.2 void OOLUA::Table::bind_script (lua_State ∗const *vm*)

Associates the instance with the lua_State vm.

Associates the instance with the lua_State vm. If the table already has a lua_State bound to it

- If the Current bound instance is not equal to vm and the table has a valid reference, it releases the currently set reference and sets vm as the bound instance.

14.50.2.3 Table& OOLUA::Table::operator= (Table const & *rhs*)

Assigns a copy of rhs's internal state to this instance.

If this table is valid then the operator will release the registry reference before assigning a copy of rhs to this instance.

Parameters

in	*rhs*	Table from which to copy the table reference

Returns

This instance.

See Also

OOLUA::Lua_ref assignment operator

14.50.2.4 template<typename T , typename T1 > bool Table::safe_at (T const & *key,* T1 & *value*) [inline]

A safe version of at, which will always return a boolean indicating the success of the function call.

This function will not throw an exception when exceptions are enabled for the library.

Template Parameters

T	Key type
T1	Value type

Parameters

`in`	*key*	
`out`	*value*	

14.50.2.5 void OOLUA::Table::set_table (std::string const & *name*)

Order of trying to initialise :

- name.empty() == true: Creates an invalid object.
- name found as a table in Lua global: Swaps the internal Lua_func_ref with an instance initialised to an id obtained from the Lua registry.
- name found as a table in Lua registry: Swaps the internal Lua_func_ref with an instance initialised to an id obtained from the Lua registry.
- else Swaps the internal Lua_func_ref with an uninitialised instance.

14.50.2.6 void OOLUA::Table::traverse (traverse_do_function *do_*)

Traverses the table using oolua_pairs.

See Also

oolua_pairs' details for the correct procedure to follow.

14.50.2.7 template<typename T , typename T1 > void OOLUA::Table::try_at (T const & *key,* T1 & *value*)

Function which throws on an error.

Note

This function is only defined when exceptions are enable for the library

Template Parameters

T	Key type
T1	Value type

Parameters

`in`	*key*	
`out`	*value*	

The documentation for this class was generated from the following file:

- oolua_table.h

14.51 TestingReturnOrder Class Reference

Inherits TestFixture.

Public Member Functions

- void luaReturnOrder_luaFunctionWhichReturnsMultipleValuesToCpp_orderFromTopOfStackIsInput2Input1 ()
- void ordering_functionWhichHasAReturnValueAndAlsoReturnsAnInOutParam_topOfStackIsTheInOutParam ()
- void ordering_functionWhichHasAReturnValueAndAlsoReturnsAnInOutParam_slotBeneathTopOfStackIs-FunctionReturn ()

14.51.1 Detailed Description

[CppTraitReturnOrderOneParam] [ProxyTraitReturnOrderOneParam] [ProxyTraitReturnOrderOneParam]

14.51.2 Member Function Documentation

14.51.2.1 void TestingReturnOrder::luaReturnOrder_luaFunctionWhichReturnsMultipleValuesToCpp_orderFromTopOfStackIs-Input2Input1 ( ) [inline]

[TestLuaReturnOrder]

14.51.2.2 void TestingReturnOrder::ordering_functionWhichHasAReturnValueAndAlsoReturnsAnInOutParam_slotBeneathTop-OfStackIsFunctionReturn ( ) [inline]

[TestTraitReturnOrderTop] [TestTraitReturnOrderNextSlot]

14.51.2.3 void TestingReturnOrder::ordering_functionWhichHasAReturnValueAndAlsoReturnsAnInOutParam_topOfStackIsThe-InOutParam ( ) [inline]

[TestLuaReturnOrder] [TestTraitReturnOrderTop]

The documentation for this class was generated from the following file:

- return_order.cpp

14.52 OOLUA::Type_error Struct Reference

Reports that a type pulled from the stack was not the type that was asked for.

`#include <oolua_exception.h>`

Inherits OOLUA::Exception.

14.52.1 Detailed Description

Reports that a type pulled from the stack was not the type that was asked for.

See Also

Error Reporting

Note

Implicit casts such as a derived class to a base class are not type errors

The documentation for this struct was generated from the following file:

- oolua_exception.h

Chapter 15

File Documentation

15.1 dsl_va_args.h File Reference

Provides a lot of the DSL procedures which make use of __VA_ARGS__.

Macros

- #define OOLUA_PROXY(...)

  Starts the generation a proxy class.
- #define OOLUA_MEM_FUNC(...)

  Generates a member function proxy which will also be the named FunctionName.
- #define OOLUA_MEM_FUNC_RENAME(...)

  Generates a member function proxy which will be the named ProxyFunctionName.
- #define OOLUA_MEM_FUNC_CONST(...)

  Generates a constant member function proxy which will also be the named FunctionName.
- #define OOLUA_MEM_FUNC_CONST_RENAME(...)

  Generates a constant member function which will be named ProxyFunctionName.
- #define OOLUA_C_FUNCTION(...)

  Generates a block which will call the C function FunctionName.
- #define OOLUA_MFUNC(...)

  Deduce and generate a proxy for a member function.
- #define OOLUA_MFUNC_CONST(...)

  Deduce and generate a proxy for a constant member function.
- #define OOLUA_CFUNC(...)

  Deduce and generate a proxy for a C function.
- #define OOLUA_SFUNC(...)

  Deduce and generate a proxy for a class static function.
- #define OOLUA_EXPORT_FUNCTIONS(...)

  Exports zero or more member functions which will be registered with Lua.
- #define OOLUA_EXPORT_FUNCTIONS_CONST(...)

  Exports zero or more const member functions which will be registered with Lua.
- #define OOLUA_TAGS(...)

  Allows more information to be specified about the proxy class.
- #define OOLUA_MGET(...)

  Generates a getter, which is a constant function, to retrieve a public instance.
- #define OOLUA_MSET(...)

Generates a setter, which is a none constant function, to set a public instance.

- #define OOLUA_MGET_MSET(...)

  Generates both a getter and a setter for a public instance.

15.1.1 Detailed Description

Provides a lot of the DSL procedures which make use of __VA_ARGS__.

15.2 lua_includes.h File Reference

Prevents name mangling and provides a potential location to enable compatibility when new Lua versions are released.

```
#include "lua/lua.h"
#include "lua/lauxlib.h"
#include "lua/lualib.h"
```

15.2.1 Detailed Description

Prevents name mangling and provides a potential location to enable compatibility when new Lua versions are released. No part of OOLua directly includes any Lua header files, instead when required they include this header. Contrary to what some people may think, this is by design. There is no way to know if a user's version of the Lua library was compiled as C++ or C.

15.3 lvd_type_traits.h File Reference

Template struct which report if the type has qualifiers and also removes some of the possible qualifiers.

15.3.1 Detailed Description

Template struct which report if the type has qualifiers and also removes some of the possible qualifiers.

15.4 lvd_types.h File Reference

Cross platform integral sized types.

```
#include "platform_check.h"
#include "type_list.h"
```

15.4.1 Detailed Description

Cross platform integral sized types.

15.5 only_for_doxygen.h File Reference

This file is not part of OOLua, the only reason for it is to allow doxygen to document some things which otherwise it can not do.

Macros

- #define OOLUA_NEW_POINTER_DEFAULT_IS_SHARED_TYPE

  ***Default:** Disabled*
- #define OOLUA_SHARED_HEADER

  ***Default:** MSC: <memory> other compilers: <tr1/memory>*
- #define OOLUA_SHARED_TYPE

  ***Default:** std::tr1::shared_ptr*
- #define OOLUA_SHARED_CONST_CAST

  ***Default:** std::tr1::const_pointer_cast*

Typedefs

- typedef int(∗ lua_CFunction)(lua_State ∗vm)

  Lua's C function signature.

15.5.1 Detailed Description

This file is not part of OOLua, the only reason for it is to allow doxygen to document some things which otherwise it can not do.

15.5.2 Typedef Documentation

15.5.2.1 typedef int(∗ lua_CFunction)(lua_State ∗vm)

Lua's C function signature.

This is a Lua type which is the required signature to bind C functions to Lua.

Parameters

in	*vm*	The virtual machine for which a function will operate on

Returns

Number of function returns to Lua

15.6 oolua.h File Reference

Kitchen sink header file for Object Oriented Lua. Which could be a good candidate for a PCH.

```
#include "lua_includes.h"
#include "oolua_dsl.h"
#include "proxy_function_exports.h"
#include "oolua_version.h"
#include "oolua_error.h"
#include "oolua_stack.h"
#include "oolua_script.h"
#include "oolua_open.h"
#include "oolua_chunk.h"
#include "oolua_registration.h"
#include "oolua_table.h"
#include "oolua_ref.h"
#include "oolua_helpers.h"
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

Functions

- template<typename T >
  bool OOLUA::set_global (lua_State ∗vm, char const ∗name, T &instance)

  Helper function to set a Lua global variable.
- bool OOLUA::set_global (lua_State ∗vm, char const ∗name, lua_CFunction instance)

  None template version.
- void OOLUA::set_global_to_nil (lua_State ∗vm, char const ∗name)

  Helper function to set a Lua global variable to nil.
- template<typename T >
  bool OOLUA::get_global (lua_State ∗vm, char const ∗name, T &instance)

  Helper function to set a Lua global variable.

15.6.1 Detailed Description

Kitchen sink header file for Object Oriented Lua. Which could be a good candidate for a PCH.

15.7 oolua_amalgamation.lua File Reference

Lua module for amalgamating the library's headers and source files into one header and source file.

–]]

Functions

- function amalgamate (include_dir, src_dir, output_dir)

  Generates an amalgamated header and source file for the library.

15.7.1 Detailed Description

Lua module for amalgamating the library's headers and source files into one header and source file.

–]]

–[[

15.8 oolua_boilerplate.h File Reference

15.8.1 Detailed Description

Date

Thu Apr 10 18:41:11 2014

Configurable values as set when generating this file

- constructor_params 5 - Maximum amount of parameters for a constructor of a proxied type (Default 5)
- lua_params 10 - Maximum amount of parameters for a call to a Lua function (Default 10)
- cpp_params 8 - Maximum number of parameters a C++ function can have (Default 8)

  Note

  Warning this file was generated, edits to the file will not persist if it is regenerated.

15.9 oolua_chunk.h File Reference

Provides methods for loading and running chunks.

```
#include <string>
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

Functions

- bool OOLUA::load_chunk (lua_State ∗vm, std::string const &chunk)

  Loads a chunk leaving the resulting function on the stack.
- bool OOLUA::run_chunk (lua_State ∗vm, std::string const &chunk)

  Loads and runs a chunk of code.
- bool OOLUA::load_file (lua_State ∗vm, std::string const &filename)

  Loads a file leaving the resulting function on the stack.
- bool OOLUA::run_file (lua_State ∗vm, std::string const &filename)

  Loads and runs the file.

15.9.1 Detailed Description

Provides methods for loading and running chunks.

15.10 oolua_config.h File Reference

Configuration options for the OOLua library.

Macros

- #define OOLUA_USE_EXCEPTIONS

  Default: *Disabled*
- #define OOLUA_STORE_LAST_ERROR

Default: *Enabled*
- #define OOLUA_RUNTIME_CHECKS_ENABLED

  Default: *Enabled*
- #define OOLUA_CHECK_EVERY_USERDATA_IS_CREATED_BY_OOLUA

  Default: *Enabled*
- #define OOLUA_USERDATA_OPTIMISATION

  Default: *Enabled*
- #define OOLUA_DEBUG_CHECKS

  Default: *Enabled when DEBUG or _DEBUG is defined*
- #define OOLUA_SANDBOX

  Default: *Disabled*
- #define OOLUA_STD_STRING_IS_INTEGRAL

  Default: *Enabled*
- #define OOLUA_USE_SHARED_PTR

  Default: *Disabled*

15.10.1 Detailed Description

Configuration options for the OOLua library.

15.11 oolua_dsl.h File Reference

Header which provides only what is needed for a class to be proxied using the DSL.

```
#include "dsl_va_args.h"
#include "proxy_class.h"
#include "proxy_constructor.h"
#include "proxy_member_function.h"
#include "proxy_none_member_function.h"
#include "proxy_public_member.h"
#include "proxy_tags.h"
#include "default_trait_caller.h"
#include "oolua_stack_fwd.h"
#include "oolua_traits.h"
```

15.11.1 Detailed Description

Header which provides only what is needed for a class to be proxied using the DSL.

15.12 oolua_dsl_export.h File Reference

Header to be used in conjunction with oolua_dsl.h when exporting proxies using the DSL.

```
#include "proxy_function_exports.h"
#include "oolua_stack.h"
```

15.12.1 Detailed Description

Header to be used in conjunction with oolua_dsl.h when exporting proxies using the DSL.

15.13 oolua_error.h File Reference

Generic header to be included when handling errors.

```
#include "oolua_config.h"
#include <string>
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

Functions

- void OOLUA::reset_error_value (lua_State ∗vm)

  Reset the error state such that a call to OOLUA::get_last_error will return an empty string.
- std::string OOLUA::get_last_error (lua_State ∗vm)

  Returns the last stored error.

15.13.1 Detailed Description

Generic header to be included when handling errors. When the library is compiled with OOLUA_USE_EXCEPTIONS == 1 it will include the oolua_exception.h header and provide dummy implementations for OOLUA::get_last_error and OOLUA::reset_error_value. When compiled with OOLUA_STORE_LAST_ERROR == 1 it provides implementations for OOLUA::get_last_error and OOLUA::reset_error_value.

See Also

Library Configuration

15.14 oolua_exception.h File Reference

Declares the exceptions which are used by OOLua when OOLUA_USE_EXCEPTIONS is set to one.

```
#include "oolua_config.h"
```

15.14.1 Detailed Description

Declares the exceptions which are used by OOLua when OOLUA_USE_EXCEPTIONS is set to one.

See Also

Library Configuration
Exception classes

15.15 oolua_function.h File Reference

Provides the class OOLUA::Lua_function which is a helper for calling Lua functions.

```
#include "lua_includes.h"
#include "oolua_stack_fwd.h"
#include "oolua_ref.h"
#include "oolua_boilerplate.h"
#include <string>
```

Classes

- struct OOLUA::Lua_function

  Structure which is used to call a Lua function.

Namespaces

- OOLUA

  This is the root namespace of the Library.

15.15.1 Detailed Description

Provides the class OOLUA::Lua_function which is a helper for calling Lua functions.

15.16 oolua_generate.lua File Reference

Lua module for generating required OOLua configurable boilerplate code.

–]]

```
#include "oolua_config.h"
```

Functions

- function defaults ()

  Gets the default options as key(string) and value(number) entries in a table.
- function gen (options, path)

  Generate boilerplate header files.
- function default_details ()

  Returns the library defaults and details.

15.16.1 Detailed Description

Lua module for generating required OOLua configurable boilerplate code.

–]]

–[[

15.17 oolua_helpers.h File Reference

Provides an index equal function which is multi Lua version compatible and a Lua Universe checking function.

Namespaces

- OOLUA

  This is the root namespace of the Library.

Functions

- bool OOLUA::idxs_equal (lua_State ∗vm, int idx0, int idx1)
- bool OOLUA::can_xmove (lua_State ∗vm0, lua_State ∗vm1)

  Uses the Lua C API to check if it is valid to move data between the states.

15.17.1 Detailed Description

Provides an index equal function which is multi Lua version compatible and a Lua Universe checking function.

15.18 oolua_open.h File Reference

Sets up the a Lua Universe to work with the library.

Namespaces

- OOLUA

  This is the root namespace of the Library.

Functions

- void OOLUA::setup_user_lua_state (lua_State ∗vm)

  Sets up a lua_State to work with OOLua.

15.18.1 Detailed Description

Sets up the a Lua Universe to work with the library.

15.19 oolua_pull.h File Reference

Implements the Lua stack operation OOLUA::pull.

```
#include "lua_includes.h"
#include "oolua_config.h"
#include "oolua_stack_fwd.h"
#include "oolua_traits_fwd.h"
#include "oolua_string.h"
#include "lvd_types.h"
#include "lvd_type_traits.h"
#include <cassert>
```

Classes

- class OOLUA::Proxy_class< T >

  A template wrapper for class objects of type T used by the script binding.

Namespaces

- OOLUA

  This is the root namespace of the Library.

Functions

- template<typename T >
  bool OOLUA::pull (lua_State *const vm, T &value)

  Pulls the top element off the stack and pops it.
- template<typename T >
  bool OOLUA::pull (lua_State *const vm, T *&value)

  Pulls the top element off the stack and pops it.

15.19.1 Detailed Description

Implements the Lua stack operation OOLUA::pull.

15.20 oolua_push.h File Reference

Implements the Lua stack operation OOLUA::pull.

```
#include "lua_includes.h"
#include "oolua_stack_fwd.h"
#include "oolua_traits_fwd.h"
#include "oolua_string.h"
#include "oolua_config.h"
#include "lvd_types.h"
#include "lvd_type_traits.h"
#include <cassert>
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

Functions

- template<typename T >
  bool OOLUA::push (lua_State *const vm, T const &value)

  Pushes an instance to top of the Lua stack.
- template<typename T >
  bool OOLUA::push (lua_State *const vm, OOLUA::lua_acquire_ptr< T > const &value)

  Pushes an instance to top of the Lua stack.

- template<typename T >
  bool OOLUA::push (lua_State ∗const vm, T ∗const &value)

  Pushes an instance to top of the Lua stack.

15.20.1 Detailed Description

Implements the Lua stack operation OOLUA::pull.

15.21 oolua_registration.h File Reference

Implements the public API register functions and internal workers.

```
#include "lua_includes.h"
#include "proxy_class.h"
#include "proxy_userdata.h"
#include "proxy_operators.h"
#include "proxy_function_dispatch.h"
#include "proxy_storage.h"
#include "proxy_tags.h"
#include "proxy_tag_info.h"
#include "proxy_base_checker.h"
#include "class_from_stack.h"
#include "push_pointer_internal.h"
#include "oolua_table.h"
#include "oolua_config.h"
#include "char_arrays.h"
#include "lvd_types.h"
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

Functions

- template<typename T >
  void OOLUA::register_class (lua_State ∗vm)

  Registers the class type T and it's bases with an instance of lua_State.

- template<typename T , typename K , typename V >
  void OOLUA::register_class_static (lua_State ∗const vm, K const &key, V const &value)

  Registers a key K and value V entry into class T.

15.21.1 Detailed Description

Implements the public API register functions and internal workers.

Copyright

15.22 oolua_registration_fwd.h File Reference

Forward declarations of public API functions used for registering a class or statics for a class type.

Namespaces

- OOLUA

  This is the root namespace of the Library.

Functions

- template<typename T >
  void OOLUA::register_class (lua_State ∗vm)

  Registers the class type T and it's bases with an instance of lua_State.
- template<typename T , typename K , typename V >
  void OOLUA::register_class_static (lua_State ∗const vm, K const &key, V const &value)

  Registers a key K and value V entry into class T.

15.22.1 Detailed Description

Forward declarations of public API functions used for registering a class or statics for a class type.

15.23 oolua_script.h File Reference

Provides the helper class OOLUA::Script.

```
#include "lua_includes.h"
#include "oolua_stack_fwd.h"
#include "oolua_registration_fwd.h"
#include "oolua_function.h"
#include <string>
```

Classes

- class OOLUA::Script

    OOLua helper class.

Namespaces

- OOLUA

    This is the root namespace of the Library.

15.23.1 Detailed Description

Provides the helper class OOLUA::Script.

15.24 oolua_stack.h File Reference

Makes available implementations for the stack operations OOLUA::push and OOLUA::pull, which have forward declarations in oolua_stack_fwd.h.

```
#include "oolua_stack_fwd.h"
#include "oolua_push.h"
#include "oolua_pull.h"
#include "stack_get.h"
```

15.24.1 Detailed Description

Makes available implementations for the stack operations OOLUA::push and OOLUA::pull, which have forward declarations in oolua_stack_fwd.h.

15.25 oolua_stack_fwd.h File Reference

Forward declarations of the push and pull methods, which provide simple interaction with the Lua stack.

```
#include "oolua_traits_fwd.h"
```

Classes

- struct OOLUA::Lua_ref< ID >

    A typed wrapper for a Lua reference.

Namespaces

- OOLUA

    This is the root namespace of the Library.

Functions

- bool OOLUA::push (lua_State ∗const vm, void ∗lightud)

  Pushes an instance to top of the Lua stack.
- bool OOLUA::push (lua_State ∗const vm, bool const &value)

  Pushes an instance to top of the Lua stack.
- bool OOLUA::push (lua_State ∗const vm, char ∗const &value)

  Pushes an instance to top of the Lua stack.
- bool OOLUA::push (lua_State ∗const vm, char const ∗const &value)

  Pushes an instance to top of the Lua stack.
- bool OOLUA::push (lua_State ∗const vm, double const &value)

  Pushes an instance to top of the Lua stack.
- bool OOLUA::push (lua_State ∗const vm, float const &value)

  Pushes an instance to top of the Lua stack.
- bool OOLUA::push (lua_State ∗const vm, oolua_CFunction const &value)

  Pushes an instance to top of the Lua stack.
- bool OOLUA::push (lua_State ∗const vm, Table const &value)

  Pushes an instance to top of the Lua stack.
- template<typename T >
  bool OOLUA::push (lua_State ∗const vm, T ∗const &value)

  Pushes an instance to top of the Lua stack.
- template<typename T >
  bool OOLUA::push (lua_State ∗const vm, OOLUA::lua_acquire_ptr< T > const &value)

  Pushes an instance to top of the Lua stack.
- template<typename T >
  bool OOLUA::push (lua_State ∗const vm, T const &value)

  Pushes an instance to top of the Lua stack.

- bool OOLUA::pull (lua_State ∗const vm, void ∗&lightud)

  Pulls the top element off the stack and pops it.
- bool OOLUA::pull (lua_State ∗const vm, bool &value)

  Pulls the top element off the stack and pops it.
- bool OOLUA::pull (lua_State ∗const vm, double &value)

  Pulls the top element off the stack and pops it.
- bool OOLUA::pull (lua_State ∗const vm, float &value)

  Pulls the top element off the stack and pops it.
- bool OOLUA::pull (lua_State ∗const vm, oolua_CFunction &value)

  Pulls the top element off the stack and pops it.
- bool OOLUA::pull (lua_State ∗const vm, Table &value)

  Pulls the top element off the stack and pops it.
- template<typename T >
  bool OOLUA::pull (lua_State ∗const vm, T ∗&value)

  Pulls the top element off the stack and pops it.
- template<typename T >
  bool OOLUA::pull (lua_State ∗const vm, T &value)

  Pulls the top element off the stack and pops it.
- template<typename T >
  bool OOLUA::pull (lua_State ∗const vm, cpp_acquire_ptr< T > const &value)

  Pulls the top element off the stack and pops it.

15.25.1 Detailed Description

Forward declarations of the push and pull methods, which provide simple interaction with the Lua stack.

15.26 oolua_string.h File Reference

Provides a method of not including the string header in DSL header files and allows other string types to be easily integrated.

```
#include "lvd_type_traits.h"
#include "proxy_test.h"
```

Namespaces

- OOLUA

  This is the root namespace of the Library.
- OOLUA::STRING

  Defines which type of string classes can be pulled and pushed from the stack with the public API and the DSL.

Macros

- #define OOLUA_CLASS_OR_BASE_CONTAINS_METHOD(StructName, MethodSignature, MethodName)

  Creates a structure that enables checking a class type for a specific function signature that has a specific name.

Functions

- template<typename StringType >
  bool OOLUA::STRING::push (lua_State ∗const vm, StringType const &value)

  Function to which public API calls resolve to.
- template<typename StringType >
  bool OOLUA::STRING::pull (lua_State ∗const vm, StringType &value)

  Function to which public API calls resolve to.
- template<typename StringType >
  void OOLUA::STRING::get (lua_State ∗const vm, int idx, StringType &value)

  Internal function used by the DSL to retrieve a string from the stack.

15.26.1 Detailed Description

Provides a method of not including the string header in DSL header files and allows other string types to be easily integrated.

15.27 oolua_table.h File Reference

Interface for the Lua unordered and ordered associative container.

```
#include "lua_includes.h"
#include <string>
#include "oolua_stack_fwd.h"
#include "oolua_ref.h"
#include "oolua_config.h"
#include "oolua_error.h"
```

Classes

- class OOLUA::Table

  Wrapper around a table in Lua which allows easy usage.

Namespaces

- OOLUA

  This is the root namespace of the Library.

Macros

- #define oolua_ipairs(table)

  Helper for iterating over the sequence part of a table.
- #define oolua_ipairs_end()
- #define oolua_pairs(table)

  Helper for iterating over a table.
- #define oolua_pairs_end()

Functions

- template<typename T , typename T1 >
  void OOLUA::table_set_value (lua_State ∗vm, int table_index, T const &key, T1 const &value)

  The table is at table_index which can be either absolute or pseudo in the stack table is left at the index.
- template<typename T , typename T1 >
  bool OOLUA::table_at (lua_State ∗vm, int const table_index, T const &key, T1 &value)

  The table is at table_index which can be either absolute or pseudo in the stack table is left at the index.
- void OOLUA::new_table (lua_State ∗vm, OOLUA::Table &t)

  Creates a new valid OOLUA::Table.
- OOLUA::Table OOLUA::new_table (lua_State ∗vm)

  Creates a new valid Table.

15.27.1 Detailed Description

Interface for the Lua unordered and ordered associative container.

15.27.2 Macro Definition Documentation

15.27.2.1 #define oolua_ipairs(*table*)

Helper for iterating over the sequence part of a table.

Parameters

table	

Declares:

- int _i_index_ : Current index into the array
- int const _oolua_array_index_ : Stack index at which table is located
- lua_State∗ lvm : The vm associated with the table

Note

Returning from inside of the loop will not leave the stack clean unless you reset it. usage:

```
oolua_ipairs(table)
{
    if(_i_index_ == 99)
    {
        lua_settop(lvm,_oolua_array_index-1);
        return "red balloons";
    }
}
oolua_ipairs_end()
return "Not enough balloons to go bang."
```

15.27.2.2 #define oolua_ipairs_end( )

See Also

oolua_ipairs

15.27.2.3 #define oolua_pairs(*table*)

Helper for iterating over a table.

Parameters

table	

When iterating over a table, for the next iteration to work you must leave the key on the top of the stack. If you need to work with the key, it is a good idea to use lua_pushvalue to duplicate it on the stack. This is because if the type is not a string and you retrieve a string from the stack with lua_tostring, this will alter the vm's stack entry.

Declares:

- int const _oolua_table_index_ : Stack index at which table is located
- lua_State∗ lvm : The vm associated with the table

usage:

```
oolua_pairs(table)
{
    \\do what ever
    lua_pop(vm, 1);\\Pop the value, leaving the key at the top of stack
}
oolua_pairs_end()
```

15.27.2.4 #define oolua_pairs_end( )

See Also

oolua_pairs

15.28 oolua_traits_fwd.h File Reference

Forward declarations of Traits.

Classes

- struct OOLUA::in_p< T >

  Input parameter trait.
- struct OOLUA::out_p< T >

  Output parameter trait.
- struct OOLUA::in_out_p< T >

  Input and output parameter trait.
- struct OOLUA::lua_out_p< T >

  Output parameter trait which will be owned by Lua.
- struct OOLUA::light_p< T >

  Input parameter trait.
- struct OOLUA::light_return< T >

  Return trait for a light userdata type.
- struct OOLUA::lua_return< T >

  Return trait for a type which will be owned by Lua.
- struct OOLUA::shared_return< T >

  Converts a raw pointer return type to the supported shared pointer type.
- struct OOLUA::maybe_null< T >

  Return trait for a pointer which at runtime maybe NULL.
- struct OOLUA::cpp_acquire_ptr< T >

  Change of ownership to C++.
- struct OOLUA::lua_acquire_ptr< T >

  Change of ownership to Lua.

Namespaces

- OOLUA

  This is the root namespace of the Library.

Enumerations

- enum OOLUA::Owner { OOLUA::No_change, OOLUA::Cpp, OOLUA::Lua }

15.28.1 Detailed Description

Forward declarations of Traits.

15.29 oolua_version.h File Reference

OOLua library version information for both the CPP and at run time.

Namespaces

- OOLUA

    This is the root namespace of the Library.

Macros

- #define OOLUA_VERSION_MAJ 2

    CPP major version number.
- #define OOLUA_VERSION_MIN 0

    CPP minor version number.
- #define OOLUA_VERSION_PATCH 1

    CPP patch version number.
- #define OOLUA_VERSION

    CPP string detailing the library version.

Variables

- static const char OOLUA::version_str [] = OOLUA_STRINGISE(OOLUA_VERSION_MAJ) "." OOLUA_STRINGISE(OOLUA_VERSION_MIN) "." OOLUA_STRINGISE(OOLUA_VERSION_PATCH)

    OOLua version string.
- static const int OOLUA::version_number = 2∗10000+0∗1000+1

    OOLua version int.

15.29.1 Detailed Description

OOLua library version information for both the CPP and at run time.

15.30 platform_check.h File Reference

15.30.1 Detailed Description

Preforms a check of platform defines and defines a macro

Remarks

Information available via `http://predef.sourceforge.net/preos.html`

15.31 proxy_base_checker.h File Reference

Checks the hierarchical bases to ensure a cast is defined.

```
#include "type_list.h"
#include "proxy_userdata.h"
#include "proxy_class.h"
#include "oolua_config.h"
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

15.31.1 Detailed Description

Checks the hierarchical bases to ensure a cast is defined. Walks a list of bases class defined in a OOLUA::Proxy_-class to find if a type can be converted to the requested type, if it is valid then the procedures will preform the cast.

15.32 proxy_caller.h File Reference

Provides implementations which actually call the member or stand alone function, it also pushes a function return to the stack if the fubction has one.

```
#include "oolua_boilerplate.h"
#include "oolua_traits_fwd.h"
#include "type_converters.h"
#include "proxy_stack_helper.h"
#include "lua_includes.h"
#include "oolua_config.h"
```

15.32.1 Detailed Description

Provides implementations which actually call the member or stand alone function, it also pushes a function return to the stack if the fubction has one.

15.33 proxy_class.h File Reference

Defines OOLUA::Proxy_class, it's bases in the hierarchical tree and internal details.

```
#include "type_list.h"
```

Classes

- class OOLUA::Proxy_class< T >

  A template wrapper for class objects of type T used by the script binding.

Namespaces

- OOLUA

  This is the root namespace of the Library.

Macros

- #define OOLUA_PROXY_END

  Ends the generation of the proxy class.
- #define OOLUA_SCOPED_ENUM(Name, Entry)

Creates a entry into a OOLUA_ENUMS block for a C++11 scoped enum.

- #define OOLUA_ENUM(EnumName)

  Creates a entry into a OOLUA_ENUMS block.

- #define OOLUA_ENUMS(EnumEntriesList)

  Creates a block into which enumerators can be defined with OOLUA_ENUM or OOLUA_SCOPED_ENUM.

15.33.1 Detailed Description

Defines OOLUA::Proxy_class, it's bases in the hierarchical tree and internal details. Defines the class, its bases in the hierarchical tree. The classes name an array used to hold the functions its make available to the script and C++ special member functions

15.34 proxy_constructor.h File Reference

Implements Proxy_class constructor handlers and the constructor block.

```
#include "lua_includes.h"
#include "oolua_traits_fwd.h"
#include "proxy_tags.h"
#include "proxy_storage.h"
#include "proxy_tag_info.h"
#include "proxy_userdata.h"
#include "proxy_stack_helper.h"
#include "proxy_constructor_param_tester.h"
#include "type_converters.h"
#include "oolua_boilerplate.h"
```

Macros

- #define OOLUA_CTOR(...)

  Generates a constructor in a constructor block.

- #define OOLUA_CTORS(ConstructorEntriesList)

  Creates a block into which none default constructors can be defined using OOLUA_CTOR.

15.34.1 Detailed Description

Implements Proxy_class constructor handlers and the constructor block.

15.35 proxy_constructor_param_tester.h File Reference

Helps test that a constructor parameter is of the requested type so that a matching constructor can be called.

```
#include "proxy_userdata.h"
#include "lua_includes.h"
#include "class_from_stack.h"
#include "oolua_config.h"
#include "type_list.h"
#include "oolua_string.h"
#include "oolua_traits_fwd.h"
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

15.35.1 Detailed Description

Helps test that a constructor parameter is of the requested type so that a matching constructor can be called.

15.36 proxy_function_dispatch.h File Reference

Provides the templated functions which are bound to Lua matching the lua_CFunction signature, which dispatch control to the correct Proxy_class functions.

```
#include "lua_includes.h"
#include "proxy_class.h"
#include "class_from_stack.h"
#include "oolua_config.h"
```

15.36.1 Detailed Description

Provides the templated functions which are bound to Lua matching the lua_CFunction signature, which dispatch control to the correct Proxy_class functions.

15.37 proxy_function_exports.h File Reference

Macros

- #define OOLUA_EXPORT_NO_FUNCTIONS(Class)

  Inform that there are no functions of interest.

15.37.1 Detailed Description

Date

Thu Apr 10 18:41:11 2014

Configurable values as set when generating this file

- class_functions 15 - Maximum amount of class functions that can be registered for each proxied type (Default 15)

  Note

  Warning this file was generated, edits to the file will not persist if it is regenerated.

15.38 proxy_member_function.h File Reference

Internal macros which generate proxy member functions.

```
#include "oolua_traits_fwd.h"
#include "oolua_boilerplate.h"
#include "proxy_caller.h"
#include "default_trait_caller.h"
#include <cassert>
#include "oolua_config.h"
```

15.38.1 Detailed Description

Internal macros which generate proxy member functions.

15.39 proxy_none_member_function.h File Reference

Contains internal macros for proxing none member functions.

```
#include "oolua_traits_fwd.h"
#include "oolua_boilerplate.h"
#include "proxy_caller.h"
#include "default_trait_caller.h"
#include "oolua_config.h"
```

15.39.1 Detailed Description

Contains internal macros for proxing none member functions.

15.40 proxy_operators.h File Reference

Internal implemenations of Proxy_class operators.

```
#include "lua_includes.h"
#include "proxy_userdata.h"
#include "proxy_storage.h"
#include "oolua_stack_fwd.h"
#include "oolua_traits_fwd.h"
#include "push_pointer_internal.h"
#include "type_list.h"
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

15.40.1 Detailed Description

Internal implemenations of Proxy_class operators. Defines operators which will be made available in scripts when a OOLUA::Proxy_class contains operator tags

15.41 proxy_public_member.h File Reference

Proxies a class public member variable.

```
#include "oolua_stack_fwd.h"
#include "proxy_test.h"
#include "lvd_type_traits.h"
```

15.41.1 Detailed Description

Proxies a class public member variable.

15.42 proxy_stack_helper.h File Reference

Helpers for the DSL which are allowed to do things the Public API is not.

```
#include "lua_includes.h"
#include "oolua_stack_fwd.h"
#include "oolua_string.h"
#include <cassert>
#include "push_pointer_internal.h"
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

15.42.1 Detailed Description

Helpers for the DSL which are allowed to do things the Public API is not.

15.43 proxy_tags.h File Reference

Possible members for the Proxy_class Tag block.

Classes

- struct OOLUA::Abstract

  The class being mirrored is an abstract class.
- struct OOLUA::Less_op

  Less than operator is defined for the type.
- struct OOLUA::Equal_op

  Equal operator is defined for the type.
- struct OOLUA::Not_equal_op

  Not equal operator is defined for the type.
- struct OOLUA::Less_equal_op

  Less than or equal operator is defined for the type.
- struct OOLUA::Div_op

Division operator is defined for the type.

- struct OOLUA::Mul_op

  Multiplication operator is defined for the type.

- struct OOLUA::Sub_op

  Subtraction operator is defined for the type.

- struct OOLUA::Add_op

  Addition operator is defined for the type.

- struct OOLUA::No_default_constructor

  There is not a default constructor in the public interface yet there are other constructors.

- struct OOLUA::No_public_constructors

  There are no constructors in the public interface.

- struct OOLUA::No_public_destructor

  There is not a destructor in the public interface and OOLua will not attempt to delete an instance of this type.

- struct OOLUA::Register_class_enums

  The class has enums to register.

- struct OOLUA::Shared

  Overrides the configuration behaviour when creating proxied types.

- struct OOLUA::No_shared

  Overrides the configuration behaviour when creating proxied types.

Namespaces

- OOLUA

  This is the root namespace of the Library.

15.43.1 Detailed Description

Possible members for the Proxy_class Tag block.

15.44 proxy_userdata.h File Reference

Contains the internal userdata type used by OOLua to represent C++ class types, also contains inlined functions for checking and setting flags in the userdata.

```
#include "oolua_config.h"
#include "lvd_types.h"
```

Namespaces

- OOLUA

  This is the root namespace of the Library.

15.44.1 Detailed Description

Contains the internal userdata type used by OOLua to represent C++ class types, also contains inlined functions for checking and setting flags in the userdata.

15.45 type_list.h File Reference

Loki Type_list from Andrei Alexandrescu's book Modern C++ Design.

```
#include "typelist_structs.h"
```

15.45.1 Detailed Description

Loki Type_list from Andrei Alexandrescu's book Modern C++ Design.

Copyright

15.46 typelist_structs.h File Reference

15.46.1 Detailed Description

Remarks

This file was auto generated

Index