Высокоуровневая библиотека для работы с Lua в Delphi

CrystalLUA

ver. 11/09/2012

Оглавление

Введение

Рад представить вашему вниманию труд, которым я горжусь особенно. CrystalLUA - это новое слово в связывании Delphi приложений и скриптов на популярном языке Lua. Для того чтобы оценить мощь и простоту библиотеки, достаточно следующего примера. Серьёзно, возьмите lua.dll, добавьте CrystalLUA в **uses** и выполните этот код. Результат никого не оставит равнодушным ;)

```
procedure TForm1.Button1Click(Sender: TObject);
const
  SCRIPT_STRING =
  'Form1 {Caption="Text", Position=poScreenCenter, Color=0x007F7F7F}';
var
  Lua: TLua;
begin
  Lua := TLua.Create;
  Lua.RegClass(TForm1);
  Lua.RegVariable('Form1', Form1, typeinfo(TForm1));
  Lua.RunScript(SCRIPT_STRING);
  Lua.Free;
end;
```

Три слова, которыми я могу охарактеризовать CrystalLUA - стабильность, простота, производительность. Множество внутренних механизмов CrystalLUA обеспечат грамотное приведение типов и максимально точно определят ошибку, если она была допущена. Широкие возможности RTTI Delphi позволяют избавить пользователя от бесконечной рутины регистрации. Ряд тщательно взвешенных решений по оптимизации позволяет выжать из тандема Delphi + Lua максимум производительности.

А началось всё несколько лет назад, когда появилась острая необходимость связать код программы на языке Delphi со скриптовым кодом на языке Lua. Не мне вам рассказывать, что подобные средства для связи уже существуют. Однако те решения, которые видел я - не устраивали меня низким уровнем, низкой производительностью или всем сразу. А так как я имею достаточный опыт в разработке удобных быстродействующих средств, CrystalLUA стал моим основным проектом на несколько месяцев). Кстати, написание "LUA" в названии проекта не случайно. CrystalLUA позиционируется как надстройка над стандартными средствами Lua. В чём именно состоят отличия - вы увидите позже. Кстати файл документации слегка отстаёт, но я обещаю переработать его, дополнить и перевести на английский (чтобы о библиотеке знали не только русскоговорящие разработчики).

Сегодня в CrystalLUA вы можете следующее:

- вообще забыть о том, что такое Lua API и виртуальный стек
- загружать скриптовые файлы, вызывать Lua-функции
- регистрировать нативные глобальные переменные, константы, методы, получать значения глобальных объектов Lua [общее глобальное пространство]
- классы: конструкторы, свойства, методы

- структуры(record, object): конструкторы, поля, методы
- массивы: как статические, так и динамические
- множества(set) и перечисляемые типы(enumerate)
- иметь доступ к Lua-таблицам
- использовать Lua-ссылки

CrystalLUA поддерживает все известные мне типы: Boolean,ByteBool,WordBool, LongBool, Shortint, Byte, Smallint, Word, Longint(Integer), Longword(Dword, Cardinal), Int64, Single, Double, Extended, Comp, Currency, Char, WideChar, Enumeration, String, WideString, Variant, Pointer, TClass, Interface, экземпляры класса, структуры, массивы, множества, события.

Но прежде, чем я начну приводить саму документацию по библиотеке, я хотел бы поговорить о минусах. Минусов мало и они спорные, но я считаю своим долгом известить вас о них прежде, чем вы начнёте использовать CrystalLUA в своих проектах. Наверное основной минус библиотеки – это её размер. Библиотека давно перешагнула рубеж в 10 000 строк кода, она использует стандартные модули SysUtils, Variants, TypInfo, Classes и добавляет от 50 до 200 килобайт веса приложению в зависимости от используемости стандартных модулей и библиотеки. Второй “минус” – не идеальная производительность. Все мы должны понимать, что использование Lua API напрямую чисто теоретически даёт лучшую производительность, нежели при использовании универсальной библиотеки. С другой стороны, все используют универсальные решения потому, что универсальные подходы минимизируют рутину. Так вот CrystalLUA как универсальная система скорее всего работает быстрее аналогичных систем. Связано это с тщательно продуманными и реализованными схемами взаимодействия Lua с нативным кодом, умным менеджментом памяти, ассемблерной реализацией в узких местах. Существует мнение, что такой скрупулёзный подход к оптимизации не оправдан в случае со скриптами. Не знаю. Мне греет душу мысль, что я выжал максимум производительности.

На сегодняшний день библиотека работает под Ansi версии Delphi (т.е. <= 2007), однако как раз в данный момент ведётся проектирование и разработка поддержки Unicode. Кто знает, возможно в будущем появится поддержка FreePascal, C++ Builder, платформ Linux, MacOS, iOS, x64 и др. Дело в том, что проект делается на полном энтузиазме и лично мне в данный момент поддержка других платформ не нужна. Однако если ты заинтересован в какой-либо доработке и твёрд в своих намерениях - можешь обращаться - всё решаемо.

Хочу поблагодарить людей, которые советом и делом помогли создать CrystalLUA. Конечно их вклад не слишком велик, но без их помощи библиотека не была бы в том виде, котором вы её видите сейчас. Благодарности для **RPGman**, **@!!ex**, **GrayFace**.

Обсуждения библиотеки проходят здесь: http://www.gamedev.ru/projects/forum/?id=140784

Если вы обнаружили ошибку – у меня огромная просьба сообщать о ней в личке. По опыту прошлых лет, сообщения об ошибках в ветке наводят неприятные панические настроения. Поэтому давайте уважать других участников ветки, и если вы обнаружили ошибку - сообщайте мне - обновления с коррективами выйдут достаточно оперативно.

Связаться со мной вы можете так же по ICQ: 250481638 или email: softforyou@inbox.ru

Текущую версию качайте здесь: http://devilhome.narod.ru/crystal_lua.zip .

Быстрый старт

Для работы с Lua вам понадобятся 2 файла (оба находятся в архиве):
- CrystalLUA.pas
- lua.dll (версия 5.1)

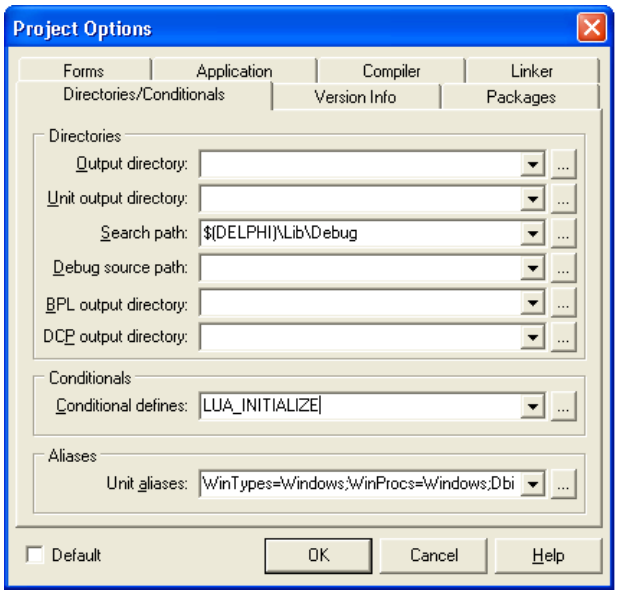
Инициализация

Для инициализации Lua вы можете поступить двумя способами:

```
// вариант 1
var
  Lua: TLua;
begin
  Lua := CreateLua;
  // возвращает экземпляр или nil в случае ненахождения dll
  // или отсутствия нужных функций
  . . .
  Lua.Free;
end;

// вариант 2
var
  Lua: TLua;
begin
  Lua := TLua.Create;
  // вызывает Exception в случае ненахождения dll
  // или отсутствия нужных функций
  . . .
  Lua.Free;
end;
```

Автоматическая инициализация

Если выставить Define “LUA_INITIALIZE”, то экземпляр Lua: TLua будет создаваться и удаляться автоматически. Если Lua = **nil**, то инициализация прошла некорректно



Фрагменты кода CrystalLUA.pas

```
function LuaClassProc(const Proc: TLuaClassPro
function LuaClassProc(const Proc: TLuaClassPro
function LuaClassProc(const Proc: TLuaClassPro

{$ifdef LUA_INITIALIZE}
var
  Lua: TLua;
{$endif}

implementation


initialization
  {$ifdef LUA_INITIALIZE} Lua := CreateLua(); {$en

finalization
  {$ifdef LUA_INITIALIZE} FreeAndNil(Lua); {$endif

end.
```

Загрузка и выполнение кода Lua

Загрузить и выполнить код вы можете тремя способами:

1) Из файла:
```
Lua.LoadScript('game_objects.lua');
```

2) Из буфера памяти:
```
procedure LoadScript(const ScriptBuffer: pointer;
          const ScriptBufferSize: integer; const ChunkName: string='');
```

3) Из строки
```
Lua.RunScript(Memo1.Lines.Text);
```

Работа с Lua на низком уровне

Кто работал с Lua знает, что там повсюду используется lua_State. TLua.Handle – это и есть lua_State. Можно использовать стандартные заголовки для lua.dll или подгружать функции динамически. Например так:

```
@lua_pcall := TLua.GetProcAddress('lua_pcall', true);
```

Изменение глобальных переменных

Изменить глобальную Lua переменную достаточно просто
```
Lua.Variable['variable_name'] := 5; // = 5
Lua.Variable['variable_name'] := Lua.Variable['variable_name'] - 18; // = -13
ShowMessage(Lua.Variable['variable_name']); // -13
```

Объявление свойства Variable выглядит следующим образом:
```
property Variable[const VariableName: string]: Variant;
```

Вы так же можете ассоциировать свою глобальную переменную или константу с глобальной переменной в Lua. Подробности смотрите в разделе «Регистрация глобальных переменных».

Определить, существует ли глобальная переменная можно с помощью
```
function TLua.VariableExists(const VariableName: string): boolean;
```

Существует так же продвинутый способ редактирования глобальной переменной в том случае если тип переменной переходит за возможности `Variant`. О возможностях `TLuaArg` читай в разделе «Тип TLuaArg».
```
property VariableEx[const VariableName: string]: TLuaArg;
```

Прописные и строчные символы

Хочу напомнить, что Lua – регистрозависимый язык. Переменные `variable_name` и `Variable_name` – принципиально разные переменные. Поэтому если у одного из ваших объектов есть свойство `Caption`, то свойства `caption` или `CAption` он не найдёт !

Вызов функций внутри Lua

Допустим есть файл с содержанием:

```
function test(a, b, c)
  return (a + b * c);
end
```

Как из кода Delphi вызвать эту функцию ?
Делается это просто. Сначала загружаете скрипт в Lua посредством одного из трёх вышеперечисленных методов, потом вызываете `Lua.Call('test', …)`:

```
var
  S: string;
begin
  S := Lua.Call('test', [1, 2, 3]).ForceString;
  ShowMessage(S); // 1 + 2 * 3 = 7. S = "7"
```

Функция TLua.Call имеет 2 объявления:

```
function Call(const ProcName: string; const Args: TLuaArgs): TLuaArg;
function Call(const ProcName: string; const Args: array of const): TLuaArg;
```

Как работать с TLuaArg – мы рассмотрим позже. Главное сейчас понять, что при вызове Call – вы указываете имя функции и её параметры. Функция может возвращать результат.

Определить, существует ли в Lua функция, можно с помощью:

```
function TLua.ProcExists(const ProcName: string): boolean;
```

Exception-ы Lua (работа с ошибками)

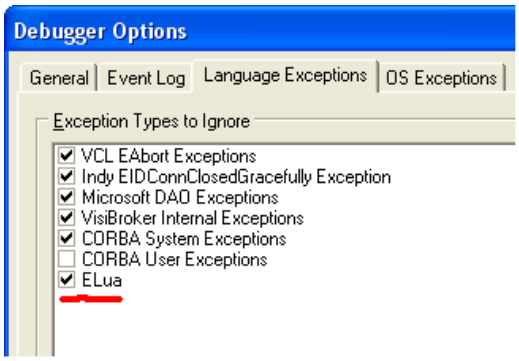
Для ошибок в CrystalLUA используется эксепшн `ELua`. Можете самостоятельно их создавать в случае необходимости. Для удобства есть классовые методы `Assert`

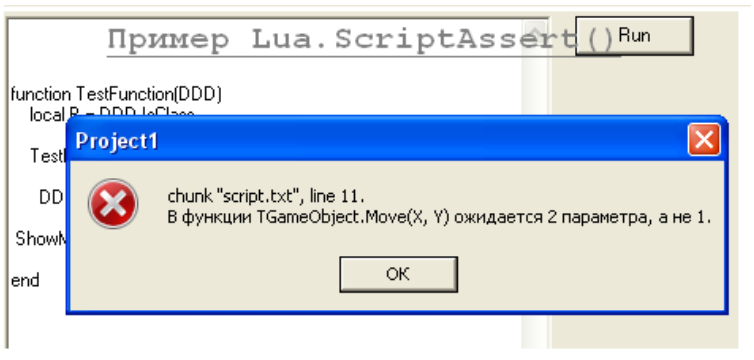
```
if not Lua.ProcExists(ProcName) then
  //raise ELua.CreateFmt('Глобальная функция "%s" не найдена !', [ProcName]);
  ELua.Assert('Глобальная функция "%s" не найдена !', [ProcName]);
```

Если скрипт неправильно вызывает код, то наиболее правильно вызывать Lua.ScriptAssert. О связывании своего кода и Lua мы будем говорить позже.

```
if (Lua.ArgsCount <> 2) then
Lua.ScriptAssert('В функции TGameObject.Move(X, Y) ожидается 2 параметра,'+
                 'а не %d', [Lua.ArgsCount]);
```

Рекомендую так же пометить `ELua` как игнорируемый эксепшн.





Препроцессинг. Точки и двоеточия

Программируя на Lua я столкнулся с одной неприятной особенностью. Классовые методы надо вызывать через двоеточие.

```
-- допустим есть глобальный объект Sprite: TSprite
-- который имеет метод TSprite.Move(const X, Y: integer)

  -- вызвав так я не укажу экземпляр класса. Ошибка!
  Sprite.Move(5, 10)

  -- правильно вызывать так. Передаётся экземпляр класса
  Sprite.Move(Sprite, 5, 10)

  -- "умные" lua-вцы придумали так. Это полностью эквивалентно
  -- предыдущей строке
  Sprite:Move(5, 10)
```

Лично для меня не совсем привычна запись `Sprite:Method()`. Кроме того Sprite может иметь свойства. И среди точек/запятых можно запутаться. Например так:

```
-- здесь нужно ставить точку
  Sprite.Speed = Sprite.Speed + 12.81

-- здесь нужно ставить двоеточие
  Sprite:GrowSpeed(12.81)

-- здесь нужно ставить точку
  local GrowSpeedFunc = Sprite.GrowSpeed
  . . . -- какие-то действия

  GrowSpeedFunc(Sprite1, Speed1)
  GrowSpeedFunc(Sprite2, Speed2)
  GrowSpeedFunc(Sprite3, Speed3)
```

Мной было принято решение написать препроцессинг скриптов и на данный момент реализована автоматическая замена точек на двоеточия в нужных местах. Теперь вы во всех случаях можете писать точки и не запариваться над двоеточиями. Например так:

```
-- свойства. точка
  Sprite.Speed = Sprite.Speed + 12.81

-- вызов метода. точка
  Sprite.GrowSpeed(12.81)

-- функция как свойство. точка
  local GrowSpeedFunc = Sprite.GrowSpeed
```

Существуют и другие особенности препроцессинга. Но "обычные скрипты" по прежнему поддерживаются – можете использовать ранее написанный Lua-код.

Тип TLuaArg

Язык Lua – это язык с динамической типизацией. Т.е. каждая переменная что-то типа Variant-а, только имеет свои особенности. Я решил взять эту идею и адаптировать под наши реалии, получился тип TLuaArg (аналог Variant, только с заточкой под Lua). Благодаря этому типу появилась возможность удобно взаимодействовать с Lua, не прибегая к Lua API, не анализируя стек, удобно преобразуя один тип к другому и вместе с тем, при необходимости, отправлять нужные данные в Lua. TLuaArg может принимать значения простых типов (Boolean, Integer, Double, String, Pointer) и сложных типов (TClass, TObject, структура, массив, множество, Lua-таблица).

Основы

```
// типы, по которым идёт взаимодействие между Lua и кодом
TLuaArgType = (ltEmpty, ltBoolean, ltInteger, ltDouble, ltString, ltPointer,
               ltClass, ltObject, ltRecord, ltArray, ltSet, ltTable);
TLuaArg = object
private
. . .
public
    property LuaType: TLuaArgType; // только чтение
    property LuaTypeName: string; // имя типа. чтение
    property Empty: boolean; // чтение/запись
. . .
```

Определить какой тип сейчас лежит внутри TLuaArg можно с помощью свойства LuaType. Его имя – значение свойства LuaTypeName. Есть свойство Empty, которое показывает пуст ли сейчас TLuaArg. При желании можно очистить такую переменную.

```
var
  Arg: TLuaArg;
begin
  Arg.AsInteger := 15; // значение 15
  if (Arg.Empty) then Arg.AsDouble := 17.85; // значение не изменилось. = 15
  Arg.Empty := true; // очистить
```

As и Force

Доступ к TLuaArg производится по As_ свойствам. Например:

```
begin
  Arg1.AsBoolean := true;
  if (Arg1.AsBoolean) then Arg2.AsDouble := 18.41 else Arg2.Empty := true;
```

Но если вы попытаетесь взять значение несоответствующего типа, то получите Exception

```
  Arg.AsPointer := @Data;
  Value := Arg.AsInteger; // Exception. Тип в Arg - не Integer !
```

Часто существуют необходимости автоматического преобразования типа. Например в случае Variant прокатывает вот такая вещь:

```
var
  V: Variant;
begin
  V := 15; // или 'строковое значение' или True или 18.21
  Form1.Caption := V; // здесь V автоматически преобразовывается к строке
```

Примерно такое же преобразование можно организовать в TLuaArg. Без Exception-ов. За это отвечают Force-версии функций

```
begin
  Arg.AsInteger := 18;
  Form1.Caption := Arg.ForceString; // '18'
  Arg.AsBoolean := False;
  Form1.Caption := Arg.ForceString; // 'False'
  Arg.AsObject := Form1;
  Form1.Caption := Arg.ForceClass.ClassName; // 'TForm1'
  Arg.AsString := '15.84';
  Form1.Caption := FloatToStr(Arg.ForceDouble + 81); // '96.84'
  { и т.д. }
```

Force-вариант никогда не вызывает Exception-ы. As-вариант вызывает Exception в случае несовместимости типов. Каким образом получать значение из TLuaArg – решать вам.

Variant

Интересно как взаимодействуют TLuaArg иVariant. Поддерживаются типы: Boolean, Integer, Double, String. Например:

```
begin
   Arg.AsVariant := 15.41; // Arg.LuaType = ltDouble; Arg.AsDouble = 15.41
   Form1.Caption := Arg.AsVariant; // = '15.41'
```

Функции LuaArg()

Для удобного создания TLuaArg существуют следующие функции:

```
function LuaArg(const Value: boolean): TLuaArg; overload;
function LuaArg(const Value: integer): TLuaArg; overload;
function LuaArg(const Value: double): TLuaArg; overload;
function LuaArg(const Value: string): TLuaArg; overload;
function LuaArg(const Value: pointer): TLuaArg; overload;
function LuaArg(const Value: TClass): TLuaArg; overload;
function LuaArg(const Value: TObject): TLuaArg; overload;
function LuaArg(const Value: TLuaRecord): TLuaArg; overload;
function LuaArg(const Value: TLuaArray): TLuaArg; overload;
function LuaArg(const Value: TLuaSet): TLuaArg; overload;
function LuaArg(const Value: Variant): TLuaArg; overload;
```

Если необходимо создать массив TLuaArg (для вызова TLua.Call например), то пользуйте:

```
function LuaArgs(const Count: integer): TLuaArgs;
```

Регистрация

Закончили с описанием общей информации, приступаем к практической части. Для того чтобы Lua взаимодействовал с нативном кодом, и нативный код взаимодействовал с Lua кодом – нужна регистрация. Поехали.

Регистрация глобальных переменных

Одна из замечательных особенностей библиотеки CystalLUA – это возможность регистрировать глобальные переменные. Глобальные переменные Lua и глобальные переменные нативного кода – теперь в одном пространстве.

Регистрация производится одной функцией `TLua` :

```
procedure RegVariable(const VariableName: string; const X;
                      const tpinfo: pointer; const IsConst: boolean = false);
```

Например:

```
var
  GameSpeed: single; // есть глобальная переменная
. . .
Lua.RegVariable('GameSpeed',GameSpeed,typeinfo(single){,true если константа})
```

Теперь в скрипте вы можете написать:

```
local speed = GameSpeed -- присвоить значение глобальной переменной GameSpeed
GameSpeed = speed + 15  -- изменить значение глобальной переменной
```

В качестве первого параметра указывается имя переменной. В качестве второго параметра указывается сама переменная. Нетипизированная запись `const X;` предусматривает указание типизированной переменной или константы. На самом деле указывается ссылка. Третий аргумент – это typeinfo такой переменной. Наиболее сложно в таком случае обстоят дела со структурами, статическими массивами, типом Pointer и TClass. Дело в том, что для данных типов RTTI может не формироваться. Для переменных типа Pointer в качестве аргумента указывайте константу typeinfoPointer, для переменной типа TClass – константу typeinfoTClass. В случае если для ваших массивов и структур не генерируются RTTI то указывайте PLuaRecordInfo и PLuaArrayInfo. Для множеств – PLuaSetInfo. Как получить эти значения – я расскажу подробно в соответствующих разделах. Если переменная – экземпляр класса, то указывайте typeinfo(TObject) или другого класса – всё равно; в библиотеке переменная будет числиться как “’экземпляр класса”.

Особое внимание уделю последнему параметру – IsConst. Существуют типизированные константы. Если это ваш случай, то обязательно указывайте флаг в True. Но самое главное, предназначение этого флага – показать, что значение нельзя менять из Lua. Будьте внимательны. Можете случайно допустить ошибку.

```
const
  GameSpeed: single = 18; // есть глобальная типизированная константа
  Lua.RegVariable('GameSpeed',GameSpeed,typeinfo(single){IsConst=False !!!});

-- Неявная логическая ошибка !
GameSpeed = 15 -- изменение глобальной типизированной константы из скрипта
```

Регистрация констант

Что такое константы, думаю рассказывать не надо

```
const
  MAX_PATH = 260;
  MAX_INT = 2147483647;
  PHYSICS_DT = 0.1;
  DEBUG_MODE = True;
  OPTIONS_FILE_NAME = 'options.xml';
  FAIL_PTR = pointer($FFFFFFFF);
```

Для регистрации таких констант в Lua существуют:

```
procedure TLua.RegConst(const ConstName: string; const Value: Variant);
procedure TLua.RegConst(const ConstName: string; const Value: TLuaArg);
```

`RegConst` принципиально отличается от `RegVariable(..., IsConst=True)`. Неизменяемая переменная всегда жёстко типизирована и получение её значения происходит в реальном времени, преобразовывая нативное значение к Lua-значению. Константы, которые регистрируются с помощью `RegConst` сразу преобразовываются к Lua-виду и хранятся в таком виде, не требуя преобразований.

Регистрация констант перечисляемых типов

Перечисляемые типы это:

```
type
  TScrollBarKind = (sbHorizontal, sbVertical);
  TAlign = (alNone, alTop, alBottom, alLeft, alRight, alClient, alCustom);
  TMouseButton = (mbLeft, mbRight, mbMiddle);
  . . .
```

Перечисляемые типы регистрируются так:

```
  Lua.RegEnum(typeinfo(TScrollBarKind)); // 'sbHorizontal' и 'sbVertical'
  Lua.RegEnum(typeinfo(TAlign)); // TAlign: 'alNone', 'alTop', ...
  Lua.RegEnum(typeinfo(TMouseButton)); // 'mbLeft', 'mbRight', 'mbMiddle'
```

Но важно понимать, что перечисляемые типы регистрируются как целочисленные значения. Нет функции, позволяющей преобразовывать строковые идентификаторы в целочисленные и обратно (в Lua). Если нативная переменная имеет перечисляемый тип, то через Lua ей вполне можно присвоить целочисленное значение. Константа `mbLeft` например регистрируется как 0. `alClient` - как 5. `sbVertical` - 1.

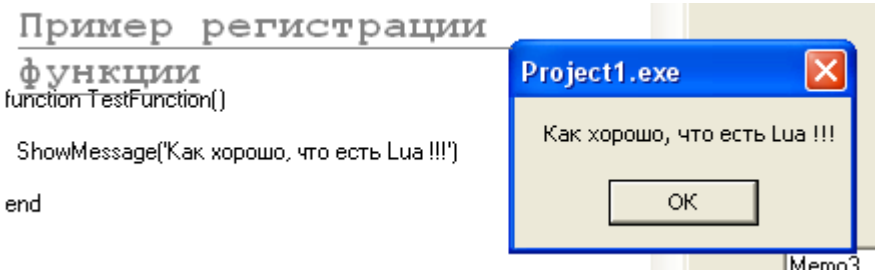
```
-- пример
local X = alTop + 15 * mbMiddle -- X = 1 + 15 * 2 = 31
```

Регистрация глобальных функций

Попробую показать, как регистрировать глобальные функции на примере ShowMessage. Как я уже говорил, Lua – язык с динамической типизацией. Lua мало что знает о нативных функциях и их аргументах, поэтому использует виртуальный стек переменных, тип которых можно узнать только во время выполнения кода. Lua API предлагает методы, с помощью которых можно взять значения указанные в качестве аргументов. Я отказался от использования такого подхода и вместо многочисленных API предлагаю использовать универсальный тип TLuaArg (описание в разделе «Тип TLuaArg»). Регистрация всех функций (в данном случае ShowMessage) происходит за счёт регистрации специальных калбеков. О классовых функциях мы поговорим позже, калбек для глобальной функции выглядит так:

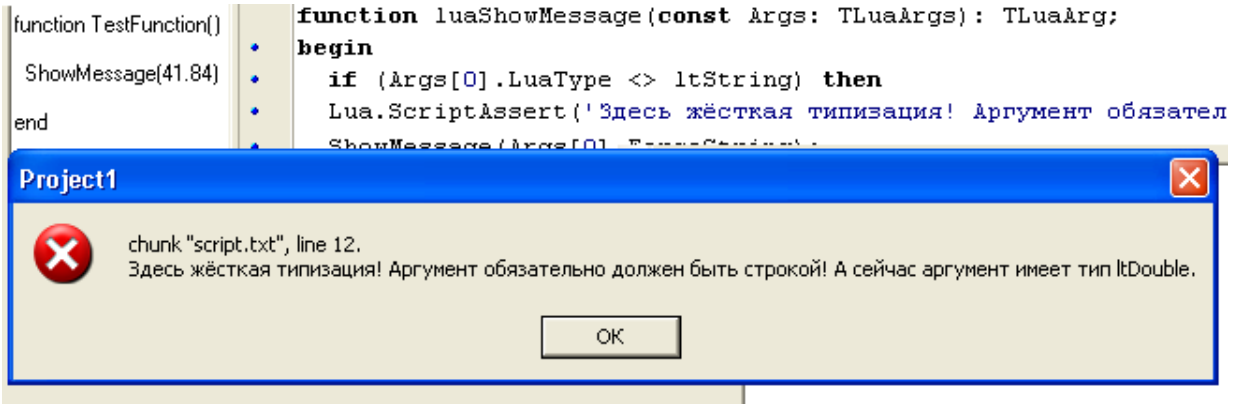
```
TLuaProc = function(const Args: TLuaArgs): TLuaArg;


// устанавливаем калбек для функции ShowMessage
// привыкайте названия калбеков писать с префиксом 'lua'
function luaShowMessage(const Args: TLuaArgs): TLuaArg;
begin
  ShowMessage(Args[0].ForceString);
end;
. . .
Lua.RegProc('ShowMessage', @luaShowMessage, 1{количество параметров.необяз.})
```

Попробуйте скрипт и получите примерно такой результат:



В случае ошибки (метод ScriptAssert)

Библиотека CrystalLUA вообще изобилует проверками на правильное использование и различными Exception-ами. Скрипт – дело ненадёжное, а я приверженец максимально надёжных подходов. Я уже рассказал немного о методе TLua.ScriptAssert() в разделе «Exception-ы Lua (работа с ошибками)». Если скрипт каким либо образом неправильно вызывает калбек, то обязательно вызывайте TLua.ScriptAssert().



Проверка на количество аргументов

Одной из самых логичных проверок является проверка на количество аргументов. Согласитесь, будет неправильно, если функция имеет например 2 параметра, а пользователь отсылает 1 или 3. Кроме того есть и второе проявление

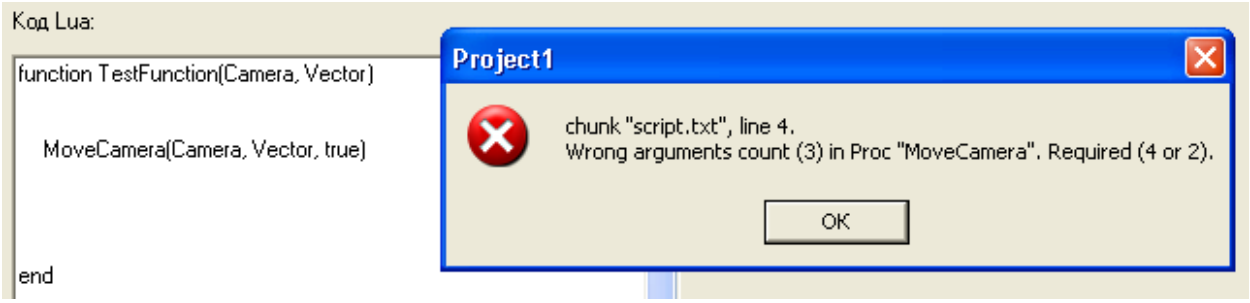
```
procedure MoveCamera(const Camera: TCamera; const X, Y, Z: single); overload;
procedure MoveCamera(const Camera: TCamera; const Vector: TVector); overload;
```

Это проявление – overload – перегрузка методов. Когда функция с одним именем может иметь разновидности: с разным количеством аргументов. Во всех случаях хорошо подходит функция TLua.CheckArgsCount(), которая не только проверяет количество аргументов, но и вызывает TLua.ScriptAssert в случае ошибки

```
function  CheckArgsCount(const ArgsCount: array of integer;
          const ProcName: string=''; const AClass: TClass=nil): integer;
function  CheckArgsCount(const ArgsCount: TIntegerDynArray;
          const ProcName: string=''; const AClass: TClass=nil): integer;
procedure CheckArgsCount(const ArgsCount: integer; const ProcName: string='';
          const AClass: TClass=nil);
```

```
// Примеры
function luaShowMessage(const Args: TLuaArgs): TLuaArg;
begin
  Lua.CheckArgsCount(1, 'ShowMessage'); // количество аргументов = 1 ?
```

```
function luaMoveCamera(const Args: TLuaArgs): TLuaArg;
var
  I: integer;
begin
  I := Lua.CheckArgsCount([4, 2], 'MoveCamera');

  case I of
    0: { MoveCamera(Camera, X, Y, Z) } ;
    1: { MoveCamera(Camera, Vector) }  ;
  //
  { а иначе вообще автоматически вызовется ScriptAssert }
  end;
```



```
// о классовых калбеках мы ещё позже поговорим
function TSprite.luaMove(const Args: TLuaArgs): TLuaArg;
begin
  Lua.CheckArgsCount(2, 'Move', TSprite); { в данном случае если количество
                аргументов не 2, то в описании ошибки будет "TSprite.Move" }
```

Существуют так же автоматические проверки на количество аргументов.

```
procedure TLua.RegProc(const ProcName: string; const Proc: TLuaProc;
                       const ArgsCount: integer=-1); overload;
```

Если количество агрументов указано явно, то проверка CheckArgsCount() происходит автоматически. В ShowMessage например можно явно указать 1. В TSprite.Move – 2. Ещё хочу сказать, что информацию об аргументах в калбеках можно так же смотреть в экземпляре TLua:

```
    property Args: TLuaArgs read FArgs;
    property ArgsCount: integer read FArgsCount;
```

Альтернативные калбеки

Иногда возникает необходимость воспользоваться “альтернативным калбеком”. Ниже приведены типы калбеков, которые так же можно использовать. “Приведение” до типа TLuaProc делается функцией LuaProc. Если вы знакомы с ассемблером и знакомы с основами использования структур и динамических массивов, то согласитесь, что во всех случаях – одинаково используются регистры EAX и EDX.

```
  TLuaProc = function(const Args: TLuaArgs): TLuaArg;
  TLuaProc0 = TLuaProc;
  TLuaProc1 = function (const Arg: TLuaArg): TLuaArg;
  TLuaProc2 = procedure(const Args: TLuaArgs; var Result: TLuaArg);
  TLuaProc3 = procedure(const Arg: TLuaArg; var Result: TLuaArg);
  TLuaProc4 = procedure(const Args: TLuaArgs);
  TLuaProc5 = procedure(const Arg: TLuaArg);
  TLuaProc6 = procedure();
  function LuaProc(const Proc: TLuaProc0): TLuaProc; overload;
  function LuaProc(const Proc: TLuaProc1): TLuaProc; overload;
  function LuaProc(const Proc: TLuaProc2): TLuaProc; overload;
  function LuaProc(const Proc: TLuaProc3): TLuaProc; overload;
  function LuaProc(const Proc: TLuaProc4): TLuaProc; overload;
  function LuaProc(const Proc: TLuaProc5): TLuaProc; overload;
  function LuaProc(const Proc: TLuaProc6): TLuaProc; overload;

// альтернативный калбек
procedure luaShowMessage(const Arg: TLuaArg);
begin
  ShowMessage(Arg.ForceString);
...
Lua.RegProc('ShowMessage', LuaProc(@luaShowMessage), 1{кол-во параметров});
```

Регистрация классов

Чаще всего при программировании в Lua (да и вообще в любом другом языке программирования) приходится придерживаться ООП парадигмы. Для того чтобы использовать классы и их экземпляры в Lua, классы нужно зарегистрировать. При регистрации класса регистрируется вся его иерархия вплоть до TObject. Например, зарегистрировав TComponent, вы также зарегистрируете TPersistent и TObject (хотя TObject уже изначально зарегистрирован). Для того чтобы зарегистрировать классы, существуют 2 функции:

```
procedure RegClass(const AClass: TClass; const use_published: boolean=true);
procedure RegClasses(AClasses: array of TClass; use_published: boolean=true);
```

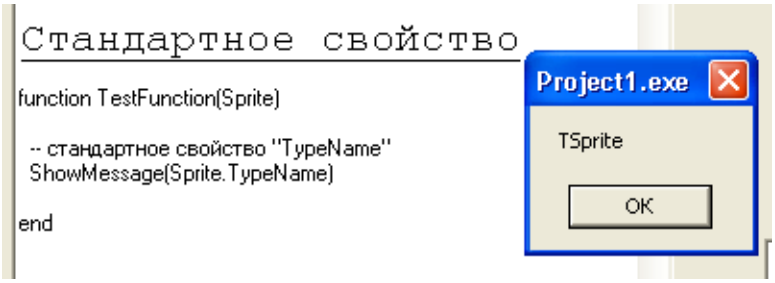
```
// зарегистрировать класс и его предков
Lua.RegClass(TComponent);

// зарегистрировать сразу несколько классов (и их предков)
Lua.RegClasses([TSprite, TGameEngine, TParticleSystem, TTexture]);

// зарегистрировать класс, не используя его published информацию
Lua.RegClass(TButton, false);
```

О том, что такое published и как это использовать – я буду подробно рассказывать в разделе «Published и авторегистраторы». Сейчас главное понять – после регистрации класса вы можете использовать их в Lua и передавать их в качестве параметра между кодом и Lua.

Стандартные свойства

Мне понравилась идея, что у классов есть стандартные свойства. Например ClassName, ClassParent, InstanceSize и другие. Похожую концепцию я решил применить и в CrystalLUA. Но стандартные свойства применимы не только к классам, но и к другим сложным объектам: структурам, множествам, массивам.

```
var
  Sprite: TSprite;
begin
  Sprite := TSprite.Create(0, 0, skWeapon, 'weapon.png') ;

  Lua.RegClass(TSprite);
  Lua.Call('TestFunction', [Sprite]); // вызвать функцию TestFunction(Sprite)
```



Следующие стандартные свойства и методы характерны для всех сложных типов: классы, структуры, множества, массивы. Однако существуют и специфичные свойства/методы, о которых в каждом конкретном случае я буду говорить.

```
{ тип объекта. TClass, PLuaRecordInfo, PLuaArrayInfo, PLuaSetInfo }
Type
{ имя типа объекта }
TypeName
{ класс-родитель (только для классов. Иначе вернётся nil) }
TypeParent
{ наследуется ли объект от класса, структуры, массива, множества }
InheritsFrom()
{ метод "присвоить". Подробнее в разделе «Метод Assign()» }
Assign()
{ является ли объект ссылкой. Чаще он показывает, создан ли объект "на стеке" Lua. Если True, то он создан и удалится на нативной стороне. Если False – то создан и удалится на стороне Lua. Об этом более подробно будет дальше }
IsRef
{ можно ли изменять значения объекта. Применимо больше к структурам }
IsConst
// является ли объект классом
IsClass
// является ли объект структурой
IsRecord
// является ли объект массивом
IsArray
// является ли объект множеством
IsSet
// является ли объект пустым (пустой массив, или пустая структура, или …)
IsEmpty
```

Конструкторы и деструкторы

CrystalLUA позволяет не только использовать созданные на нативной стороне объекты, но и создавать/удалять их внутри Lua. Обычный подход для создания объекта на стороне Lua – это такой:

```
local X = TSprite() -- конструктор TSprite без параметров
local Y = TIntegerDynArray() -- конструктор динамического массива
local Z = TPoint(10, 15) -- конструктор TPoint с параметрами
local A = TMouseButtonSet(mbLeft, mbRight) -- конструктор множества
local B = TStringArray("1", "2", "test", "") – конструктор и наполнение

-- что характерно, для всех этих переменных: X, Y, Z, A, B
-- они будут автоматически удалены в Lua по GC ("чистка мусора")
```

Позже я расскажу как создавать множества, структуры и массивы, сейчас сосредоточимся на конструкторах и деструкторах именно классов.

Как я уже говорил, если вы зарегистрировали класс, то можете им пользоваться. Если вы зарегистрировали класс TSprite и вызовите вот этот код, то в тексте сообщения будет “TSprite”, а сам объект (X) будет удалён при ближайшем GC (garbage collection).

```
local X = TSprite()
ShowMessage(X.TypeName)
```

Кстати говоря, чистку мусора, вы всегда можете вызвать самостоятельно:

```
procedure TLua.GarbageCollection();
```

Классы отличаются от всех остальных сложных типов (массивы, структуры, множества) тем, что они могут вызывать конструктор, не только “на стеке”, но и “в куче“. За создание и удаление экземпляров классов “в куче“ отвечают привычные нам методы Create и Free.

```
-- создание и удаление экземпляра класса “в куче”
local X = TSprite.Create()
ShowMessage(X.TypeName)
X.Free() -- явный деструктор
```

Create и Free работают только с классами. Free (как и Create) – это метод, поэтому вы обязаны указывать скобки “()”. Экземпляру класса, созданному “на стеке” вы можете вызвать явный деструктор. Но учтите, что если объект создан на стеке, то вызвав его деструктор на нативной стороне, вы повлечёте ошибки.

```
procedure luaSpriteMove(const Args: TLuaArgs);
var
  Sprite: TSprite;
begin
  Sprite := TSprite(Args[0].AsObject);
  Sprite.Move(Args[1].AsInteger, Args[2].AsInteger)
  Sprite.Free; // объект удалится. но в Lua он числится как доступный
. . .
  local Sprite = TSprite()
  SpriteMove(Sprite, 10, 20)
  -- в GC будет вызван деструктор объекта Sprite и будет вызван Exception
  -- если бы Free() был вызван в Lua , то работа была бы корректной.
```

Всем понятно, что конструкторы часто имеют параметры. Для того чтобы использовать конструктор с параметрами в CrystalLUA достаточно зарегистрировать метод ‘constructor’. Подробнее о регистрации классовых методов смотрите в следующем разделе.

```
// конструктор в Lua - это метод, инициализируйщий уже созданный объект
procedure TSpriteConstructor(var X; const Args: TLuaArgs);
var
  Sprite: TSprite absolute X;
begin
  Sprite.X := Args[0].AsInteger; Sprite.Y := Args[1].AsInteger;
. . .
Lua.RegProc(TSprite, LUA_CONSTRUCTOR, LuaClassProc(TSpriteConstructor));
. . .
local Sprite = TSprite(10,20) -- а можно local Sprite = TSprite.Create(10,20)
```

Методы

Кроме стандартных свойств и методов можно регистрировать свои свойства и методы. О регистрации свойств речь пойдёт позже, сейчас поговорим о методах. Калбек для классовых методов в общем случае выглядит так:

```
TLuaClassProc = function(const Args: TLuaArgs): TLuaArg of object;
```

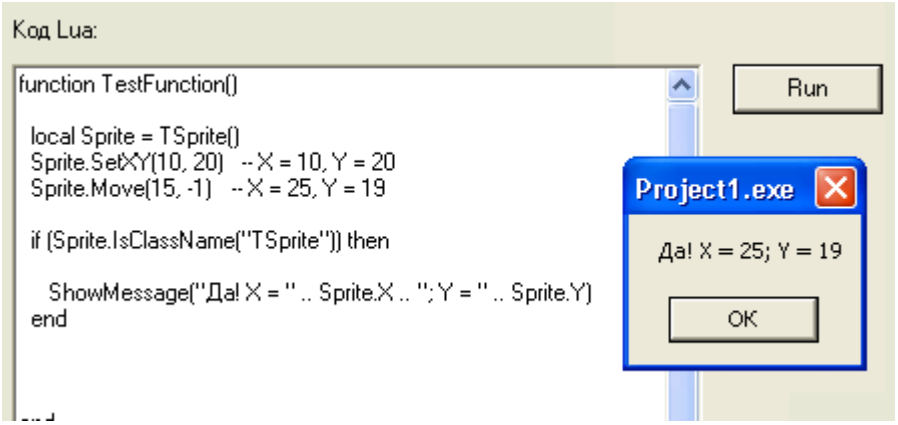
И для него есть очень много аналогов:

```
// функция является членом класса
TLuaClassProc0 = TLuaClassProc;
TLuaClassProc1 = function (const Arg: TLuaArg): TLuaArg of object;
TLuaClassProc2 = procedure(const Args:TLuaArgs; var Result:TLuaArg)of object;
TLuaClassProc3 = procedure(const Arg: TLuaArg; var Result:TLuaArg) of object;
TLuaClassProc4 = procedure(const Args: TLuaArgs) of object;
TLuaClassProc5 = procedure(const Arg: TLuaArg) of object;
TLuaClassProc6 = procedure() of object;

// глобальные методы. Первый аргумент - экземпляр класса
TLuaClassProc7 = function(const AObject:TObject;const Args:TLuaArgs):TLuaArg;
TLuaClassProc8 = function(const AObject: TObject;const Arg:TLuaArg) :TLuaArg;
TLuaClassProc9 = procedure(const AObject: TObject; const Args: TLuaArgs;
                           var Result: TLuaArg);
TLuaClassProc10 = procedure(const AObject: TObject; const Arg: TLuaArg;
                            var Result: TLuaArg);
TLuaClassProc11 = procedure(const AObject: TObject; const Args: TLuaArgs);
TLuaClassProc12 = procedure(const AObject: TObject; const Arg: TLuaArg);
TLuaClassProc13 = procedure(const AObject: TObject);

// калбеки для структур
TLuaClassProc14 = function (var X; const Args: TLuaArgs): TLuaArg;
TLuaClassProc15 = function (var X; const Arg: TLuaArg): TLuaArg;
TLuaClassProc16 = procedure(var X; const Args: TLuaArgs;var Result: TLuaArg);
TLuaClassProc17 = procedure(var X; const Arg: TLuaArg; var Result: TLuaArg);
TLuaClassProc18 = procedure(var X; const Args: TLuaArgs);
TLuaClassProc19 = procedure(var X; const Arg: TLuaArg);
TLuaClassProc20 = procedure(var X);

// калбеки для классовых методов
TLuaClassProc21 = function(const AClass: TClass;const Args:TLuaArgs):TLuaArg;
TLuaClassProc22 = function (const AClass: TClass; const Arg:TLuaArg):TLuaArg;
TLuaClassProc23 = procedure(const AClass: TClass; const Args: TLuaArgs;
                            var Result: TLuaArg);
TLuaClassProc24 = procedure(const AClass: TClass; const Arg: TLuaArg;
                            var Result: TLuaArg);
TLuaClassProc25 = procedure(const AClass: TClass; const Args: TLuaArgs);
TLuaClassProc26 = procedure(const AClass: TClass; const Arg: TLuaArg);
TLuaClassProc27 = procedure(const AClass: TClass);
```

Если вы знакомы с ассемблером и основами использования структур и массивов, то увидите, что калбеки эквивалентны, потому что одинаково используют регистры EAX, EDX и ECX. Перевод “аналога” к TLuaClassProc производится функцией LuaClassProc.

```
function LuaClassProc(const Proc: TLuaClassProc0): TLuaClassProc; overload;
. . .
function LuaClassProc(const Proc: TLuaClassProc27): TLuaClassProc; overload;
```

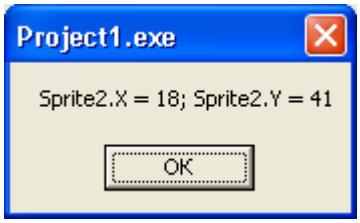
```
type                        { -- Пример регистрации -- }
  TSprite = class(TGameObject)
  private
    FX, FY: integer;
    procedure luaMove(const Args: TLuaArgs); // член класса
  public
    procedure SetXY(AX, AY: integer); // FX := AX; FY := AY;
    procedure Move(AX, AY: integer); // FX := FX + AX; FY := FY + AY;

    property X: integer read FX;
    property Y: integer read FY;
  end;

// калбек для "Move". является членом класса
procedure TSprite.luaMove(const Args: TLuaArgs);
begin
  Move(Args[0].AsInteger, Args[1].AsInteger);
end;

// калбек для "SetXY". является глобальной функцией
procedure luaTSpriteSetXY(const AObject: TObject; const Args: TLuaArgs);
var
  Sprite: TSprite absolute AObject;
begin
  Sprite.SetXY(Args[0].AsInteger, Args[1].AsInteger);
end;

// произвольный классовый метод
function luaIsClassName(const AClass: TClass; const Arg: TLuaArg): TLuaArg;
begin
  Result.AsBoolean := (AClass.ClassNameIs(Arg.ForceString));
end;

// регистрация
Lua.RegProc(TSprite, 'Move', LuaClassProc(TSprite(nil).luaMove), 2);
Lua.RegProc(TSprite, 'SetXY', LuaClassProc(@luaTSpriteSetXY), 2);
Lua.RegProc(TSprite,'IsClassName',LuaClassProc(@luaIsClassName),1, TRUE{!!!);
```



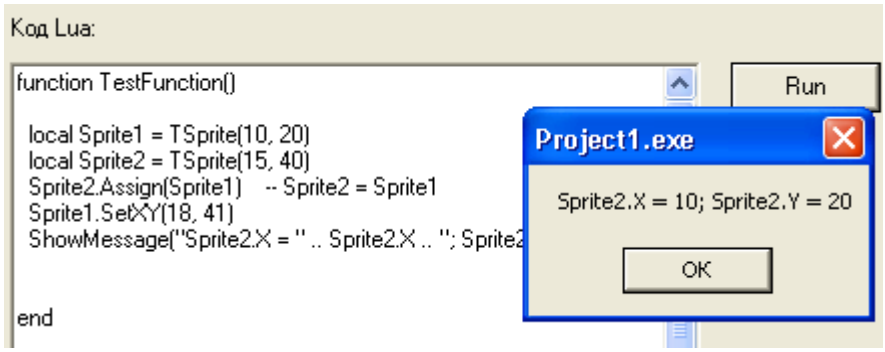
Метод Assign()

Lua по своей структуре оперирует не самими данными, а ссылками на них. Это особенно заметно на сложных типах: классах, структурах, массивах, множествах

```
local Sprite1 = TSprite(10, 20)
local Sprite2 = TSprite(15, 40)
Sprite2 = Sprite1 --копируются не данные, а ссылка. старый объект удаляется
Sprite1.SetXY(18, 41)
ShowMessage("Sprite2.X = " .. Sprite2.X .. "; Sprite2.Y = " .. Sprite2.Y)
```



Как использовать свойства – мы рассмотрим в следующем разделе. Сейчас сосредоточимся на методе Assign. Метод Assign копирует данные – как для классов, так и для структур, и для массивов, и для множеств.



Метод Assign достаточно умный. Он копирует не только обычные данные, но и сложные: строки, массивы, интерфейсы, варианты – всё по правилам RTTI. Классу никогда не будет присвоена структура, а массиву – никогда множество. Но есть вещи, которые не поддаются стандартным алгоритмам. Например это могут быть дополнительные инициализации, собственные списки и т.д. Поэтому в CrystalLUA есть возможность переопределить метод Assign для структур и классов. Если оба аргумента (в данном случае к примеру Sprite1 и Sprite2) – потомки класса TPersistent, то у первого аргумента будет вызван метод TPersistent.Assign(...), который в свою очередь можно переопределить. В противном случае приходится регистрировать стандартный метод ‘assign’

```
procedure TSpriteAssign(const AObject: TObject; const Arg: TLuaArg);
var
  SpriteSrc: TSprite absolute AObject;
  SpriteValue: TSprite;
begin
  SpriteValue := Arg.AsObject;
  { заполнить SpriteSrc полями SpriteValue }
. . .
// регистрация метода Assign
Lua.RegProc(TSprite, LUA_ASSIGN{'assign'}, LuaClassProc(TSpriteAssign));
```

Свойства

Свойствам в CrystalLUA уделено **очень** много времени. Всю прелесть свойств вы узнаете в разделе «Published и авторегистраторы», но я прошу вас не забегать вперёд. Я ещё не совсем определён, какую типизацию надо выбрать: жёсткую или мягкую, но на данный момент свойства работают – и это хорошо. Свойства в CrystalLUA регистрируются просто

```
procedure TLua.RegProperty(const AClass: TClass; const PropertyName: string;
                           const tpinfo: pointer; const PGet, PSet: pointer;
                           {о доп. параметрах поговорим позже} );
```

Первый параметр – класс, в котором регистрируется свойство. В нашем случае TSprite. Второй аргумент – имя свойства. В нашем случае это “X”и ”Y”. Третий параметр – это typeinfo, в нашем случае typeinfo(integer). Если свойство – структура, множество или массив – то указывается соответствующий “Handle” (о которых мы поговорим позже). Если свойство – TClass – тогда в качестве аргумента указываем константу typeinfoTClass (для TClass не создаётся typeinfo), Если свойство имеет тип Pointer (или любой другой указатель), то в качестве аргумента указывайте константу typeinfoPointer. PGet и PSet – это

- либо указатели на поле
- либо указатели на функцию сеттер/геттер (можно виртуальные)
- либо **nil**

```
type
  TSprite = class(TGameObject)
  private
    FX, FY: integer;

    procedure SetY(const Value: integer);
    function GetName: string; virtual;
  public
    property X: integer read FX write FX;
    property Y: integer read FY write SetY;
    property Name: string read GetName;
  end;

. . .

// регистрация read/write по полю
Lua.RegProperty(TSprite, 'X', typeinfo(integer),
                @TSprite(nil).FX, @TSprite(nil).FX);

// read по полю, а write через сеттер
Lua.RegProperty(TSprite, 'Y', typeinfo(integer),
                @TSprite(nil).FY, @TSprite.SetY);

// readonly (через виртуальный! геттер)
Lua.RegProperty(TSprite, 'Name', typeinfo(string), @TSprite.GetName, nil{!});
```

Сложные свойства

Полное описание TLua.RegProperty:

```
procedure TLua.RegProperty(const AClass: TClass; const PropertyName: string;
                           const tpinfo: pointer; const PGet, PSet: pointer;
        const parameters: PLuaRecordInfo=nil; const default: boolean=false);
```

Оставшиеся 2 параметра (parameters и default) отвечают за регистрацию сложных свойств. Что такое сложное свойство? Они используются очень часто

```
type
  TSpriteStorage = class(TGameResource)
  private
  . . .
  public
    // обычное свойство
    property Count: integer read GetCount;
    // сложное индексированное дефолтное свойство
    property Item[Index: integer]: TSprite read GetItem; default;
    // сложное свойство, взять по имени
    property SpriteByName[const Name: string]: TSprite read GetSpriteByName;

// класс TStrings (модуль Classes.pas)
property Strings[Index: Integer]: string read Get write Put; default;

// класс TCanvas (модуль Graphics.pas)
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
```

Со всеми этим свойствами тоже можно работать! Все сложные свойства не обращаются к полям напрямую, а взаимодействие с ними осуществляется через сеттеры и геттеры

```
{  фрагмент кода модуля Graphics.pas  }
function TCanvas.GetPixel(X, Y: Integer): TColor;
begin
  RequiredState([csHandleValid]);
  GetPixel := Windows.GetPixel(FHandle, X, Y);
end;

procedure TCanvas.SetPixel(X, Y: Integer; Value: TColor);
begin
  Changing;
  RequiredState([csHandleValid, csPenValid]);
  Windows.SetPixel(FHandle, X, Y, ColorToRGB(Value));
  Changed;
end;
```

Т.к. количество аргументов в сложных свойствах неограниченно, а тип их заранее неизвестен, поэтому со сложными свойствами не получится работать как с простыми. Чтобы зарегистрировать сложное свойство, вам потребуется сначала зарегистрировать структуру параметров, по которым идёт взаимодействие. Для свойства Pixels[,] она будет

```
type
  TPixelsProperty = record
    X: integer; // первый аргумент
    Y: integer; // второй аргумент
  end;
```

О том как зарегистрировать структуру – мы будем говорить в разделе «Регистрация структур». Сейчас давайте предположим, что структуру TPixelsProperty мы зарегистрировали. Регистрация свойства Pixels будет выглядеть так

```
type
  TCanvasEx = class(TCanvas) // враппер для класса TCanvas
  private
    // function GetPixel(X, Y: Integer): TColor;
    function GetPixel(const{!!!} Params: TPixelsProperty): TColor;
    // procedure SetPixel(X, Y: Integer; Value: TColor);
    procedure SetPixel(const{!!!} Params: TPixelsProperty; Value: TColor);
  end;

function TCanvasEx.GetPixel(const Params: TPixelsProperty): TColor;
begin  Result := Self.Pixels[Params.X][Params.Y];  end;

procedure TCanvasEx.SetPixel(const Params: TPixelsProperty; Value: TColor);
begin  Self.Pixels[Params.X][Params.Y] := Value;  end;

// регистрация
Lua.RegProperty(TCanvas, 'Pixels', typeinfo(TColor), @TCanvasEx.GetPixel,
                @TCanvasEx.SetPixel, Lua.RecordInfo['TPixelsProperty']);

-- в тексте скрипта можно будет пользоваться сложным свойством Pixels
Bitmap.Canvas.Pixels[1][2] = clRed
```

До сих пор мы не коснулись вопроса дефолтных свойств. Дефолтное свойство – это сложное свойство, которое подразумевается, если к экземпляру класса производят обращение как к сложному свойству. При объявлении класса его обозначают директивой **default**. В TLua.RegProperty(…) в таком случае явно указывайте флаг default = true.

```
Lua.RegProperty(TSpriteStorage, 'Item', typeinfo(TSprite),
                @TSpriteStorage.GetItem, nil, INDEXED_PROPERTY, TRUE{!!!});

-- в скрипте можно написать так:
local Sprite = SpriteStorage[5] -- эквивалент  Sprite = SpriteStorage.Item[5]
```

Настало время поговорить об индексных свойствах. И свойствах “по имени”. Это сложные свойства. Отличие их от других сложных свойств состоит в том, что во-первых, они самые распространённые сложные свойства, во-вторых, для них не нужно создавать дополнительных структур и врапперов. Смотрите сами. Необходимость в “структурах параметров” возникает потому, что возникает необходимость передать несколько параметров. И универсально нам на это нужно 4 байта. Т.е. когда CrystalLUA вызывает калбек свойства с параметрами – он вызывает калбек и в качестве первого аргумента указывает ссылку на стуктуру. Размер ссылки равен 4 байта. Поэтому (обратите внимание на комментарии в листинге) – я акцентирую внимание на то, что структуры должны быть указаны с директивой **const**. В случае одного параметра (число или строка) данные можно указывать напрямую (потому что в обоих случаях sizeof(parameter) = 4). И промежуточные калбеки в данном случае не нужны! Для индексных свойств и свойств по имени существуют 2 константы: INDEXED_PROPERTY и NAMED_PROPERTY, которые нужно указывать в RegProperty вместо структуры параметров.

```
type
  TSpriteStorage = class(TGameResource)
  private
  . . .
    function GetItem(Index: integer): TSprite;
    function GetSpriteByName(const Name: string): TSprite;
  public
    // сложное индексированное дефолтное свойство
    property Item[Index: integer]: TSprite read GetItem; default;
    // сложное свойство, взять по имени
    property SpriteByName[const Name: string]: TSprite read GetSpriteByName;


// индексированное свойство (и дефолтное)
Lua.RegProperty(TSpriteStorage, 'Item', typeinfo(TSprite),
               @TSpriteStorage.GetItem, nil, INDEXED_PROPERTY{!!!}, true);

// индексированное свойство (и деволтное)
Lua.RegProperty(TSpriteStorage, 'SpriteByName', typeinfo(TSprite),
              @TSpriteStorage.GetSpriteByName,nil,NAMED_PROPERTY{!!!},false);


-- текст в скрипте
local Sprite

Sprite = SpriteStorage.Item[2] --воспользоваться индексным свойством.Индекс 2
Sprite = SpriteStorage.SpriteByName[“MouseSprite”] -- свойство “по имени”
Sprite = SpriteStorage[2] -- дефолтное свойство
```

В заключение хочу сказать, что использование сложных свойств чрезвычайно удобно. Но в плане скорости – пользоваться обычными методами всё же чуть быстрее. Будьте осторожны. Если скорость приоритетна – то лучше воспользоваться методами.

События (TMethod)

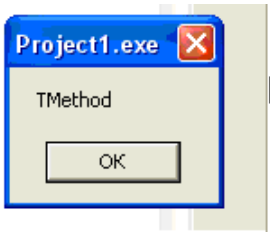
Если честно, я слабо представляю, как в Lua можно использовать события. Возможно пользователям была бы приятна “Delphi-йская” запись:

```
-- допустим в TSprite есть событие OnMove(Sender: TSprite)
-- присваиваем ему метод объекта
-- по логике событий, при каждом движении спрайта
-- будет “возникать событие” – будет вызываться обработчик
Sprite.OnMove = GameEngine.UpdateSpriteOnMove
```

Но в Lua это **невозможно**! Потому что если мы зарегистрировали для TGameEngine метод “UpdateSpriteOnMove”, то запись `GameEngine.UpdateSpriteOnMove` вернёт адрес калбека “TGameEngine.luaUpdateSpriteOnMove”, а не самого метода напрямую. В общем процесс инициализации, присвоения и другого управления событиями в Lua достаточно сложен. Почти невозможен. Я бы оперировал событиями на нативной стороне.

Но есть всё же функционал, который может пригодиться. Начнём с того, что событие по сути своей – это структура TMethod. Если возьмёте sizeof(TNotifyEvent) или любого другого типа-события – то увидите, что он равен 8. Четыре байта уходит на указатель на метод и 4 байта – на “Self”. Т.е. Delphi-йская запись `GameEngine.UpdateSpriteOnMove` вернёт структуру, 4 байта из которых – указатель на TGameEngine.UpdateSpriteOnMove, а оставшиеся 4 байта – конкретный экземпляр класса – GameEngine.

```
end

function TestFunction(Sprite)

 if (Sprite.OnMove ~= nil) then
   ShowMessage(Sprite.OnMove.TypeName)
 end

end
```




```
end; фрагмент System.pas

TMethod = record
  Code, Data: Pointer;
end;

{ TObject.Dispatch accepts any data ty
```

Структура TMethod изначально зарегистрирована в CrystalLUA. О том как регистрировать структуры - мы поговорим позже. Сейчас осветим, как можно работать с TMethod.

Во-первых, TMethod, как и другие структуры имеет поля. TMethod, как и описано в System.pas, имеет 2 поля: Code и Data – оба имеют тип Pointer. TMethod, как и многие структуры – имеет конструктор. В конструкторе указывается сначала Code, потом Data. TMethod, как и другие структуры (в CrystalLUA) можно сравнивать. Например можно сравнить одно событие с другим: ровно ли одно второму. TMethod как структура кардинально отличается от остальных, потому что его можно сравнивать с **`nil`**. Событие же может принимать пустое значение (**`nil`**). Но что самое интересное – события можно вызывать. Наиболее правильно вызывать события через калбеки

```
// калбек-вызов на нативной стороне
procedure luaCallNotifyEvent(const Args: TLuaArgs);
begin
  { первый аргумент – TMethod, второй – Sender }
  TNotifyEvent(Args[0].AsRecord.Data^)(Args[1].AsObject);
end;
```

```
// регистрация "CallNotifyEvent"
Lua.RegProc('CallNotifyEvent', LuaProc(@luaCallNotifyEvent), 2);

{ регистрация события. Которое по сути своей является свойством }
Lua.RegProperty(TSprite, 'OnMove', typeinfo(TNotifyEvent),
                @TSprite(nil).FOnMove, @TSprite(nil).FOnMove);

-- вызов события OnMove из скрипта
-- делается это через специально зарегистрированный калбек, в параметрах
-- которого указывается событие и последующие параметры
if (Sprite.OnMove ~= nil) then
  CallNotifyEvent(Sprite.OnMove, Sprite) -- второй аргумент - это Sender
end
```

Я честно говоря не представляю, зачем может понадобиться "ручной" вызов события. Из Lua. Но если конкретно в вашем случае такая необходимость возникла – вы знаете – нужно зарегистрировать специальную функцию и передать все параметры в неё.

Существует и второй, более удобный, но менее безопасный вызов событий. Если событие имеет не больше 2х параметров, если каждый из аргументов укладывается в 4 байта (не single), если структуры/массивы/множества указаны со ссылочной директивой (**const**{!!!}, **var**, **out**), и нет ссылочных директив для обычных типов, то событие (TMethod) можно использовать как обычный метод:

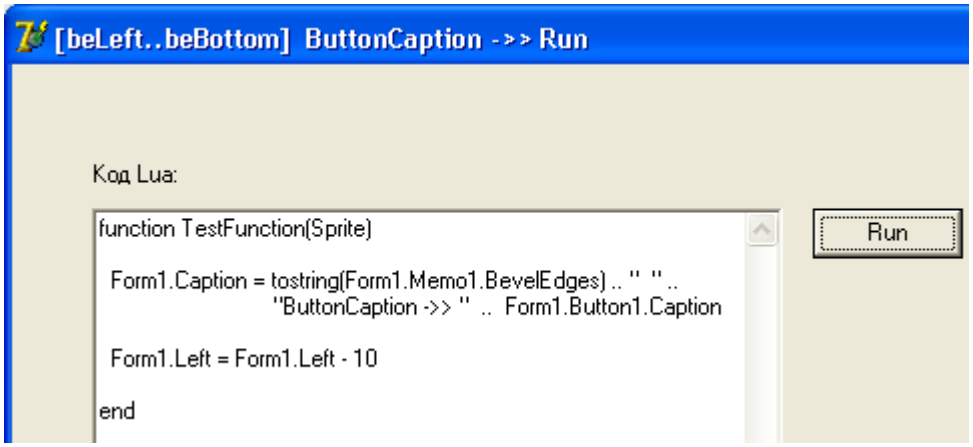
```
  -- пример как из кода Lua можно вызывать событие как метод
  -- событие должно быть простым.
  if (Sprite.OnMove ~= nil) then
    Sprite.OnMove(Sprite) -- в качестве Sender - Sprite
  end

// события которые подходят для простого вызова
TNotifyEvent = procedure(Sender: TObject) of object;
TExceptionEvent = procedure(Sender: TObject; E: Exception) of object;
THashEvent = procedure(HashArray: PHashArray; HashItem: PHashItem) of object;
TOnSpinButtonClick=procedure(Sender: TGameSpinButton; Up: boolean) of object;

// не подходит - параметров больше 2х
TMouseEvent = procedure(Sender: TObject; Button: TMouseButton;
              Shift: TShiftState; X, Y: Integer) of object;

// не подходит - Key указан с директивой var
TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;

// не совсем подходит - стандартный вызов не предусматривает результат
// хотя структура с директивой var допустима
TWindowHook = function (var Message: TMessage): Boolean of object;
```

Published и авторегистраторы

Настало время поговорить о самой “вкусной” части информации о классах. До сих пор считалось, что Lua будет знать о классе в точности то, что вы в нём явно зарегистрируете. Но Delphi обладает рядом преимуществ по сравнению с другими языками, такими как например RTTI. К сожалению RTTI не такое подробное, как лично мне хотелось бы. Но часть задач он существенно облегчит. Чтобы долго не теоретизировать, я лучше сразу покажу основные особенности.

```
// две строчки, которые делают Form1: TForm1 доступной в Lua,
// при этом автоматически регистрируются все published свойства и события,
// а так же published-поля (список компонентов на форме).
// полностью регистрируются классы published-полей,
// используемые множества и перечисляемые константы

Lua.RegClass(TForm1, {use_published по умолчанию true} true);
Lua.RegVariable('Form1', Form1, typeinfo(TForm1){можно typeinfo(TObject)});
```

Иными словами, почти всё что можно выжать из RTTI мы выжимаем, установив флаг use_published. Попробуйте следующий код:



В это сложно поверить, но 2мя строками кода вы автоматически регистрируете такие свойства как Caption, Text, Name, Left, Top, Width, Height, BorderIcons, Position – в общем всё то, что доступно из Object Inspector. Правда здорово ?

Теперь осталось показать, как этим пользоваться в жизни. Ясное дело, компонентами в Lua вы будете пользоваться нечасто. Под скриптование обычно попадут классы типа нашего TSprite. На его примере я и намерен показать возможности published подхода.

```
{$M+}                  { -- Пример published регистрации -- }
  TSprite = class(TGameObject)
  private
    FX, FY: integer;
  public
    procedure Move(AX, AY: integer);
  published // доступны как обычные поля, так и сеттеры/геттеры (+ virtual)
    property X: integer read FX write FX;
    property Y: integer read GetY write SetY;
  end; {$M-}
```

Хочу обратить особое внимание на то, появилась директива `{$M+}`. Она указывает, что для классов указанных ниже нужно создавать RTTI. Обычно эта директива отключена, поэтому даже если вы переместили все свои свойства в секцию published – автоматически регистрироваться они не будут. По правилам формирования RTTI для классов, если RTTI генерируется для класса X, то оно так же будет генерироваться и для всех его потомков. С помощью этой особенности вы смогли зарегистрировать TForm1 (так же как и любой другой компонент) – только потому что их общий родитель TPersistent обладает RTTI.

```
{$M+} // фрагмент модуля Classes.pas
  TPersistent = class(TObject)
  . . .
  end;
{$M-}
```

Надеюсь, я приятно вас удивлю, сказав, что published-подход позволяет регистрировать не только свойства, но и методы. Правда это значительно сложней. Для того чтобы автоматически регистрировать классовые методы в CrystalLUA, класс должен содержать published-метод, начинающийся с ‘lua’. Например, чтобы автоматически зарегистрировать метод TSprite.Move(), в TSprite должен находиться published-метод luaMove().

```
{$M+}
  TSprite = class(TGameObject)
  . . .
  published    // наличие этого LuaClassProc метода в секции published
    . . .      // позволяет автоматически зарегистрировать метод Move()
    procedure luaMove(const Args: TLuaArgs);
  end; {$M-}
```

Надеюсь вы согласить со мной, что городить published-методы, пихать в секцию published свойства и постоянно следить, генерируется ли по классу RTTI – не всегда удобно. Поэтому в CrystalLUA введена так же концепция авторегистраторов. Авторегистратор – это отдельный класс, наследуемый от регистрируемого и начинающийся с ‘lua’. Авторегистраторы удобны тем, что позволяют отделить “регистрацию класса” от самого функционала класса. Например для класса TSprite это будет выглядеть так:

```
type
  // обычная реализация класса. без {$M+}
  TSprite = class(TGameObject)
  private
    FX, FY: integer;
  public
    procedure Move(AX, AY: integer);

    property X: integer read FX write FX;
    property Y: integer read GetY write SetY;
  end;
```

```
type
  {$M+} // класс авторегистратор
  luaTSprite = class(TSprite)
  published
    // авторегистрируемые методы
    procedure luaMove(const Args: TLu

    // авторегистрируемые свойства
    property X;
    property Y;
  end; {$M-}
```

Просто зарегистрировав класс luaTSprite (use_published не важен), вы зарегистрируете класс TSprite (а не luaTSprite), но он будет обладать свойствами X, Y и методом Move. Подход с авторегистраторами позволяет легко и просто регистрировать классы, свойства и их методы, разделяя при этом обычный функциональный код и код, связанный с Lua.

Наслаждайтесь !

Регистрация структур

Структура – второй по важности сложный объект в CrystalLUA. Ещё неизвестно: с чем вам чаще придётся работать, со структурами или с классами. Всё зависит от задачи. В качестве теста будем рассматривать 2 простые и частоиспользуемые структуры:

```
type
  TPoint = record
    X: integer;
    Y: integer;
  end;

  TRect = packed record
    case Integer of
      0: (Left, Top, Right, Bottom: integer);
      1: (TopLeft, BottomRight: TPoint);
  end;
```

PLuaRecordInfo

Сложно сказать, чем обусловлено создание PLuaRecordInfo. Наверное каждый сложный объект нужно однозначно идентифицировать – к какому “классу” он относится. У классов всё просто – принадлежность экземпляра класса к тому или иному классу определяется элементарно. PLuaRecordInfo содержит всю важную информацию по структуре: поля, их типы, typeinfo и т.д. Для того чтобы зарегистрировать PLuaRecordInfo в CrystalLUA, нужно вызвать функцию

```
// регистрация структуры
TLua.RegRecord(const Name: string; const tpinfo: ptypeinfo): PLuaRecordInfo;
```

В качестве первого аргумента принимается имя структуры – в нашем случае “TPoint” или “TRect”. В качестве второго аргумента принимается typeinfo структуры. Здесь, думаю, надо сказать отдельное слово. TypeInfo – это набор RTTI данных, необходимых в основном для внутренних нужд. Для инициализации например или для финализации (удалении после того как переменная потеряла свою актуальность). К сложным данным (которые в первую очередь учитываются в RTTI) относят строки, динамические массивы, Variant-ы, Interface-ы, структуры(содержащие сложные данные) или статические массивы, которые тоже содержат сложные данные. В Delphi typeinfo для структур или статических массивов, не содержащих сложных данных – вообще не создаётся. Иными словами для наших TPoint и TRect запись typeinfo(тип) вызовет ошибку компилятора “Type ‘TRect’ has no type info”. Если бы TRect имела например такой вид, то typeinfo для неё 100% бы был.

```
  TRect = packed record
    S: string; // любой сложный тип делает из "простой" структуры "сложную"
    case Integer of
      0: (Left, Top, Right, Bottom: integer);
      1: (TopLeft, BottomRight: TPoint);
  end;
```

Если конкретно в вашей структуре typeinfo есть, то указывать её необходимо обязательно! Это нужно потому, что CrystalLUA работает по всем правилам RTTI – создаёт и удаляет объекты, инициализирует их и финализирует, инкрементирует ссылки и декрементирует. Если вы не будете указывать typeinfo, то в лучшем случае это обернётся утечкой памяти. В худшем (и это более вероятно) - EAccessViolation.

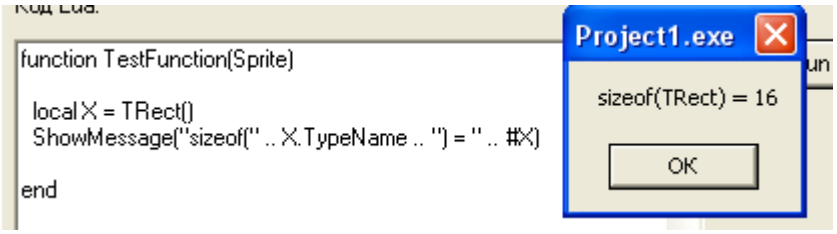
Теперь давайте разберёмся со случаями, когда RTTI для структур не создаётся. В таких случаях указывайте sizeof структуры – CrystalLUA правильно поймёт второй параметр и правильно всё зарегистрирует.

```
var
  PointInfo, RectInfo: PLuaRecordInfo;
begin
  PointInfo := RegRecord('TPoint', pointer(sizeof(TPoint)));
  RectInfo := RegRecord('TRect', pointer(sizeof(TRect)));
```

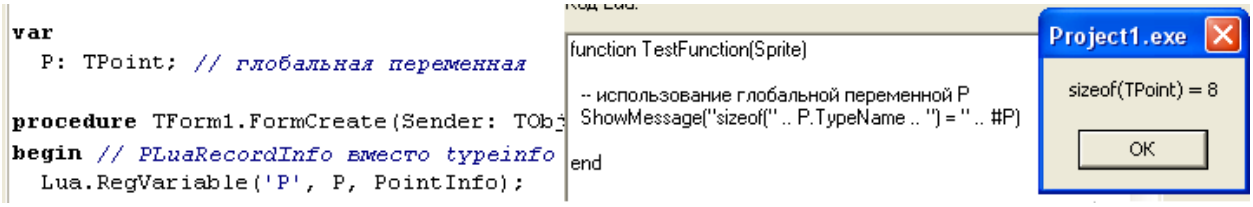
Есть ещё и третий вариант как указать typeinfo структуры. Дело в том, что динамические массивы всегда имеют typeinfo. Анализируя RTTI по динамическому массиву можно узнать информацию об элементе такого массива. Пользуйтесь подобным случаем, если на момент регистрации не знаете: будет ли структура иметь typeinfo или нет.

```
type
  TRectDynArray = array of TRect;
var
  RectInfo: PLuaRecordInfo;
begin  // такой подход позволяет всегда видеть нужную информацию по структуре
  RectInfo := RegRecord('TRect', typeinfo(TRectDynArray));
```

Зарегистрировав структуру в Lua (указав в качестве typeinfo один из трёх вариантов), вы получаете возможность работать со структурами внутри Lua.



Забыл сказать. Для всех сложных объектов действуют оператор взятия размера (#) и функция преобразования к строке (tostring). Ещё вам следует знать, что обладая PLuaRecordInfo, вы можете регистрировать глобальные переменные или свойства.



Если на конкретный момент вы не обладаете PLuaRecordInfo, то вы можете его найти:

```
// public свойство TLua, позволяющее найти Info по структуре
property RecordInfo[const Name: string]: PLuaRecordInfo read GetRecordInfo;
```

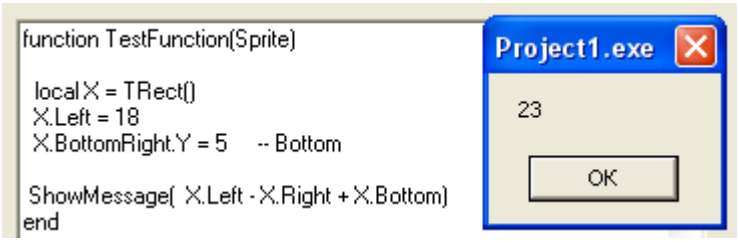
Регистрация полей

Это конечно здорово – обладать зарегистрированной структурой, но всё же главное в ней – это поля. Поля регистрируются через интерфейс PLuaRecordInfo:

```
// регистрация по смещению
procedure RegField(const FieldName: string; const FieldOffset: integer;
          const tpinfo: pointer); overload;

// регистрация по "указателю"
procedure RegField(const FieldName: string; const FieldPointer: pointer;
          const tpinfo: pointer; const pRecord: pointer = nil); overload;
```

К примеру структуру TRect (по смещению) можно зарегистрировать так:

```
with RectInfo^ do
begin
  RegField('Left', 0, typeinfo(integer));
  RegField('Top', 4, typeinfo(integer));
  RegField('Right', 8, typeinfo(integer));
  RegField('Bottom', 12, typeinfo(integer));
  RegField('TopLeft', 0, PointInfo); // поле-структура
  RegField('BottomRight', 8, PointInfo); // поле-структура
end;
```



Регистрировать поля таким образом можно, но не безопасно. Во-первых, можно ошибиться со смещением, во-вторых, при модификации кода поля могут как добавляться, так удаляться, так и перемещаться. Поэтому наиболее правильно регистрировать поля "по указателю".

```
with TRect(nil^) do // <-- обратите внимание на запись
begin
  RectInfo.RegField('Left', @Left, typeinfo(integer));
  RectInfo.RegField('Top', @Top, typeinfo(integer));
  RectInfo.RegField('Right', @Right, typeinfo(integer));
  RectInfo.RegField('Bottom', @Bottom, typeinfo(integer));
  RectInfo.RegField('TopLeft', @TopLeft, PointInfo);
  RectInfo.RegField('BottomRight', @BottomRight, PointInfo);
end;
```

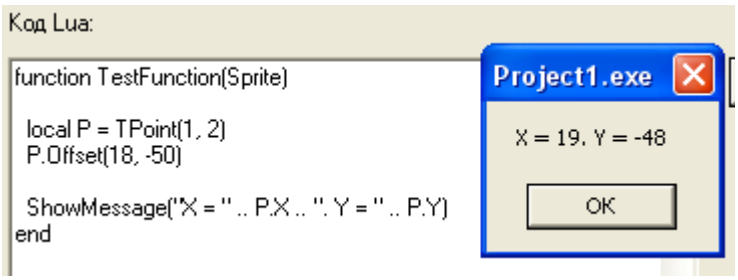
Регистрация методов

До недавнего времени язык Delphi не позволял методов в структуре. Методы и свойства мог содержать **object** (который вы кстати тоже можете использовать в RegRecord). Я долго думал над наполнением структур и пришёл к выводу, что структуры в Lua будут содержать поля и методы, свойств не будет. Метод добавляется с помощью

```
// регистрация метода в структуре или object-е
procedure TLuaRecord.RegProc(const ProcName:string; const Proc:TLuaClassProc;
                                    const ArgsCount: integer=-1);
```

Для демонстрации регистрации методов, я предлагаю зарегистрировать метод Offset и конструктор для типа TPoint.

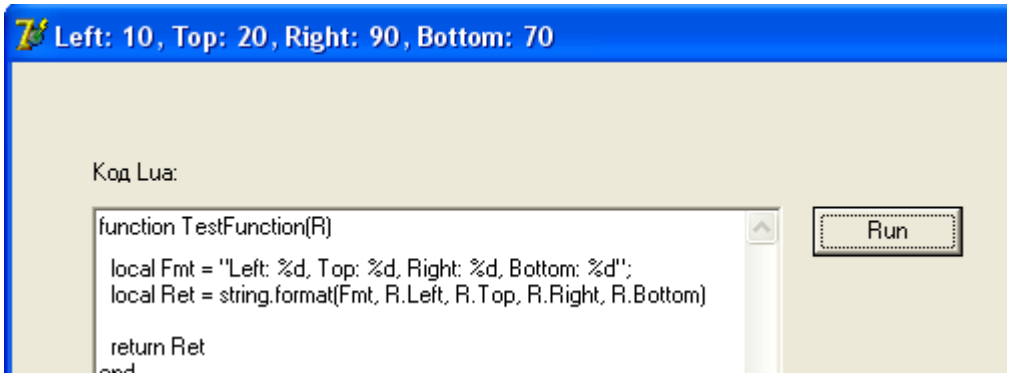
```
// TPoint.Offset(X, Y)
procedure TPointOffset(var X; const Args: TLuaArgs);
var
  P: TPoint absolute X;
begin
  P.X := P.X + Args[0].AsInteger;
  P.Y := P.Y + Args[1].AsInteger;
end;

// TPoint(X, Y) - конструктор
procedure TPointConstructor(var X; const Args: TLuaArgs);
var
  P: TPoint absolute X;
begin
  P.X := Args[0].AsInteger;
  P.Y := Args[1].AsInteger;
end;

// регистрация
P.RegProc('Offset', LuaClassProc(TPointOffset));
P.RegProc(LUA_CONSTRUCTOR, LuaClassProc(TPointConstructor));
```



TLuaRecord

Если по экземпляру класса всегда видно, к какому он классу принадлежит, то со структурами такой роскоши нет. Для структур в CrystalLUA есть TLuaRecord, который в первую очередь представляет собой указатель на структуру и PLuaRecordInfo.

```
procedure ShowRect(const R: TRect);
var
  Args: TLuaArgs;
  LuaRecord: TLuaRecord;
begin
  ZeroMemory(@LuaRecord, sizeof(LuaRecord));
  LuaRecord.Data := @R;
  LuaRecord.Info := Lua.RecordInfo['TRect'];

  SetLength(Args, 1);
  Args[0].AsRecord := LuaRecord;

  Form1.Caption := Lua.Call('TestFunction', Args).ForceString;
end;
. . .

ShowRect(Bounds(10, 20, 80, 50));
```



Данным примером я хотел показать, как отправлять структуру в Lua. Присваиваете аргументу (TLuaArg) параметры передаваемой структуры (TLuaRecord: указатель и Info) и всё. Аналогичным образом структуру из Lua можно передать на нативную сторону. В этом случае придёт TLuaArg, LuaType которого будет ltRecord, а свойство AsRecord вернёт указатель на структуру и его PLuaRecordInfo. Кроме того что поля TLuaRecord можно заполнять вручную, существует так же функция, облегчающая задачу:

```
// функция, упрощающая создание TLuaRecord
function LuaRecord(const Data: pointer; const Info: PLuaRecordInfo;
                   const IsRef: boolean=true; const IsConst: boolean=false)
                                                          : TLuaRecord;
```

Что такое IsRef и IsConst – мы рассмотрим прямо сейчас.

IsRef и IsConst

Понятия “IsRef” и “IsConst” будут встречаться часто. Они относятся ко всем сложным объектам и входят в «Стандартные свойства». Флаг IsRef показывает, в Lua будет использоваться ссылка на объект или полная его копия. Если отправляя структуру в Lua, вы укажете флаг IsRef, тогда в Lua вы будете оперировать не самими данными, а только ссылкой на них. Если укажите False, то в Lua будет создана отдельная структура-копия, которая удалится при GC. Использование дублированных данных не рекомендуется. Особенно когда в этом нет абсолютно никакой необходимости.

IsConst показывает, является ли объект “только для чтения”. Иными словами если вы попытаетесь изменить что-то в IsConst структуре – то получите Exception.

Структура в качестве результата функции

Особую сложность представляют задачи, когда нужно добавить в Lua функцию, возвращающую структуру. Ну возьмём к примеру функцию Bounds()

```
// функция из модуля Types.pas
function Bounds(ALeft, ATop, AWidth, AHeight: Integer): TRect;

// калбек для функции мог бы выглядеть так
function luaBounds(const Args: TLuaArgs): TLuaArg;
var
  R: TRect; // результат хранится на стеке !
begin
  R := Bounds(Args[0].AsInteger, Args[1].AsInteger, ... , Args[3].AsInteger);
  Result.AsRecord := LuaRecord(@R, Lua.RecordInfo['TRect'], {!!!} True);
end;
```

Проблема в том, что какой бы вы флаг IsRef не выставляли, если результат функции храниться в стеке, то дальнейшие манипуляции библиотеки это значение затрут. Специально для случаев, когда в калбеке нужно вернуть сложный объект, в TLua есть глобальный объект ResultBuffer. Правильно калбек luaBounds будет написать так

```
// правильный калбек luaBounds, учитывая ResultBuffer
function luaBounds(const Args: TLuaArgs): TLuaArg;
var
  R: ^TRect; // указатель, который будет указывать на буфер
begin
  R := Lua.ResultBuffer.AllocRecord(RecordInfo {Lua.RecordInfo['TRect']});
  R^ := Bounds(Args[0].AsInteger, Args[1].AsInteger, ..., Args[3].AsInteger);
  Result.AsRecord := LuaRecord(R, RecordInfo, False {в Lua - копия});
end;
```

Операторы

Структуры отличаются от других сложных объектов тем, что для них можно определить операторы. К операторам в CrystalLUA относят унарный минус(-), сложить(+), вычесть(-), умножить(*), разделить(/), узнать остаток от деления(%), возвести в степень(^), сравнить (<, >, ==, ~=, <=, >=). Для операторов в CrystalLUA нужно знать следующие 3 типа:

```
// типы, важные для операторов
TLuaOperator = (loNeg, loAdd, loSub, loMul, loDiv, loMod, loPow, loCompare);
TLuaOperators = set of TLuaOperator;
TLuaOperatorCallback=procedure(var _Result,_X1,_X2; const Kind:TLuaOperator);
```

Для того чтобы зарегистрировать операторы для структуры, необходимо заполнить 2 поля PLuaRecordInfo:

```
property Operators: TLuaOperators read/write
property OperatorCallback: TLuaOperatorCallback read/write
```

Operators – это список операторов, которые должна поддерживать структура. Если хотите зарегистрировать все операторы, то можете воспользоваться константой `ALL_OPERATORS`. OperatorCallback – это функция, которой и стоит проделать действия по оператору. Параметры _Result, _X1 и _X2 принимают разные значения - в зависимости от оператора.

`loNeg`:

_Result – это результирующая структура, _X1 = _X2 – структура, от которой нужно взять отрицательное значение (унарный минус)

`loAdd, loSub`:

_Result – это результирующая структура, _X1 и _X2 – это структуры, слагаемые (для сложения) или уменьшаемое/вычитаемое (для вычитания)

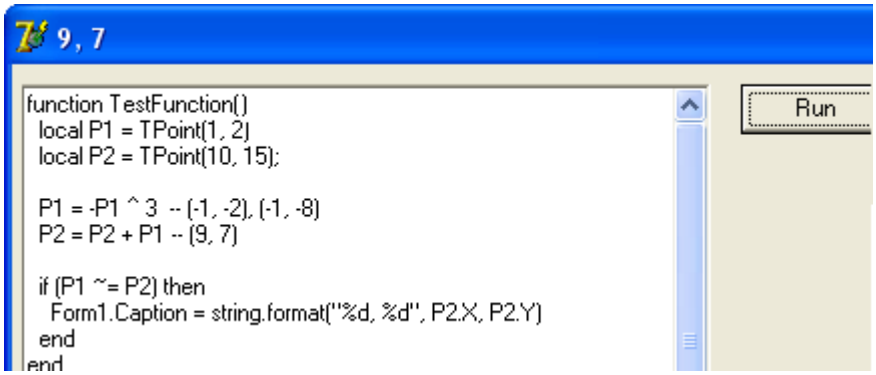
`loMul, loDiv, loMod, loPow`:

_Result – это результирующая структура, _X1 – это исходная структура, _X2 – Double, на который нужно умножить, разделить, взять остаток или возвести в степень.

`loCompare`:

_Result – это integer, который принимает значения -1(_X1<X2),+1(_X1>_X2) и 0(_X1= X2)

В качестве примера использования и реализации операторов я возьму TPoint.



```
function IntFmod(const X: integer; const Value: double): integer;
begin Result := X - round(Floor(X/Value)*Value); end;

procedure TPointOperator(var _Result, _X1, _X2; const Kind: TLuaOperator);
const SIGNS: array[boolean] of integer = (-1, 1);
var
  Result: TPoint absolute _Result;
  ResultInt: integer absolute _Result;
  P1: TPoint absolute _X1;
  P2: TPoint absolute _X2;
  DoubleValue: double absolute _X2;
begin
  case (Kind) of
    loNeg: begin Result.X := -P1.X; Result.Y := -P1.Y; end;
    loAdd: begin
             Result.X := P1.X+P2.X;
             Result.Y := P1.Y+P2.Y;
           end;
    loSub: begin
             Result.X := P1.X-P2.X;
             Result.Y := P1.Y-P2.Y;
           end;
    loMul: begin
             Result.X := round(P1.X * DoubleValue);
             Result.Y := round(P1.Y * DoubleValue);
           end;
    loDiv: begin
             Result.X := round(P1.X / DoubleValue);
             Result.Y := round(P1.Y / DoubleValue);
           end;
    loMod: begin
             Result.X := IntFmod(P1.X, DoubleValue);
             Result.Y := IntFmod(P1.Y, DoubleValue);
           end;
    loPow: begin
             Result.X := round(Power(P1.X, DoubleValue));
             Result.Y := round(Power(P1.Y, DoubleValue));
           end;
  else
    // loCompare
    if (P1.X <> P2.X) then ResultInt := SIGNS[P1.X > P2.X]
    else
    if (P1.Y <> P2.Y) then ResultInt := SIGNS[P1.Y > P2.Y]
    else
    ResultInt := 0;
  end;
end;
```

Регистрация массивов

Работа с массивами во многом похожа на структуры. Например их тоже нужно регистрировать. Для них тоже актуальна концепция указатель-Info. У них тоже есть свойства IsRef и IsConst. Для результатов функций тоже нужно задействовать ResultBufer.

PLuaArrayInfo

Для регистрации массивов нужно использовать функцию

```
function  RegArray(const Identifier: pointer; const itemtypeinfo: pointer;
                   const Bounds: array of integer): PLuaArrayInfo;
```

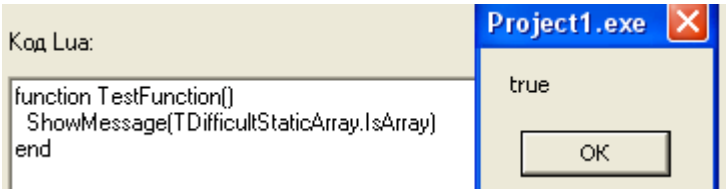
Если “потеряете” PLuaArrayInfo, можно будет получить его с помощью свойства

```
property ArrayInfo[const Name: string]: PLuaArrayInfo read GetArrayInfo;
```

Но я настоятельно рекомендую хранить PLuaArrayInfo в отдельной переменной, а не получать его каждый раз при помощи свойства ArrayInfo. Хоть ArrayInfo и использует быстрые хешированные поиски, но это всёравно трудозатраное действие.

Теперь подробнее с параметрами функции RegArray. Массивы бывают статическими и динамическими. Для динамических массивов Identifier – typeinfo массива. Для статических – typeinfo или pchar(Имя). Допускаются как одномерные массивы, так и многомерные. itemtypeinfo – это typeinfo для обычных элементов или “Handle” для сложных. Bounds – имеет значение только для статических массивов, для динамических он должен быть равен [].

```
type
  TIntegerDynArray = array of integer;
  TIntegerStaticArray = array[0..9] of integer;

  // многомерные массивы (3)
  TDifficultDynArray = array of array of array of string;
  TDifficultStaticArray = array[0..2, 1..5, 15..18] of string;

// регистрация
Lua.RegArray(typeinfo(TIntegerDynArray), typeinfo(integer), []);
Lua.RegArray(pchar('TIntegerStaticArray'), typeinfo(integer), [0,9]);
Lua.RegArray(typeinfo(TDifficultDynArray), typeinfo(string), []);
// можно pchar(), а можно и через typeinfo
Lua.RegArray(pchar('TDifficultStaticArray'),typeinfo(string),[0,2,1,5,15,18])
```

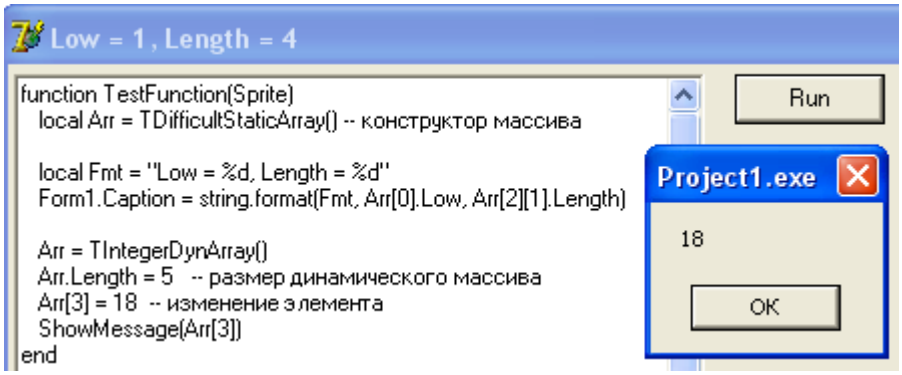


TLuaArray

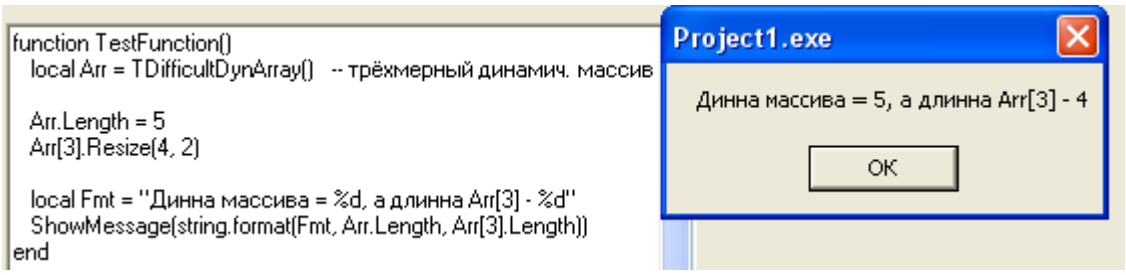
Собственно структура TLuaArray ничем принципиально не отличается от структуры TLuaRecord. Всё то же самое. Тот же Data – указатель на массив(**важно!** Data в любом случае – указатель на массив; хоть на статический, хоть на динамический). Всё тот же Info – только не PLuaRecord, а PLuaArrayInfo. Если вы регистрируете функцию, которая в качестве результата возвращает массив, то берите указать не ResultBuffer.AllocRecord(…), а ResultBuffer.AllocArray(const ArrayInfo: PLuaArrayInfo). Всё так же актуальны свойства IsRef и IsConst. А так же все остальные «Стандартные свойства».

Свойства Low, High, Length. Метод Resize()

Как и в Delphi, в CrystalLUA для массивов доступны свойства Low, High, Length. Только в Delphi это функции, а в CrystalLUA – они свойства. Изменение свойства Length доступно только для динамических массивов. Low, High и Length доступны как для экземпляров массивов, так и для их “типов”.



Метод Resize() доступен только для динамических массивов. По сути он является многократным изменением Length, только значительно круче ☺. Изменять размер можно как самого массива, так и подмассивов (для многомерных массивов).

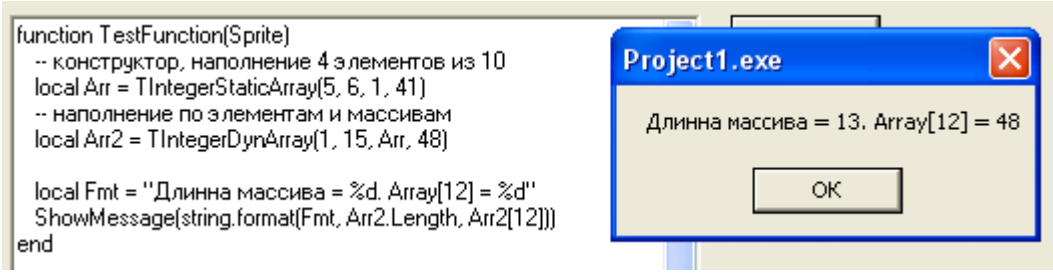


Надеюсь вы понимаете как работают динамические массивы, и понимаете что строки

```
Arr.Length = 3
Arr[0].Resize(4, 2)
Arr[1].Resize(4, 2)
Arr[2].Resize(4, 2)

-- можно заменить на:
Arr.Resize(3, 4, 2)
```

Конструкторы

До сих пор мы вызывали конструкторы массивов без параметров. Но параметры можно задавать. Конструкторы массивов в CrystalLUA работают в режиме наполнения. Наполнение работает только для одномерных массивов: и статических, и динамических. В качестве аргументов конструкторов принимаются как элементы массива, так и другие одномерные массивы таких элементов.



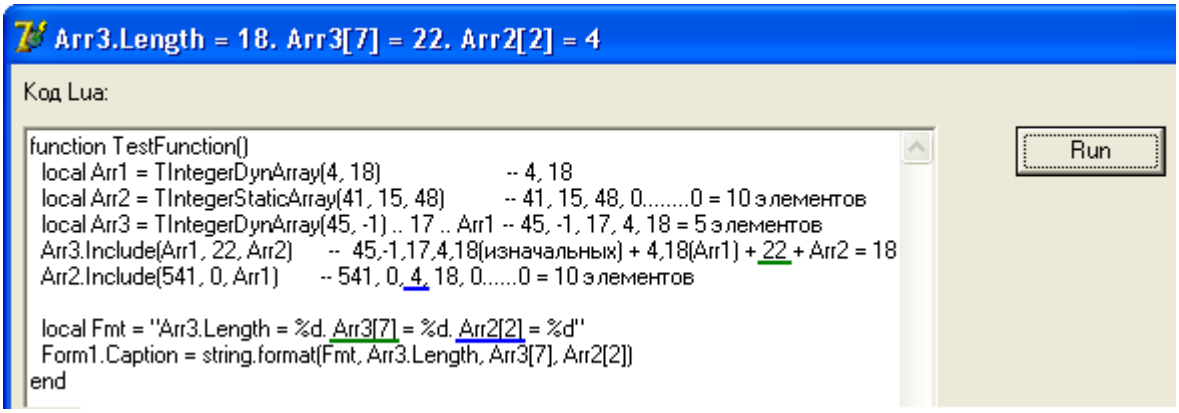
Метод Include()

Метод Include() по действию полностью похож на наполнение в конструкторе. Отличие только в том, что для динамических массивов Include() добавляет элементы в конец массива, а для статических - каждый Include() наполняет сначала.

Конкатенация динамических массивов

Для одномерных динамических массивов в CrystalLUA определён оператор конкатенации (“..”). Я не рекомендую пользоваться этим оператором больше чем для двух операндов – намного лучше воспользоваться методом Include().

Надеюсь на данном сложном примере вам удастся понять, как работают конструкторы, Include() и оператор конкатенации



Регистрация множеств

Множества – самый простой тип из сложных. Он всегда имеет typeinfo, в typeinfo есть все константы, которые он использует. Для множеств актуально всё, что актуально для структур и массивов, и по сути, для работы с множествами нужны всего 3 функции

```
// регистрация множества
function  TLua.RegSet(const tpinfo: ptypeinfo): PLuaSetInfo;

// нахождение PLuaSetInfo
property TLua.SetInfo[const Name: string]: PLuaSetInfo read GetSetInfo;

// создать TLuaSet
function LuaSet(const Data: pointer; const Info: PLuaSetInfo;
    const IsRef: boolean=true; const IsConst: boolean=false): TLuaSet;
```

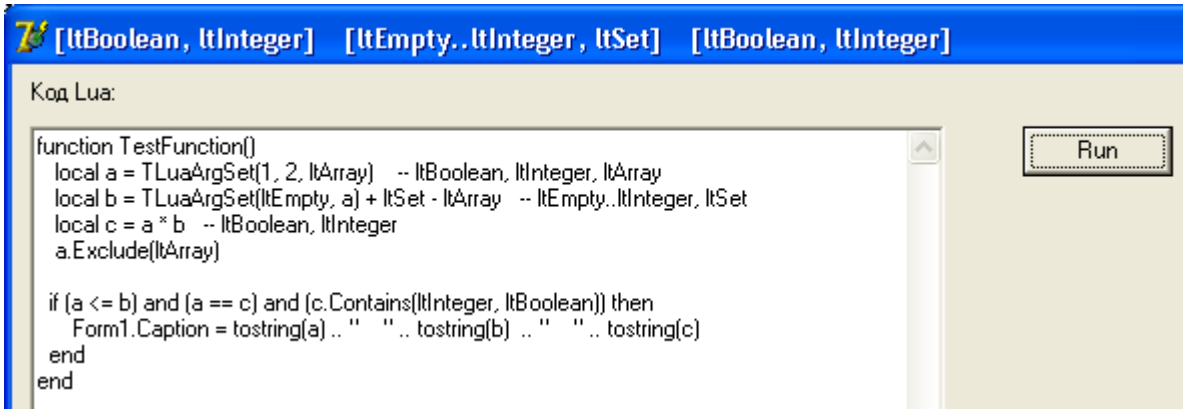
Конструктор, Include(), Exclude(), Contains()

Конструкторы и метод Include(), как и для массивов работают в режиме наполнения. В качестве аргументов принимаются как элементы множества, так и сами множества. Exclude работает наоборот – над исключением элементов или множеств из множества. Функция Contains() определяет, содержит ли множество элементы или множества.

Операторы

В CrystalLUA над множествами применимы ровно те операторы, которые применимы в Delphi. Это объединение(+), различия(-), пересечение(*), сравнение на равенство (==, ~=) и определение, является ли одно множество подмножеством другого (<=, >=). Тем не менее я не рекомендовал бы использовать операторы больше чем для 2х операндов.

```
type
{TLuaArgType = (ltEmpty, ltBoolean, ltInteger, ltDouble, ltString, ltPointer,
                ltClass, ltObject, ltRecord, ltArray, ltSet, ltTable);      }
TLuaArgSet = set of TLuaArgType;
```



Таблицы и ссылки

Настало время поговорить о вещах, специфичных для Lua. Ссылки – это средства, помогающие существенно увеличить скорость доступа к глобальным объектам(о ссылках мы поговорим в конце). Таблицы – это вообще самое распространённое, что используется в Lua. Таблицы используются как массивы, как ассоциативный массив (ключ, значение), как словарь и т.д. Даже глобальные переменные – и те хранятся в таблице. Создавать Lua-таблицы на нативной стороне можно, но CrystalLUA такого не позволяет. Но вы всегда можете пользоваться таблицами, созданными на стороне Lua. Можно как брать значение в таблице, так и изменять значения. Если вы посмотрите TLuaArg, то увидите, что AsTable – это единственное свойство “readonly”

```
public
  . . .
  property AsArray: TLuaArray read GetArray write SetArray;
  property AsSet: TLuaSet read GetSet write SetSet;
  property AsTable: PLuaTable read GetTable; // Lua таблица
```

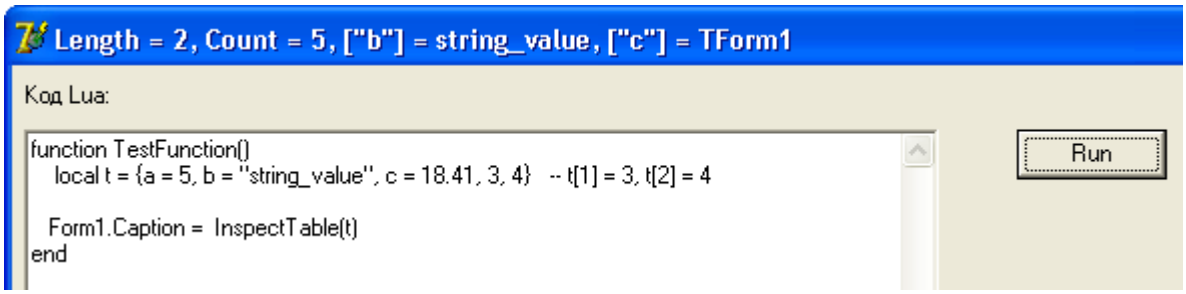
Свойства PLuaTable

```
type
  TLuaTable = object
  . . .
    // длинна
    property Length: integer read GetLength; // длинна (для массивов)
    property Count: integer read GetCount; // общее количество элементов

    // значения
    property Value[const Index: integer]: Variant read/write
    property KeyValue[const Key: string]: Variant read/write
    property ValueEx[const Index: integer]: TLuaArg read/write
    property KeyValueEx[const Key: Variant]: TLuaArg read/write
  end;
```

Если использовать таблицу как массив (т.е. в качестве ключа указывать целочисленное значение), то работа с таблицей будет значительно быстрее. Свойство Length возвращает максимальный целочисленный индекс и эквивалентен Lua-оператору “#”. Однако общее количество элементов может отличаться и для этого существует Count – свойство, которое в обработчике перебирает все элементы, поэтому вызывать это свойство не советую. Значения таблицы делятся на Value (обращение к индексу массива) и KeyValue (значение по ключу). Есть обычные варианты (Value[Index], KeyValue[Key]) и продвинутые (Ex) – использующие в качестве аргумента TLuaArg.

Предлагаю рассмотреть возможности взаимодействия нативного когда с Lua-таблицей на следующем примере. Регистрируем калбек и исследуем таблицу, переданную в качестве параметра.

```
// калбек, принимающий таблицу в качестве аргумента
function luaInspectTable(const Arg: TLuaArg): TLuaArg;
var
  Table: PLuaTable;
  BValue: string;
begin
  Table := Arg.ForceTable;

  BValue := Table.KeyValue['b'];
  Table.KeyValueEx['c'] := LuaArg(TForm1);

  Result.AsString := Format('Length = %d, Count = %d, ["b"] = %s, ["c"] = '+
  '%s', [Table.Length,Table.Count,BValue,Table.KeyValueEx['c'].ForceString]);
end;
```



Перебор всех элементов

Иногда возникает необходимость перебрать все элементы в таблице. В CrystalLUA за это отвечают интерфейс TLuaPair и две функции PLuaTable:

```
function Pairs(var Pair: TLuaPair): boolean; overload;
function Pairs(var Pair: TLuaPair;const FromKey: Variant):boolean;overload;
```

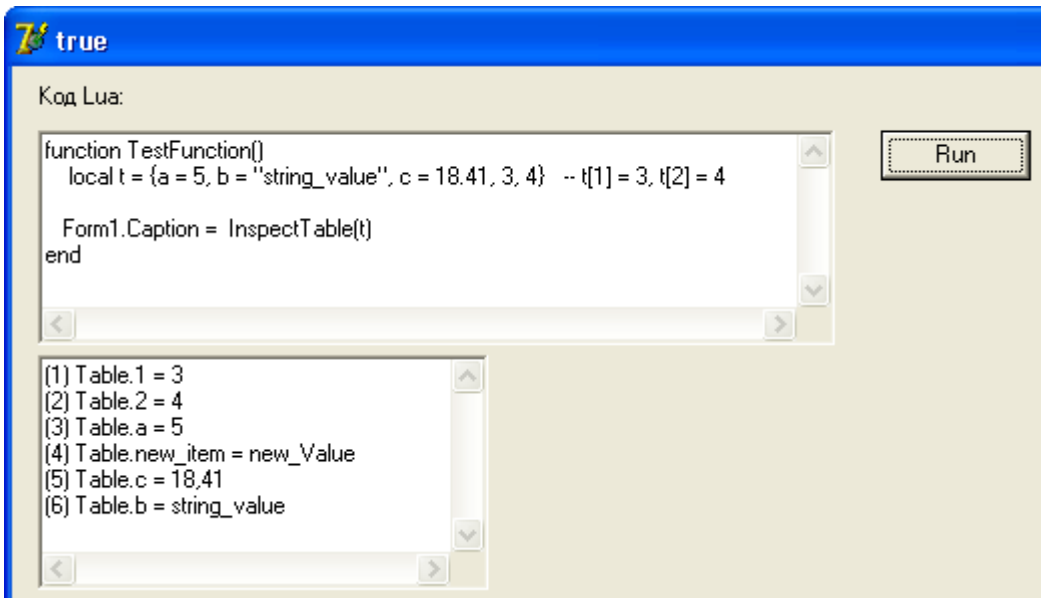
Функции PLuaTable.Pairs() позволяют начать перебор таблицы сначала или с конкретного ключа. Хочу обратить ваше внимание на то, что элементы, значения которых **nil** – в таблице не хранятся! Поэтому если вы вдруг захотите найти экземпляр TSprite, а он был равен **nil**, то в таблице он не хранится!

Pairs() использует Lua-стек, поэтому вы обязаны либо просмотреть таблицу до конца, либо насильно вызвать TLuaPair.Break(). Интерфейс TLuaPair:

```
TLuaPair = object
public
  function  Next(): boolean; // следующая итерация перебора
  procedure Break(); // насильно прекратить перебор
  property Iteration: integer read FIteration; // номер текущей итерации
  property Broken: boolean read GetBroken; // остановлен ли перебор
  property Key: string read GetKey; // текущий ключ
  property KeyEx: Variant read GetKeyEx; // текущий ключ
  property Value: Variant read GetValue write SetValue; // значение
  property ValueEx: TLuaArg read GetValueEx write SetValueEx; // значение
end;
```

Перебор элементов таблицы может выглядеть примерно так

```
// калбек, принимающий таблицу в качестве аргумента
function luaInspectTable(const Arg: TLuaArg): TLuaArg;
var
  Table: PLuaTable;
  Pair: TLuaPair;
begin
  Table := Arg.ForceTable;

  Table.KeyValue['new_item'] := 'new_Value'; // новое значение
  Form1.Memo2.Lines.Clear;

  if (Table.Pairs(Pair{, ключ})) then
  repeat
    { if (Something) then Pairs.Break(); }

    Form1.Memo2.Lines.Add(Format('(%d) Table.%s = %s',
                         [Pair.Iteration, Pair.Key, Pair.Value]));

    {  Pair.Value доступен для изменения !  }
  until not Pair.Next();

  Result.AsBoolean := Pair.Broken;
end;
```



Ссылки (TLuaReference)

Честно говоря, я не так часто пользуюсь Lua. И когда я спросил своего знакомого зачем нужны ссылки в Lua, он ответил следующее:

lua_ref нужно чтобы самому не городить глобальные массивы для хранения чего-нибудь на стороне луа, и для быстрого доступа к хранимому. Константные стринги на нативной стороне быстрее и удобнее хранить в реестре, чем каждый раз делать pushstring. Еще есть вещи, которые могут понадобится позже, но gc может стуктуть их, если на них не осталось ни одной ссылки. Для таких объектов нужно хранить копию на стороне луа. Да, можно велосипедить, никто не мешает. Но зачем, если lua_ref уже есть, и пашет немного быстрее, чем самописный велик, юзающий стандартные механизмы луа? Вообще нахрена массивы объектов на нативной стороне? Можно же конструировать их прямо в нужном месте :) (c) **RPGman**

В общем ссылки-Lua нужны для того, чтобы с максимальной скоростью отправлять в Lua Lua-объекты. Если вы передаёте integer на Lua-сторону, то это занимает какое-то время. Если передаёте строку – то времени уйдёт ещё больше – потому что преобразования нативная сторона <--> Lua занимают ещё больше времени. Если вы передаёте в Lua структуру или экземпляр класса или массив – то это тоже занимает время – нужно создать userdata, проинициализировать его, навесить на него метатаблицу. Если достаёте этот объект (или любую другу переменную) из глобального пространства – то плюсуется время, которое тратится на поиск переменной в глобальном пространстве. И т.д. Я не знаю сколько времени вы можете выиграть за счёт Lua-ссылок, но раз такая возможность имеется и ей многие люди пользуются, то я счёл своим долгом реализовать Lua-ссылки в библиотеке CrystalLUA.

Lua-ссылка в CrystalLUA представляет собой класс TLuaReference, экземпляры которого создаются с помощью:

```
function TLua.CreateReference(const global_name: string=''): TLuaReference;
```

Если в качестве аргумента указать имя глобального объекта (переменной или процедуры), то в ссылку попадёт значение этой глобальной переменной или процедуры. Если аргумент не указан, то изначальное значение по ссылке равно **nil**. Когда ссылка потеряет свою актуальность – обязательно удаляйте её, чтобы не плодить утечек памяти. Значение по ссылке можно брать или изменять. Делается это с помощью свойств:

```
type
  TLuaReference = class
  . . .
  public
    . . .
    property Value: Variant read GetValue write SetValue;
    property ValueEx: TLuaArg read GetValueEx write SetValueEx;
  end;
```

Причём что интересно – в качестве ValueEx вы **можете задавать Lua-таблицу**! Но брать Lua-таблицу из Value и ValueEx вы не можете!

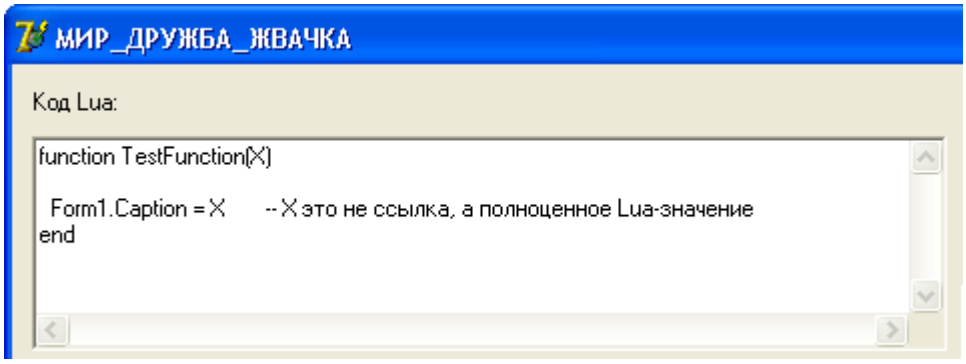
Если вы знаете, что по ссылке лежит Lua-таблица, тогда TLuaReference представляет другой интерфейс. Не забывайте, что Lua-таблица – единственный тип, которого нельзя повторить на нативной стороне и приходится использовать вот такие велосипеды, чтобы грамотно обращаться с Lua-стеком.

```
type
  TLuaReference = class
  . . .
  public
    function AsTableBegin(var Table: PLuaTable): boolean;
    function AsTableEnd(var Table: PLuaTable): boolean;
    . . .
  end;
```

Не забывайте вызывать AsTableEnd() после того как обработали таблицу.

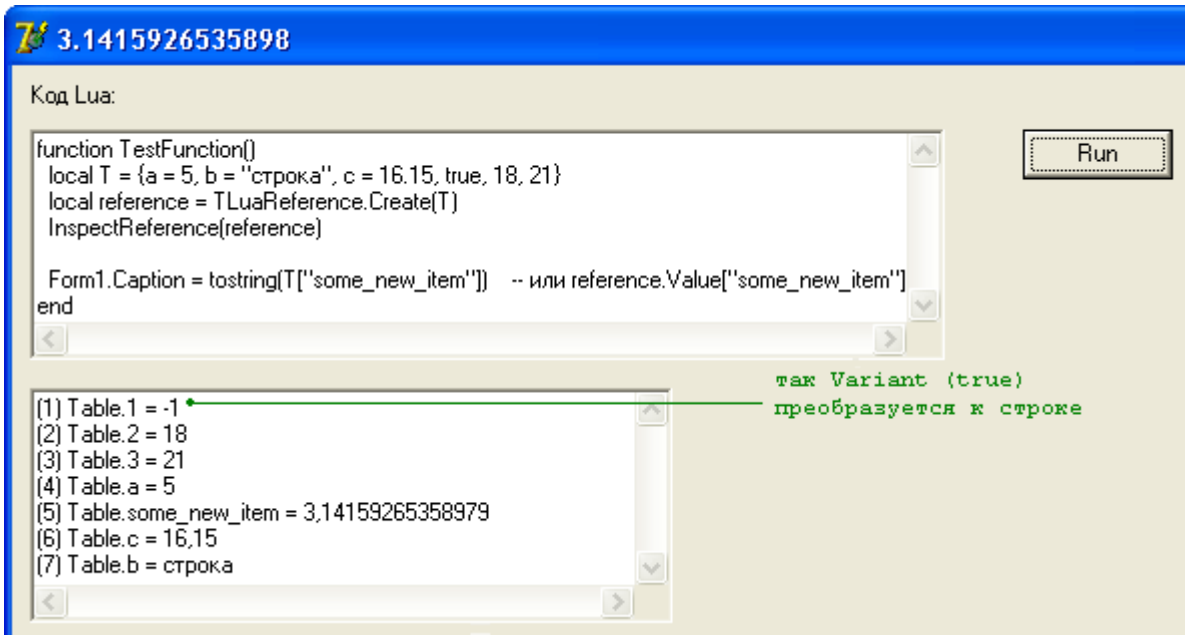
Для того чтобы передать значение ссылки в Lua, нужно воспользоваться экземпляром класса TLuaReference как обычным экземпляром класса. CrystalLUA распознаёт TLuaReference и отправляет в Lua не экземпляр класса, а само значение по ссылке.

```
var
  ConstStr: TLuaReference;
implementation
. . .
  ConstStr := Lua.CreateReference();
  ConstStr.Value := 'МИР_ДРУЖБА_ЖВАЧКА'; // инициализация значения ссылки

. . .
  // вызов функции, в качестве аргумента указать Lua-ссылку
  // в Lua приходит не TLuaReference, а значение ссылки !
  // можно указывать ссылку как параметр,
  // а можно и через TLuaArg: Arg.AsObject := ConstStr;
  Lua.Call('TestFunction', [ConstStr]);

. . .
  // после того как ссылка потеряла свою актуальность,
  // обязательно удалите её - чтобы не плодить утечек памяти
  ConstStr.Free;
```



Для того чтобы окончательно запутать пользователя ☺, я добавил возможность создавать TLuaReference на стороне Lua. На стороне Lua, TLuaReference – типичный класс, который может создаваться как “на стеке”, так и “в куче” и в качестве параметра принимает значение, которое стоит занести в ссылку. TLuaReference в Lua имеет свойство Value. В качестве примера использования ссылок, я как раз рассмотрю вариант таблицы.

```
procedure luaInspectReference(const Arg: TLuaArg);
var
  Ref: TLuaReference;
  Table: PLuaTable;
  Pair: TLuaPair;
begin
  // на нативную сторону, ссылка приходит как
  // экземпляр TLuaReference
  Ref := Arg.AsObject as TLuaReference;

  // взять таблицу из ссылки
  Ref.AsTableBegin(Table);

      // изменить таблицу
      Table.KeyValue['some_new_item'] := PI; // константа “Пи”

      // список того, что есть в таблице
      Form1.Memo2.Lines.Clear;
      if (Table.Pairs(Pair{, ключ})) then
      repeat
        Form1.Memo2.Lines.Add(Format('(%d) Table.%s = %s',
                              [Pair.Iteration, Pair.Key, Pair.Value]));
      until not Pair.Next();

  // окончить работу с таблицей
  Ref.AsTableEnd(Table);
end;
```



FAQ

Задавайте вопросы – список будет пополняться

Вопрос: Не компилится/не работает под Delphi2009/FPC. Что делать?
Ответ: На данный момент ведутся работы по развитию библиотеки. Если хочешь – присоединяйся !

Вопрос: Есть ли вариант CrystalLUA для других платформ? Linux там MacOS…
Ответ: Пока нет. Но всё реализуемо. Смотри предыдущий вопрос.

Вопрос: Я обнаружил <такую-то> ошибку.
Ответ: Это хорошо. По всем ошибкам обращайся в личку. Координаты на первой странице.

Вопрос: Что быстрее: массивы или Lua-таблицы?
Ответ: На нативной стороне быстрее нативные данные, на Lua-стороне быстрее Lua-данные. На стороне Lua – значительно быстрее будут таблицы.

Вопрос: Что лучше использовать: свойства или методы ?
Ответ: Разница не большая, но свойства быстрее.

Приложение. Интерфейс TLua

```
TLua = class(TObject)
public
    constructor Create;
    destructor Destroy; override;
    procedure GarbageCollection();
    function CreateReference(const global_name: string=''): TLuaReference;
    class function GetProcAddress(ProcName, throw_exception=false): pointer;

    // загрузка и запуск скриптов
    procedure RunScript(const Script: string);
    procedure LoadScript(const FileName: string); overload;
    procedure LoadScript(const ScriptBuffer: pointer;
    const ScriptBufferSize: integer; const ChunkName: string=''); overload;

    // проверка при калбеке
    procedure ScriptAssert(const FmtStr: string; const Args: array of const);
    function  CheckArgsCount(const ArgsCount: array of integer;
     const ProcName: string=''; const AClass: TClass=nil): integer; overload;
    function  CheckArgsCount(const ArgsCount: TIntegerDynArray;
     const ProcName: string=''; const AClass: TClass=nil): integer; overload;
    procedure CheckArgsCount(const ArgsCount: integer;
     const ProcName: string=''; const AClass: TClass=nil); overload;

    // вызовы
    function VariableExists(const Name: string): boolean;
    function ProcExists(const ProcName: string): boolean;
    function Call(const ProcName: string; const Args: TLuaArgs): TLuaArg;
    function Call(const ProcName: string; const Args:array of const):TLuaArg;

    // регистрация
    procedure RegClass(const AClass:TClass;const use_published:boolean=true);
    procedure RegClasses(const AClasses: array of TClass;use_published=true);
    function  RegRecord(Name: string;const tpinfo: ptypeinfo):PLuaRecordInfo;
    function  RegArray(Identifier, itemtypeinfo, Bounds): PLuaArrayInfo;
    function  RegSet(const tpinfo: ptypeinfo): PLuaSetInfo;
    procedure RegProc(ProcName: string; const Proc: TLuaProc; ArgsCount =-1);
    procedure RegProc(AClass; ProcName; Proc; ArgsCount; with_class=false);
    procedure RegProperty(AClass; PropertyName; tpinfo, PGet, PSet: pointer;
              parameters: PLuaRecordInfo; default: boolean=false);
    procedure RegVariable(const VariableName: string; const X;
              tpinfo: pointer; IsConst: boolean = false);
    procedure RegConst(const ConstName: string; const Value: Variant);
    procedure RegConst(const ConstName: string; const Value: TLuaArg);
    procedure RegEnum(const EnumTypeInfo: ptypeinfo);

    // вспомогательные свойства
    property ResultBuffer: TLuaResultBuffer read
    property Variable[const Name: string]: Variant read/write
    property VariableEx[const Name: string TLuaArg read/write
    property RecordInfo[const Name: string]: PLuaRecordInfo read
    property ArrayInfo[const Name: string]: PLuaArrayInfo read
    property SetInfo[const Name: string]: PLuaSetInfo read

    // основные свойства
    property Handle: pointer read
    property Preprocess: boolean read/write
    property Args: TLuaArgs read
    property ArgsCount: integer read
  end;
```