Название: Extern Logic Interpreter (Интерпретатор Внешней Логики).

Принцип работы: построчная интерпретация команд.

Форма: динамически связываемая библиотека (dll).

Назначение

Extern Logic Interpreter (далее ELI) был разработан для вынесения логических манипуляций за пределы кода исходного приложения. Его непосредственное назначение – максимально абстрагировать программу от логических операций, вынеся их реализацию во внешние файлы, которые можно легко модифицировать без перекомпиляции основного приложения. В идеале приложение, использующее ELI, может содержать только обособленные механизмы взаимодействия с окружением (например, функции для работы с файлами, вывод на экран и т.п.), а также специальные функции-обертки, предоставляющие ELI доступ к этим механизмам. Связыванием всего этого в определенный алгоритм займется ELI.

Механизм работы

ELI является построчным интерпретатором, использующим команды, схожие по синтаксису с языками программирования высокого уровня, такими как, например, C++. В большинстве случаев трансляция строк происходит последовательно, исключение составляют объявления пользовательских процедур, условий и циклов, тела которых парсятся отдельно, до запуска трансляции основного скрипта.

Единицей трансляции считается скрипт. Скрипт состоит из строк, разделенных символом «;» и может быть, как получен библиотекой ELI напрямую из основного приложения, так и загружен из внешнего файла. Единиц трансляции может быть несколько, их файлы включаются в трансляцию с помощью директивы #include или функции _Run().

Перед трансляцией интерпретатор выполняет подготовку скрипта: выделяет в тексте строковые константы и фрагменты кода, заключенные в фигурные скобки «{}». Фрагменты кода заменяются в тексте скрипта на специальные идентификаторы, а их тела заносятся в специальный стек. Такие фрагменты называются отложенными, их трансляция начинается только при непосредственном обращении.

Важно: в файле может быть только один скрипт.

Важно: файл скрипта не должен содержать пустых строк в конце.

Текст скрипта может содержать комментарии. Закомментированная строка должна начинаться двух символов прямого слеша «//» и заканчиваться точкой с запятой «;».

ELI предоставляет следующий функционал:

Директивы: прямые команды интерпретатору.

Переменные: внутренние переменные интерпретатора, существуют на протяжении работы скрипта. Область видимости: текущий контекст.

Простые и сложные математические операции с числами.

Циклы.

Условия.

Объекты: абстракции, представляющие собой набор типизированных данных и позволяющие работать с ним, как с привычными объектами классов, используя методы и свойства. Область видимости: глобальная.

Функции: указатели на функции-обертки в основном приложении. Позволяют использовать алгоритмы, содержащиеся в основном приложении. Могут передавать данные из основного приложения в скрипт с помощью возвращаемых значений. Область видимости: глобальная.

Процедуры: фрагменты кода, которые вызываются (и транслируются) при непосредственном обращении. В отличие от функций, не возвращают значений. Область видимости: глобальная.

Стек параметров: является прослойкой между основным приложением и ELI, служит для обмена данными. Представляет собой динамически расширяемое множество типизированных структур, представляющих собой понятие «параметр». Доступ к стеку двусторонний. Область видимости: глобальная.

Вывод стека сообщений: ошибки и служебные сообщения, возникающие в процессе выполнения скрипта, заносятся в специальную строковую переменную, которая может быть передана в основное приложение с помощью соответствующей функции.

Вывод стека переменных: содержимое стека переменных в форматированном виде выводится в строковую переменную.

Вывод стека функций: вывод в строковую переменную содержимого стека функций-оберток. Выводятся как определенные в библиотеке интерпретатора функции, так и объявленные в основном приложении.

Вывод стека параметров.

Вывод стека объектов.

Дебаггер скриптов: включаемая опция, задается при запуске каждого скрипта. Включает логирование транслируемых строк в файл, туда же выводит сообщения об ошибках. Логирование замедляет выполнение скрипта.

Дебаггер библиотеки: опция, включаемая с помощью специальной функции интерпретатора или метода ELI::SetDebug(). Включает логирование действий самого ELI, перенаправляет вывод либо в файл, либо в STDOUT. Логирование в файл сильно замедляет выполнение скриптов.

Реализация

ELI написан на C++, использует контейнеры std::vector<std::wstring> для хранения строк скрипта, а также специальные классы для представления стеков переменных, параметров, функций, объектов и отдельных фрагментов кода, экземпляры которых содержатся в классе **ELI**. Кодировка строк – UTF-8. Внутренний язык ELI по сути является оберткой для C++ кода. Каждая строка, с помощью специального алгоритма, интерпретируется в набор команд C++, после чего выполняется. Приложение, используя функцию класса ELI, передает интерпретатору текст скрипта (или имя файла, который нужно транслировать) и необязательный список входных параметров, ELI выполняет трансляцию строк, после чего возвращает результат выполнения (тип const wchar_t*), если он предполагается. В случае ошибки возвращает «-err-».

Интеграция в приложение

Основной код ELI выполнен в виде динамически связываемой библиотеки, которую необходимо подключить к основному приложению. Чтобы работать с функционалом ELI приложению понадобится интегрировать объект класса ELI из кода библиотеки. Для этой цели используется абстрактный интерфейс, объявленный в заголовочном файле **eli_interface.h**, который распространяется вместе с библиотекой.

Файл содержит интерфейс **ELI_INTERFACE**, который является родительским для класса **ELI**, объявленного в коде библиотеки. Также в конце файла объявлены две экспортные функции-фабрики, с помощью которых происходит инициализация и уничтожение интерфейса.

```
//---------------------------------------------------------------------------------------------------------------
__declspec(dllexport) int __stdcall GetELIInterface(ELI__INTERFACE **eInterface);
typedef int (__stdcall *GETELIINTERFACE)(ELI__INTERFACE **eInterface);

__declspec(dllexport) int __stdcall FreeELIInterface(ELI__INTERFACE **eInterface);
typedef int (__stdcall *FREEELIINTERFACE)(ELI_INTERFACE **eInterface);
//---------------------------------------------------------------------------------------------------------------
```

Функции возвращают 1 в случае успешного выполнения или 0 в случае ошибки.

После подключения файла eli_interface.h к проекту основного приложения, необходимо определить указатели на функции-фабрики и инициализировать их.

```
//---------------------------------------------------------------------------------------------------------------
GETELIINTERFACE GetELI;
FREEELIINTERFACE FreeELI;

ELI_INTERFACE *eIface;

GetELI = (GETELIINTERFACE) GetProcAddress(dllhandle, "GetELIInterface");
FreeELI = (FREEELIINTERFACE) GetProcAddress(dllhandle, "FreeELIInterface");
//---------------------------------------------------------------------------------------------------------------
```

Инициализировать указатель на объект-интерфейс с помощью определенной ранее функции.

```
//---------------------------------------------------------------------------------------------------------------
GetELI(&eIface);
//---------------------------------------------------------------------------------------------------------------
```

Теперь достаточно использовать указатель **eIface**, чтобы получить доступ к методам класса **ELI**, описанного внутри библиотеки. Таким образом в приложении можно использовать несколько экземпляров ELI, которые будут действовать независимо друг от друга.

Список методов интерфейса ELI_INTERFACE

const wchar_t * __stdcall GetVersion() – возвращает текущую версию ELI.

const wchar_t * __stdcall ShowVarStack() – возвращает содержимое стека переменных в текстовом виде.

const wchar_t * __stdcall ShowObjStack() – возвращает содержимое стека объектов в текстовом виде.

const wchar_t * __stdcall ShowClassStack() – возвращает содержимое стека классов в текстовом виде.

const wchar_t * __stdcall ShowProcStack() – возвращает содержимое стека процедур в текстовом виде.

const wchar_t * __stdcall ShowFuncStack() – возвращает содержимое стека функций в текстовом виде.

const wchar_t * __stdcall ShowParamStack() – возвращает содержимое стека параметров в текстовом виде.

const wchar_t * __stdcall ShowFragmentStack() – возвращает содержимое стека отложенных фрагментов кода.

const wchar_t * __stdcall ShowInfoMessages() – возвращает список сообщений интерпретатора.

const wchar_t * __stdcall RunScript(const wchar_t *imptext, const wchar_t *parameter, bool log) – используется для выполнения скрипта. Аргумент log служит флагом использования дебаггера. Аргумент parameter может содержать несколько значений разделенных символом «|».

const wchar_t * __stdcall RunScriptFromFile(const wchar_t *filepath, const wchar_t *parameter, bool log) – используется для выполнения скрипта из файла. Аргумент log служит флагом использования дебаггера. Аргумент parameter может содержать несколько значений разделенных символом «|». Путь к файлу может быть как абсолютный так и условный («.\<имя файла>»). В последнем случае поиск ведется в каталоге где расположена dll интерпретатора.

void __stdcall SetDebug(bool enable_dbg, bool in_file) – используется для включения/отключения записи диагностической информации о работе самого ELI и указания куда именно направить вывод.

bool __stdcall DebugEnabled() – проверяет, используется ли режим дебага.

void __stdcall AddFunction(const wchar_t *name, const wchar_t *params, func_ptr fptr) – добавляет функцию в стек.

void __stdcall DeleteFunction(const wchar_t *name) – удаляет функцию из стека.

void __stdcall CallFunction(const wchar_t *name) – вызывает функцию.

wchar_t * __stdcall GetFunctionResult(const wchar_t *name) – преобразует возвращаемое значение функции в строку и возвращает указатель на нее. В случае ошибки возвращает NULL.

void __stdcall SetFunctionResult(const wchar_t *name, const char* result) – устанавливает возвращаемое значение функции.

void __stdcall SetParam(const wchar_t *name, const wchar_t *new_val) – устанавливает новое значение параметра или добавляет новый параметр.

int __stdcall GetParamToInt(const wchar_t *name) – преобразует параметр в integer и возвращает его.

float __stdcall GetParamToFloat(const wchar_t *name) – преобразует параметр в float и возвращает его.

const wchar_t * __stdcall GetParamToStr(const wchar_t *name) – преобразует параметр в строку и возвращает указатель на нее. В случае ошибки возвращает NULL.

const wchar_t * __stdcall GetCurrentFuncName() – возвращает имя текущей исполняемой функции ELI. Может использоваться для того, чтобы сообщить функции-обертке из библиотеки, подключаемой во время выполнения скрипта, сопоставленное ей имя функции ELI.

void __stdcall AddToLog(const wchar_t *msg) – добавляет запись в список сообщений интерпретатора.

Объявление функции-обертки

Чтобы интерпретатор мог использовать код основного приложения, этот код нужно обернуть в специальные функции. Функции-обертки не должны быть членами класса или объявляться в классе со спецификатором **static**.

Важно: возвращаемый тип и аргументы функции-обертки строго регламентированы, так как вызов функций происходит по указателю, тип которого определен в файле eli_interface.h:

```
//---------------------------------------------------------------------------------------------------------------
typedef void (__stdcall *func_ptr)(void*);
//---------------------------------------------------------------------------------------------------------------
```

Аргумент void* используется, чтобы передать приложению указатель на текущий экземпляр объекта ELI. Это позволяет разным экземплярам интерпретатора запускать функции из основного приложения независимо друг от друга.

```
//---------------------------------------------------------------------------------------------------------------------
void __stdcall foo(void *p)
{
  //приводим указатель p к типу ELI_INTERFACE
  ELI_INTERFACE *ELI = (ELI_INTERFACE*)p;
  int x = ELI->GetParamToInt("pInd"); //получение параметра из стека dll

  /*некоторый код основного приложения*/

  ELI->SeFunctionResult("_foo", "0"); //установка нужного возвращаемого значения
}
//---------------------------------------------------------------------------------------------------------------------
```

Здесь из стека ELI читается параметр с именем pInd и преобразуется к типу integer. После чего выполняется нужный код на стороне основного приложения и с помощью интерфейса в dll передается нужное значение, которое ELI передаст скрипту.

Важно: Использование метода ELI_INTERFACE::SetFunctionResult() в конце тела функции является обязательным. Каждая функция должна возвращать значение, интерпретатор будет ожидать этого. Возвращаемое значение всегда строкового типа, интерпретатор сам преобразует его в нужный тип, если это будет возможно и/или необходимо.

После объявления нужно добавить функцию в стек.

```
//---------------------------------------------------------------------------------------------------------------------
ELI->AddFunction("_foo", "num pInd", &foo);
//---------------------------------------------------------------------------------------------------------------------
```

«_foo» это имя функции, которое будет использовать ELI. Символ «_» является обязательным, с его помощью интерпретатор понимает, что далее в тексте скрипта идет имя функции. Имя функции нечувствительно к регистру.

Строка «num pInd» описывает список аргументов функции и их тип. Если аргументов несколько, их следует указывать через запятую, пробелы ставятся только между типом и именем аргумента. Один аргумент не выделяется запятой. Имена аргументов не чувствительны к регистру и не являются строго регламентированными.

Теперь, когда интерпретатор обнаружит в строке скрипта выражение типа «_foo(12)», он обратится к стеку функций, найдет там функцию _foo(), проверит количество и тип передаваемых ей аргументов и вызовет функцию-обертку foo() из основного приложения, передав ей в качестве аргумента указатель на свой экземпляр. После чего получит результат и вернет его в обрабатываемую строку скрипта.

Кроме того, можно передавать скрипту различные данные в любом месте кода основного приложения, используя стек параметров.

```
//-------------------------------------------------------------------------------------------------------------------
ELI->SetParam(«NewParamName», <значение>);
//-------------------------------------------------------------------------------------------------------------------
```

Метод ELI_INTERFACE::SetParam() добавляет новый элемент в стек параметров только если имя нового параметра не присутствует в стеке. В случае совпадения имен, метод просто обновит значение параметра, присутствующего в стеке. Это сделано для экономии ресурсов. В теле скрипта получить параметр из стека можно с помощью соответствующей встроенной функции. Входящие параметры для скрипта также заносятся в стек параметров с предопределенными именами типа INPRM<порядковый номер>.

Синтаксис внутреннего языка

Типы данных

Внутренний язык ELI оперирует двумя типами данных.
num – числовой тип.
sym – символьный тип.

Числовой тип используется для работы как с целыми числами, так и с вещественными. По умолчанию все значения хранятся как вещественные. Разделителем целой и дробной части числа является точка «.». Точность числа: 3 знака. Приведение к целочисленному отображению осуществляется с помощью встроенных функций round() и _int(), либо автоматически, если того требует алгоритм.

Внутренний язык ELI является языком с нестрогим соответствием типов. Если в выражении строго не указано, что должен использоваться числовой тип, значение будет рассматриваться интерпретатором как текст. Так, например, строковой переменной можно присвоить значение 21, при этом оно будет преобразовано в текст. Но обратная операция, например, присвоение числовой переменной значения '22', вызовет ошибку и остановку выполнения скрипта.

Переменные

Объявление переменных производится с помощью записи типа:

```
//-------------------------------------------------------------------------------------------------------------------
$n = num;
$s = sym;
//-------------------------------------------------------------------------------------------------------------------
```

Здесь $n – имя переменной, num – тип, а после идет значение, которым инициализируется переменная. Отсутствие значения приводит к инициализации значением по умолчанию. В данном случае, после выполнения строки, в стек добавится переменная числовая $n, равная 0.000 и строковая $s, имеющая значение «». Переменная может быть

инициализирована константным значением, скалярным выражением, другой переменной, свойством объекта или возвращаемым значением функции либо метода объекта. Строковая переменная может быть инициализирована значениями числовых переменных, при этом их значения будут преобразованы в текст.

```
//-------------------------------------------------------------------------------------------------------------
$n = num;
$s = sym 10$n;
//-------------------------------------------------------------------------------------------------------------
```

Здесь переменная $s будет инициализирована строковым значением «100.000», которое представляет собой результат конкатенации строковой константы «10» и «0.000» – приведенного к строковому типу значения переменной $n. Кроме того, конструкция $str = sym <arg1> + <arg2> воспринимает выражение в правой части как строку ('2+2'). Но результатом конструкции $str = 2 + 2 будет 22, то есть конкатенация.

Символ «$» является обязательной частью имени, это флаг, сообщающий ELI, что дальше идет имя переменной. В одной строке может быть объявлена только одна переменная.

Объявление типа можно опустить и использовать конструкцию типа $var = <значение>, тогда переменной автоматически будет присвоен тип на основании вычисления правой части выражения.

Присваивание значений переменным производится с помощью следующей конструкции:

```
//-------------------------------------------------------------------------------------------------------------
$n = 12;
$s = some text;
//-------------------------------------------------------------------------------------------------------------
```

В правой части выражения могут находится: константа, допустимый для данного типа набор операций (сложение, вычитание, конкатенация и т.д.), свойство или метод объекта и результат функции. Причем в данном случае переменной $s будет присвоено значение «sometext», интерпретатор опустит пробел. Однако, если заключить значение в одинарные кавычки, интерпретатор воспримет этот фрагмент кода, как константную строку и присвоит переменной значение «some text».

Константные строки выбираются интерпретатором на стадии анализа текста скрипта и заносятся в стек переменных под специальным именем. В дальнейшем, при выполнении строки кода, содержащего такую переменную, ELI подставляет вместо нее значение из стека. Константные строки хранятся в стеке в единственном экземпляре. Если в коде скрипта несколько раз встречается одна и та же константная строка, интерпретатор во всех случаях подставит значение одного и того же элемента стека переменных.

Важно: следует быть осторожным при использовании в строковых константах символа «$», парсер ELI воспримет его как символ-маркер переменной.

Строковые переменные помимо операции конкатенации (обозначается знаком «+»), поддерживают возможность прямого объединения значений. Вообще, при работе со

строковым типом, все содержимое правой части выражения по умолчанию рассматривается интерпретатором как текст. Любые символы (кроме специальных), строковые константы, заключенные в одинарные кавычки, значения функций, методов, свойств и переменных будут приведены к типу sym и помещены в одну строку. Включение в строку происходит последовательно, причем конкатенация («+») не требуется, достаточно расположить значения в нужном порядке. Пробелы (если они находятся не в строковых константах) игнорируются. Любые знаки математических операций, кроме «+», считаются обычными символами.

Операции

ELI поддерживает все основные математические операции для числового типа («+», «-», «*», «/»), включая оператор приоритета «()». Кроме того для чисел доступны операции сравнения, такие как:

= присваивание;

== равно;

> больше;

< меньше;

>= больше или равно;

<= меньше или равно;

!= не равно;

Также числовые переменные поддерживают операции инкремента («++») и декремента («--»).

К строковому типу применима только конкатенация «+» и операция присваивания «=».

Условия

```
//-------------------------------------------------------------------------------------------------------------
if (<условие>)
  {
    <список действий>
  }
else if (<условие>)
  {
    <список действий>
  }
else
  {
    <список действий>
  }
//-------------------------------------------------------------------------------------------------------------
```

ELI поддерживает ветвленные условия (if - else if - else), а также вложенность условий. Синтаксис конструкции следующий:

```
//-------------------------------------------------------------------------------------------------------------
$i = 5.15;

if ($i > 5)
  {
    $i = 0;
  }
else if ($i == 5)
  {
    $i = 1;
  }
else
  {
    $i = -1;
  }
//-------------------------------------------------------------------------------------------------------------
```

В качестве параметров проверяемого выражения могут выступать: число, числовая переменная либо свойство объекта, а также функции и методы объектов, возвращающие числовой результат. При вычислении, дробная часть значений также учитывается, поэтому в приведенном выше примере $i будет больше 5. В качестве операции может использоваться любая операция сравнения. В случае отсутствия в выражении символа операции сравнения, выражение проверяется на истинность. Выражение является истинным, если результат его вычисления больше 0.

Конструкция select

```
//-------------------------------------------------------------------------------------------------------------
select (<параметр>)
{
  when <значение1> then {<список действий>}
  when <значение2> then {<список действий>}
  ...
}
//-------------------------------------------------------------------------------------------------------------
```

Альтернативой ветвленным условиям if-else служит конструкция select. Ее суть сводится к проверке переданного входного параметра на соответствие со значениями, которые содержатся в исполнительном блоке. Проверка осуществляется с помощью специального служебного слова **when**, а описание действий, которые нужно совершить если выполняется равенство параметра и проверочного значения, идет после служебного слова **then**.

В качестве параметра конструкции select может применяться: переменная, результат функции, свойство или метод объекта. Скалярные выражения не допускаются. Значения, по которым производится проверка, могут быть только константами. Синтаксис допускает смешивание строковых и числовых значений в пределах конструкции, это значит, что, например, результат выполнения функции может быть проверен как на числовое значение 2, так и на строковое значение «два». Конечно, числовую переменную нельзя сравнить со

строковым значением, но в этом случае интерпретатор просто отбросит это условие, как ошибочное и продолжит выполнение исполнительного блока.

В исполнительном блоке конструкции select, помимо блоков-проверок when...then могут содержаться любые другие действия, однако они будут выполнены в любом случае. Эту особенность можно использовать, например, для возврата результата по умолчанию.

```
//-------------------------------------------------------------------------------------------------------------------
$res = num _foo(); //$res получает некоторое значение

select ($res)
{
  when 10 then {_return(1);}
  when 21 then {_return(2);}

 _return(0)
}
//-------------------------------------------------------------------------------------------------------------------
```

Область видимости исполнительного блока конструкции – глобальная. Это значит, что все операции, произведенные с переменной $res из предыдущего примера при истинности первого проверочного блока, отразятся на результате выполнения последующих проверочных блоков.

Циклы

Интерпретатор предоставляет три типа циклов: **for**, **while** и **count (**являющийся упрощенной реализацией for). Циклы могут быть вложенными. В качестве параметров условий циклов (кроме count) могут применяться числа, числовые переменные либо свойства объектов, а также функции и методы объектов, возвращающие результат, который может быть интерпретирован как число. Переменные, объявленные в цикле, будут инициализироваться на каждой итерации.

Цикл for

```
//-------------------------------------------------------------------------------------------------------------------
for (<начальное значение>, <условие срабатывание>, <шаг>)
{<некоторые действия>}
//-------------------------------------------------------------------------------------------------------------------
```

Выполняет код, заключенный в фигурные скобки, пока операция сравнения между первым и вторым параметрами условия является истинной. Третий параметр представляет собой шаг, на который изменяется значение первого параметра условия. Шаг может быть целым и дробным числом, а знаки «+» и «-» определяют способ изменения первого параметра. Второй параметр состоит из знака операции сравнения и числового значения, обозначающего верхнюю границу условия. Допустимые операции сравнения: «>», «<», «>=» и «<=». Первый параметр может задаваться числовой константой, результатом выполнения функции или метода и числовым свойством объекта либо переменной, которая должна быть

инициализирована некоторым значением до выполнения цикла. Если в качестве первого параметра используется переменная, ее значение изменяется после каждой итерации.

```
//-------------------------------------------------------------------------------------------------------------------
for ($var, <= 10, +1)
{<некоторые действия>}

for (0, <= 10, -1)
{<некоторые действия>}
//-------------------------------------------------------------------------------------------------------------------
```

Цикл while

```
//-------------------------------------------------------------------------------------------------------------------
while(<значение1> != <значение2>)
{<некоторые действия>}
//-------------------------------------------------------------------------------------------------------------------
```

Выполняет код, заключенный в фигурные скобки, пока операция сравнения между первым и вторым параметрами условия является истинной. В отличие от предыдущего цикла, while не содержит шага, изменяющего параметры условия. Условие проверяется на ложность перед каждой итерацией. Если условие ложно еще до первой итерации – тело цикла не будет выполнено. Оба параметра условия могут представлять собой числовые константы, числовые переменные, результаты выполнения функций или методов и числовые свойства объектов. Допустимые операции сравнения: «>», «<», «>=», «<=», «!=» и «==».

Цикл count

```
//-------------------------------------------------------------------------------------------------------------------
count(<счетчик>)
{<некоторые действия>}
//-------------------------------------------------------------------------------------------------------------------
```

Представляет собой упрощенную версию цикла for. Единственный параметр задает количество итераций выполнения тела цикла. Параметр может быть числовой константой, результатом выполнения функции или метода и числовым свойством объекта либо переменной, которая должна быть инициализирована некоторым значением до выполнения цикла. Параметр обязательно должен быть положительным значением.

Директивы

Директивы это служебные команды интерпретатора, обозначающие то или иное действие, обязательное к исполнению. Директивы бывают следующие:

#begin – обозначает начало скрипта. Обязательная.

#end – обозначает конец скрипта. Обязательная.

#include <имя файла> – включает код, содержащийся в отдельном внешнем файле, в скрипт. Все переменные, объявленные во включаемом файле, инициализируются в текущем активном стеке.

#exit – принудительное завершение скрипта.

#procedure <имя процедуры>(<аргументы>){**<тело процедуры>**} – объявляет пользовательскую процедуру.

#drop procedure <имя процедуры> – удаляет пользовательскую процедуру.

#make <имя переменной> {<некоторый код>} – присваивает переменной идентификатор отложенного фрагмента кода. В дальнейшем этот идентификатор можно передать другой переменной или присвоить свойству объекта.

#run <имя переменной> – выполняет фрагмент кода, идентификатор которого содержится в переменной. Все переменные, созданные во время выполнения этого фрагмента, будут помещены в стек, который являлся активным для того контекста, где была вызвана директива #run. После трансляции, фрагмент удаляется из стека фрагментов.

#class <имя класса> {тело класса} – создает новый класс объектов.

#modify class <имя класса> {тело класса} – изменяет существующий класс объектов.

#property <имя свойства>=<значение> – добавляет новое частное свойство в выбранный класс. Применима лишь внутри кода тела класса.

#public property <имя свойства>=<значение> – добавляет новое публичное свойство в выбранный класс. Применима лишь внутри кода тела класса.

#method <имя метода>(<аргументы>){**<тело метода>**} – добавляет новый частный метод в выбранный класс. Применима лишь внутри кода тела класса.

#public method <имя метода>(<аргументы>){**<тело метода>**} – добавляет новый публичный метод в выбранный класс. Применима лишь внутри кода тела класса.

#drop property <имя свойства> – удаляет свойство из класса.

#drop method <имя метода> – удаляет метод из класса.

#drop class <имя класса> – удаляет класс объектов.

#return <значение> – осуществляет возврат значения из метода класса.

#protect {<некоторый код>} – выполняет защищенный участок кода. Код, содержащийся в этом фрагменте, будет выполнен в любом случае и не прервет трансляцию даже в случае возникновения исключения. Все ошибки тем не менее будут внесены в лог трансляции.

#trigger <условие> {<некоторый код>} – объявляет триггер. Триггер это отложенный фрагмент кода, который будет выполнен, если условие, заявленное в триггере, будет истинным. Триггер объявляется один раз и проверяется на срабатывание после трансляции каждой строки скрипта. Выполнение триггера никак не влияет на трансляцию основного скрипта, даже если при трансляции тела триггера были получены ошибки или выброшено исключение.

#drop trigger <условие> – удаляет триггер.

#set {<директива>} – настраивает интерпретатор ELI. В теле директивы используются несколько служебных директив.

#cnum | #!cnum – включает/выключает парсинг числовых констант. Доступно только в теле директивы #set. По умолчанию включено.

#csym | #!csym – включает/выключает парсинг символьных констант. Доступно только в теле директивы #set. По умолчанию включено.

#keepobjects | #!keepobjects – указывает сохранять/не сохранять содержимое стека объектов после трансляции скрипта. Доступно только в теле директивы #set. По умолчанию включено.

#keepclasses | #!keepclasses – указывает сохранять/не сохранять содержимое стека классов после трансляции скрипта.. Доступно только в теле директивы #set. По умолчанию включено.

Функции

ELI может использовать любые функции, определенные в пользовательском приложении, при условии, что они будут объявлены в стеке функций. Кроме этого в состав ELI входят несколько встроенных функций:

_random(num pArea) – возвращает псевдослучайное значение из диапазона pArea.

_round(num pNumber, num pPrecision) – возвращает результат округления. Аргумент pPrecision указывает на точность округления (от 0 до 2 знаков после запятой).

_int(num pNumber) – возвращает значение аргумента pNumber, приведенное к целому числу. Дробная часть игнорируется.

_strlen(sym pStr) – возвращает длину строки.

_streq(sym pStr1, sym pStr2) – сравнивает строки, возвращает 1 если строки равны, 0 если нет. Учитывает регистр.

_istreq(sym pStr1, sym pStr2) – сравнивает строки, возвращает 1 если строки равны, 0 если нет. Не учитывает регистр.

_substr(sym pTargetStr, num pPos, num pCount) – возвращает подстроку.

_return(sym pReturnVal) – осуществляет возврат значения из скрипта.

_throw(sym pException) – завершение трансляции и добавление в лог пользовательского исключения с произвольным текстом.

_free(sym pVarName) – удаляет из стека переменную, возвращает 1 если успешно, в противном случае возвращает 0.

_LoadObjStack(sym pFilePath, num pClear) – загружает стек объектов из файла, возвращает 1 если успешно, 0 – в противном случае. Если pClear > 0 стек объектов очищается перед загрузкой из файла. Путь к файлу может быть как абсолютный так и условный («.\<имя файла>»). В последнем случае поиск ведется в каталоге где расположена dll интерпретатора.

_SaveObjStack(sym pFilePath) – сохраняет стек объектов в файл, возвращает 1 если успешно, 0 – в противном случае. Путь к файлу может быть как абсолютный так и условный («.\<имя файла>»). В последнем случае поиск ведется в каталоге где расположена dll интерпретатора.

_SaveObjects(sym pFilePath, sym pCathegory) – сохраняет в файл все объекты из стека с указанной категорией. Путь к файлу может быть как абсолютный так и условный («.\ <имя файла>»). В последнем случае поиск ведется в каталоге где расположена dll интерпретатора.

_CompactObjStack() – сжимает стек, удаляя из него записи, помеченные как удаленные, т.е. для которых поле Keep == 0.

_RemoveObjects(sym pCathegory) – удаляет из стека все объекты с указанной категорией.

_ClearObjStack() – очищает стек объектов.

_Run(sym pVarName) – выполняет код, содержащийся в переменной с именем pVarName. Соответствует вызову связки директив **#make** и **#run**. Применяется в том случае, если исполняемый код генерируется прямо во время трансляции скрипта, например, если другая функция (например _LoadFileToVar() или ReadIn()) возвращает текст, который должен быть транслирован. Если нужный код указывается прямо в теле скрипта, необходимо использовать директивы #make и #run. Код исполняется в пределах текущего контекста.

_GetParamAsNum(sym pParam) – возвращает значение параметра pParam из стека параметров преобразованное в число.

_GetParamAsStr(sym pParam) – возвращает значение параметра pParam из стека параметров преобразованное в строку.

_SetParam(sym pParam, sym pVal) – добавляет в стек параметр с именем pParam и значением pVal. Если параметр с таким именем уже присутствует в стеке, его значение будет изменено.

_LoadFileToVar(sym pFile, sym pTarget) – загружает содержимое текстового файла в переменную с именем pTarget. Путь к файлу может быть как абсолютный так и условный («.\<имя файла>»). В последнем случае поиск ведется в каталоге где расположена dll интерпретатора.

_SaveVarToFile(sym pTarget, sym pFile) – сохраняет значение переменной в указанный файл. Путь к файлу может быть как абсолютный так и условный («.\<имя файла>»). В последнем случае поиск ведется в каталоге где расположена dll интерпретатора. Если путь к файлу не существует, будет создан новый файл.

_SaveFragmentToFile(sym pTarget, sym pFile) – сохраняет отложенный фрагмент с меткой, содержащейся в переменной pTarget в указанный файл. Путь к файлу может быть как абсолютный так и условный («.\<имя файла>»). В последнем случае поиск ведется в каталоге где расположена dll интерпретатора. Если путь к файлу не существует, будет создан новый файл.

Важно: функция _SaveFragmentToFile() сохраняет фрагмент «как есть», не разворачивая фрагменты тех меток, которые встретятся в тексте.

_GetConfig(sym pFile, sym pLine) – загружает значение параметра pLine из файла pFile. Путь к файлу может быть как абсолютный так и условный («.\<имя файла>»). В последнем случае поиск ведется в каталоге где расположена dll интерпретатора. Файл должен иметь структуру <параметр>=<значение>.

_SaveState() – сохраняет текущее состояние всех стеков ELI в файл state.log в рабочем каталоге ELI.

_SaveVarStack(num pLevel) – сохраняет состояние стека переменных в файл varstack.log в рабочем каталоге ELI. Если аргумент pLevel равен 0 сохраняется содержимое глобального стека, если 1 – локального (то есть используемого в данный момент).

_WriteOut(sym pStr) – выводит в стандартный поток вывода содержимое аргумента pStr. Кроме стандартного набора допустимых значений для аргумента, в нем могут быть переданы шесть зарезервированных слов:

#varstack – выводит в поток содержимое стека переменных. Используется текущий активный стек.

#funcstack – выводит в поток содержимое стека функций.

#prmstack – вывод стека параметров.

#objstack – вывод стека объектов.
#clstack – вывод стека классов.
#procstack – вывод стека процедур.
#frgstack – вывод стека отложенных фрагментов кода.
#**endl** – вывод символа конца строки (перенос каретки).

_ReadIn(sym pVar) – читает символы из стандартного потока ввода и записывает их в переменную pVar. Максимальное количество символов для ввода: 4096.

_System(sym pCmd) – выполняет команду Windows, содержащуюся в аргументе pCmd. Аналог C++ функции system().

_LastError() – возвращает описание последней зафиксированной ошибки.

_ConnectLib(sym pPath) – подключает к интерпретатору внешнюю библиотеку dll. Возвращает дескриптор подключенной библиотеки. В случае неудачи возвращает -1. Путь к файлу может быть как абсолютный так и условный («.\<имя файла>»). В последнем случае поиск ведется в каталоге где расположена dll интерпретатора.

_FreeLib(num pHandle) – освобождает подключенную внешнюю библиотеку.

ImportFunc(num pHandle, sym pExtName, sym pInName,sym pArgList) – импортирует из подключаемой библиотеки, имеющей дескриптор pHandle, функцию-обертку с именем pExtName и добавляет ее в стек интерпретатора под именем pInName и списком аргументов pArgList. Значениями аргументов pInName и pArgList должны быть константные строки, поскольку в pInName используется символ признака функции «», а pArgList содержит пробелы, которые будут потеряны без использования константной строки.

_DebugIntoFile() – включает логирование действий интерпретатора, делает вывод в файл в рабочем каталоге ELI.

_DebugIntoScreen() – включает логирование действий интерпретатора, делает вывод в стандартный поток вывода.

_StopDebug() – отключает логирование действий интерпретатора.

_sleep(num pMsec) – приостанавливает выполнение трансляции на pMsec миллисекунд. Аналог функции Sleep() в C++.

Аргументы функций представлены типами данных языка ELI:
num – числовой.
sym – строковый.

Тип параметра определяет лишь механизм интерпретации значения, которое будет передано функции в теле скрипта, сами параметры хранятся в стеке без указания типа. В самих же функциях-обертках используются методы класса ELI, типа GetParamToInt() для получения из стека параметра с нужным типом.

```
//-------------------------------------------------------------------------------------------------------------
$n = _random(10);
//-------------------------------------------------------------------------------------------------------------
```

Аргументами функции могут выступать: переменные, константы, скалярные выражения, свойства и методы объектов, а также другие функции.

Процедуры

Процедуры не возвращают значений, но их реализация полностью зависит от пользователя. Кроме того, в отличие от функций, предопределенных кодом библиотеки ELI или основного приложения, процедуры могут быть созданы, использованы и удалены во время выполнения скрипта. Определенная в одном контексте (например, в теле другой процедуры), процедура может быть использована в любом другом, до тех пор, пока не будет освобождена библиотека ELI. Имя процедуры – уникально, и не должно совпадать с именем встроенной функции. Все переменные, объявленные в теле процедуры, видимы только в ней и будут уничтожены, как только процедура завершит работу. В теле процедуры нельзя использовать переменные, объявленные вне ее, так как они размещены в другом стеке переменных.

Объявление процедуры:

```
//-------------------------------------------------------------------------------------------------------------
#procedure foo($arg1, $arg2)
{
<некоторый код>
}
//-------------------------------------------------------------------------------------------------------------
```

Обращение:

```
//-------------------------------------------------------------------------------------------------------------
:foo(10, 5);
//-------------------------------------------------------------------------------------------------------------
```

Удаление из списка:

```
//-------------------------------------------------------------------------------------------------------------
#drop foo;
//-------------------------------------------------------------------------------------------------------------
```

У процедуры может быть произвольное число аргументов. Аргументы инициализируются в локальном стеке переменных процедуры как переменные типа sym. При вызове процедуры, значения, переданные ей в качестве аргументов, будут присвоены соответствующим переменным. Передача аргументов происходит по значению, поэтому значения аргументов, переданных процедуре, остаются неизменными. В качестве аргументов

могут выступать: константные значения, переменные, результаты выражений, свойства объектов и возвращаемые значения функций и методов.

Пользовательские процедуры хранятся в специальном стеке. При объявлении процедуры в нем создаются два элемента с категорией «procedure» и именем, указанным в строке с объявлением. Один содержит список аргументов, второй – ссылку-идентификатор, указывающую на тело процедуры, хранящееся в стеке фрагментов кода.

Объекты

В языке ELI понятие «объект» представляет собой абстракцию, олицетворяющую некоторое множество записей из специального стека объектов. Для интерпретатора каждый элемент стека – это набор типизированных полей, описанных в структуре:

```
//-------------------------------------------------------------------------------------------------------------------
struct RESOURCE
{
  UINT Index;                    //индекс
  std::wstring ObjectCathegory;  //категория объекта-владельца
  std::wstring ObjectID;         //ID объекта-владельца
  std::wstring PropertyID;       //ID свойства
  std::wstring Value;            //значение
  std::wstring KeepInStack;      //хранить ресурс в стеке после компакта
  std::wstring SaveInFile;       //сохранять ли значение ресурса в файл на диске
};
//-------------------------------------------------------------------------------------------------------------------
```

Стек объектов описан как std::vector<RESOURCE> и включен в специальный класс RESOURCESTACK. Объектом в представлении интерпретатора является набор элементов соответствующего стека, для которых поле ObjectID одинаковое. Поле PropertyID обозначает свойство объекта, а поле Value – значение этого свойства. Ключевой особенностью ELI при работе с объектами является возможность добавлять свойства прямо «на лету». Набор методов, напротив, строго ограничен. Каждый объект содержит следующие методы:

Create(sym cathegory, sym ctor_args) – создает новый объект с указанной категорией. Аргумент ctor_args содержит список аргументов для конструктора класса.

Exist() – проверяет, присутствует ли в стеке объект, возвращает 1, если существует хотя бы один элемент стека объектов, для которого ObjectID равно имени объекта, использующего данный метод.

Have(sym prop_name) – проверяет существует ли свойство у объекта.

Add(sym prop_name, sym val) – добавляет свойство объекта.

Несмотря на то, что аргумент val метода Add() объявлен как sym, его можно использовать, чтобы передавать числовое значение. Таким же образом можно присваивать

свойствам объектов числовые значения, хотя поле Value структуры RESOURCE является строкой. Это достигается благодаря нестрогому соответствию типов языка.

Remove(sym prop_name) – удаляет свойство с именем prop_name.

Destroy() – удаляет созданный объект.

Keep(sym prop_name, sym istrue) – указывает нужно ли хранить свойство в стеке. Аргумент istrue может принимать значения 1 или 0.

Save(sym prop_name, sym istrue) – указывает нужно ли сохранять строку с этим свойством в файл. Аргумент istrue может принимать значения 1 или 0.

Execute(sym prop_name) – выполняет отложенный фрагмент кода, идентификатор которого содержится в свойстве prop_name. Идентификатор должен быть предварительно создан с помощью директивы #make. При вызове этого метода создается временная процедура с одним аргументом $this, в котором содержится имя объекта.

Важно: хотя метод **Execute** исполняет ту же функцию, что и директива **#run** и функция **_Run()**, между ними есть существенная разница. Директива #run и функция _Run() работают в рамках текущего контекста, в то время как метод Execute создает новый контекст и код, разворачивающийся при вызове метода, выполняется в рамках этого контекста.

Show() – выводит на экран все свойства объекта.

GetName() – возвращает имя объекта.

ExportIn(sym pPropNames, sym pPropVals) – экспортирует свойства объекта в два объекта-списка, с заданными именами. Каждое свойство объекта-списка pPropNames будет иметь числовой индекс, а значением этого свойства будет имя свойства исходного объекта. У объекта pPropVals значения свойств будут содержать значения свойств исходного объекта. Кроме того последним свойством обоих списков будет Count, содержащее количество индексных элементов списка.

Например:

```
//-------------------------------------------------------------------------------------------------------------
&Hero.Create(NPC, '');
&Hero.Add(Strength, 100);
&Hero.ExportIn(x, y);
//-------------------------------------------------------------------------------------------------------------
```

Результатом выполнения &x.Show() будет:

```
//-------------------------------------------------------------------------------------------------------------
0 = Active
1 = Strength
Count = 2
//-------------------------------------------------------------------------------------------------------------
```

Результатом выполнения &y.Show() будет:

```
//-----------------------------------------------------------------------------------------------------------------
0 = 1
1 = 100
Count = 2
//-----------------------------------------------------------------------------------------------------------------
```

Общий синтаксис при работе с объектами следующий:

```
//-----------------------------------------------------------------------------------------------------------------
&Hero.Create(NPC, '');
&Hero.Add(Strength, 100);
&Hero.Destroy();
//-----------------------------------------------------------------------------------------------------------------
```

Первая строка создает объект с именем &Hero и категорией NPC, вторая добавляет объекту числовое свойство Strength со значением 100, третья – уничтожает объект. У созданного объекта по умолчанию уже два служебных свойства: Owner со значением «<none>» и ObjectName со значением, которое соответствует имени из строки скрипта («Hero» в данному случае).

Обращаться к свойствам объектов можно с помощью служебного оператора «.», так же, как это происходит во многих языках программирования при работе с полями структур. Присвоение значений производится с помощью оператора «=».

```
//-----------------------------------------------------------------------------------------------------------------
$x = &Hero.Strength;
&Hero.Strength = $x * 2;
//-----------------------------------------------------------------------------------------------------------------
```

Имя объекта уникально, чувствительно к регистру и может содержать только буквы, цифры и символ «_». Имя объекта может определяться значением переменной (или нескольких) – такая конструкция называется псевдонимом.

```
//-----------------------------------------------------------------------------------------------------------------
$npcname = SuperHero;
&$npcname.Create (NPC, '');

$part1 = John;
$part2 = Connor;
&$part1$part2.Create (Player, '');
//-----------------------------------------------------------------------------------------------------------------
```

Процедуре можно передать имя объекта в качестве параметра. Для этого нужно в вызове процедуры опустить в аргументе спецсимвол &, а в теле процедуры использовать конструкцию &<имя аргумента процедуры> в качестве псевдонима объекта.

```
//-------------------------------------------------------------------------------------------------------------------
#procedure objfoo($arg1, $arg2)
{&$arg1.$arg2 = 5;}

&Hero.Create(NPC, '');
&Hero.Add(Strength, 100);
:objfoo(Hero, Strength);
//-------------------------------------------------------------------------------------------------------------------
```

Псевдоним свойства, так же, как псевдоним объекта, может быть составлен из значений переменных. Однако, стоит быть внимательным при использовании числовых переменных в псевдонимах объектов или свойств. Дело в том, что значения переменных, явно определенных как числа, хранятся с десятичной точкой и именно в таком виде они будут подставлены в псевдоним. Следует учитывать, что при присваивании свойству значения ELI сначала попытается привести правую часть выражения к числовому типу. Если же это невозможно – будет использован строковый тип данных.

Объекты классов

Выше были описаны принципы работы с простыми объектами, набор свойств которых задается пользователем для каждого экземпляра индивидуально. Кроме того, набор методов для простых объектов строго ограничен стандартными методами интерпретатора. В сущности, такие объекты можно назвать структурами. Однако, ELI поддерживает также возможность создания типизированных объектов по определенному шаблону. Такой шаблон называется классом.

Набор свойств класса определяется пользователем при объявлении. В дальнейшем, при создании объекта данного класса, все свойства, указанные при объявлении, будут добавлены к экземпляру объекта автоматически. Кроме того, класс позволяет включать в себя не только свойства, но и определенные пользователем методы, что выгодно отличает объект класса от простого объекта.

Пример простого класса, представляющего собой список:

```
//-------------------------------------------------------------------------------------------------------------------
#class List
{
  #property Next = 0;
  #public method AddItem($val){$x = &$this.Next; &$this.Add($x, $val); &$this.Next = ++1;}
  #public method Count(){#return &$this.Next;}
  #public method Change($ind, $val){&$this.$ind = $val;}
  #public method Get($ind){ #return &$this.$ind;}
}
//-------------------------------------------------------------------------------------------------------------------
```

Теперь после использования стандартного метода Create() с аргументом List будет создан объект, обладающий свойством Next, которое имеет значение 0.000 и четырьмя методами. После каждого вызова метода AddItem(), к свойствам объекта будет добавляться еще одно с указанным значением. Само собой, к объектам классов применимы все

встроенные методы ELI. Имя класса чувствительно к регистру, так что &obj_1.Create(List, '') и &obj_2.Create(list, '') создадут два разных объекта. Имя класса уникально, поэтому нельзя объявить два класса с одинаковым именем, но можно переопределить класс, удалив его с помощью специальной директивы

```
//-------------------------------------------------------------------------------------------------------------------
#drop class List;
//-------------------------------------------------------------------------------------------------------------------
```

или добавить и удалить свойства и методы

```
//-------------------------------------------------------------------------------------------------------------------
#modify class List
{
  #drop method Change;
  #public method Clear()
  {
    $i = 0;
    $cnt = &$this.Count();

    for ($i, $i < $cnt, +1)
      {&$this.Remove($i);}

    &$this.Next = 0;
  }
}
//-------------------------------------------------------------------------------------------------------------------
```

Важно: всем свойствам, при описании класса, должно быть задано значение по умолчанию.

Важно: изменения, сделанные при помощи директивы #modify class применяются только для объектов класса, объявленных после использования директивы. Объекты, созданные ранее, остаются без изменений.

Важно: если метод класса участвует в вычисляемом выражении, он обязательно должен возвращать некоторый результат с помощью директивы #return, иначе интерпретатор не сможет подставить результат выполнения метода в выражение.

Конструктор класса

Часто необходимо, чтобы свойства объекта класса были определены сразу после его создания. Для этого, при объявлении класса, можно указать метод, имя которого совпадает с именем класса, именно его интерпретатор будет считать конструктором. В теле конструктора можно определять и производить действия над любыми членами класса, но возвращать

результат – нельзя. Конструктор объявляется так же, как и любой метод класса. Список аргументов конструктора передается во втором аргументе встроенного метода Create().

Важно: конструктор класса должен объявляться как публичный, иначе метод Create() не сможет его вызвать.

```
//-------------------------------------------------------------------------------------------------------------
#class A
{
  #public method A($x){&$this.X = $x;}
  #property X = 0;
}

&obj.Create(A, 5);
//-------------------------------------------------------------------------------------------------------------
```

Если аргументов несколько их нужно передавать константной строкой.

```
//-------------------------------------------------------------------------------------------------------------
#class A
{
  #public method A($x, $y, $z){&$this.X = $x; &$this.Y = $y; &$this.Z = $z;}
  #property X = 0;
  #property Y = 0;
  #property Z = 0;
}

&obj.Create(A, '5, 6, 7');
//-------------------------------------------------------------------------------------------------------------
```

Деструктор класса

ELI рассматривает все объекты как набор данных, которые содержатся в единице трансляции, и зачастую так оно и есть, поэтому для уничтожения объекта достаточно удалить все его поля из стека объектов при помощи встроенного метода Destroy(). Однако некоторые из таких объектов могут содержать указатели или дескрипторы, которые ссылаются на адреса в куче (heap). Например, если метод объекта инкапсулирует функцию из сторонней dll или основного приложения, в которой создается объект при помощи оператора new. Конечно хорошим тоном для разработчика библиотеки (приложения) будет создание механизма очищения памяти таких объектов в куче, но этот механизм еще нужно запустить из контекста ELI. Для этого в функционал классов был добавлен деструктор.

Деструктор объявляется так же, как и любой другой метод класса. Аргументов не имеет. Не наследуется. Деструктор будет автоматически вызван во время вызова метода Destroy().

Важно: деструктор обязательно должен объявляться как публичный, в противном случае метод Destroy() не сможет вызвать его.

```
//-------------------------------------------------------------------------------------------------------------------
#class A
{
  #property X = 5;
  #public method ~A(){&$this.X = 0;}
}

&obj.Destroy();
//-------------------------------------------------------------------------------------------------------------------
```

Частные и публичные члены класса

Классы могут иметь как частные (закрытые) так и публичные (открытые) методы и свойства. По умолчанию, член класса, объявленный директивой #property или #method объявляется как частный (private). Чтобы объявить публичный член, необходимо использовать директиву #public property или #public method.

Кроме того, ELI поддерживает один из принципов ООП: инкапсуляцию. Это означает, что любые частные члены класса могут использоваться только внутри его методов.

Наследование классов

Классы могут наследовать свойства и методы других классов. Множественное наследование не поддерживается. Наследуются только публичные члены. Конструктор также наследуется, но его имя наследуется от родительского класса и не является конструктором по умолчанию для потомка. Тем не менее унаследованный конструктор может быть вызван, как и любой другой метод. Самый младший класс-потомок в иерархии унаследует конструкторы каждого из ее членов.

Класс-потомок может определить свой член с именем, соответствующим имени наследуемого члена, при этом предпочтение отдается определяемому, а не наследуемому члену.

Все унаследованные члены по умолчанию объявляются как публичные.

```
//-------------------------------------------------------------------------------------------------------------------
#class Point
{
  #public method Point($x, $y){&$this.pX = $x; &$this.pY = $y;}
  #public property pX = 0;
  #public property pY = 0;
  #public method SetX($pos){&$this.pX = $pos;}
  #public method SetY($pos){&$this.pY = $pos;}
}

#class 3DPoint : Point
```

```
{
  #property pZ = 0;
  #method SetZ($pos){&$this.pZ = $pos;}
}
//-----------------------------------------------------------------------------------------------------------------
```

В этом примере класс 3DPoint не имеет своего конструктора, но наследует конструктор от класса Point, так же как и все прочие члены. Таким образом, для того, чтобы полностью определить точку по трем координатам, объект класса 3DPoint может либо вызвать унаследованный конструктор Point(), либо использовать унаследованные методы SetX() и SetY(), после чего вызвать свой частный метод SetZ().

Вложенность объектов

Объекты классов могут использовать в качестве свойств другие объекты классов. Для этого в описании класса нужно объявить свойство, используя следующую конструкцию:

```
//-----------------------------------------------------------------------------------------------------------------
#class B
{
  #property Prop1 = #class A(0);
}
//-----------------------------------------------------------------------------------------------------------------
```

Где A это имя конструктора класса, а в скобках указан список аргументов конструктора. Если класс не имеет конструктора (или его вызов по каким-либо причинам не требуется), следует использовать такую конструкцию:

```
//-----------------------------------------------------------------------------------------------------------------
#class B
{
  #property Prop1 = #class A;
}
//-----------------------------------------------------------------------------------------------------------------
```

При каждом создании объекта класса B, интерпретатор создаст объект класса A с автоматически сгенерированным именем и присвоит свойству Prop1 имя этого объекта. После этого можно обращаться к членам вложенного объекта класса A через свойство Prop1. Например:

```
//-----------------------------------------------------------------------------------------------------------------
#class Point
{
  #public method Point($pos){&$this.Set($pos);}
  #property Pos = 0;
  #public method Set($pos){&$this.Pos = $pos;}
}
```

```
#class 2DPoint
{
  #public method 2DPoint($x, $y){&$this.Set($x, $y);}
  #public property X = #class Point(0);
  #public property Y = #class Point(0);
  #method Set($x, $y){&$this.X.Set($x); &$this.Y.Set($y);}
}

&point.Create(2DPoint, '5, 5');

&point.X.Pos = 10; //ошибка – свойство Pos является частным;
&point.Y.Set(20); //правильно;
//-------------------------------------------------------------------------------------------------------------
```

Автоматически сгенерированный объект отличается от пользовательского тем, что его служебное свойство Owner хранит имя объекта-владельца. При уничтожении объекта-владельца, все связанные с ним вложенные объекты также уничтожаются.

Существует еще один способ создания свойства-объекта: использование встроенного метода Add() и передачи ему в качестве второго аргумента конструкции типа #class Class(<аргументы конструктора>). Метод Add() добавляет публичное свойство.

```
//-------------------------------------------------------------------------------------------------------------
#class B
{
  #property Prop1 = #class A;
}

&obj.Create(B, '');
&obj.Add(Prop2, #class A);
//-------------------------------------------------------------------------------------------------------------
```

Важно: в связи с особенностями парсера, для передачи нескольких аргументов конструктора в данном случае следует использовать строковую константу.

```
//-------------------------------------------------------------------------------------------------------------
#class B
{
  #property Prop1 = #class 2DPoint(2, 2);
}

&obj.Create(B, '');
&obj.Add(Prop2, #class 2DPoint('5, 5'));
//-------------------------------------------------------------------------------------------------------------
```

Расширение функционала с помощью внешних библиотек

Конечно, ELI неспособен предоставить весь спектр возможностей, которые придут (или могут прийти) на ум пользователю. Именно на этот случай в нем предусмотрена возможность расширения функционала с помощью подключения внешних библиотек и использования их кода в своих скриптах. Конкретно речь идет об использования функций-оберток, объявленных во внешних библиотеках, так же, как это предусмотрено для запускающего приложения. Достаточно обернуть код в функцию-обертку при проектировании библиотеки, и импортировать ее в ELI во время работы скрипта, после чего эта функция будет доступна для вызова до окончания работы интерпретатора или до освобождения внешней библиотеки. Разумеется, для того, чтобы библиотека была совместима с ELI, при ее проектировании необходимо включить в проект заголовочный файл **eli_interface.h**, содержащий виртуальный интерфейс.

```
//-------------------------------------------------------------------------------------------------------------------
$hinst = _ConnectLib(.\testextdll.dll);
//-------------------------------------------------------------------------------------------------------------------
```

Этот код подключит к интерпретатору библиотеку testextdll.dll из каталога, в котором находится библиотека ELI, и присвоит переменной $hinst ее дескриптор. Предположим, в библиотеке имеется функция

```
//-------------------------------------------------------------------------------------------------------------------
__declspec(dllexport) void __stdcall Summ(void *p)
{
  ELI_INTERFACE *ep = (ELI_INTERFACE*)p;

  float num = ep->GetParamToFloat("pNum1") + ep->GetParamToFloat("pNum2");
  wchar_t res[10];
  swprintf(res, "%.2f", num);

  ep->SetFunctionResult(ep->GetCurrentFuncName(), res);
}
//-------------------------------------------------------------------------------------------------------------------
```

которая выполняет сложение двух чисел с плавающей точкой и возвращает результат. Чтобы предоставить интерпретатору возможность вызова этой функции, необходимо включить в текст скрипта следующий код:

```
//-------------------------------------------------------------------------------------------------------------------
_ImportFunc($hinst, Summ, '_summ', 'num pNum1,num pNum2');
//-------------------------------------------------------------------------------------------------------------------
```

Здесь **$hinst** – это дескриптор внешней библиотеки, **Summ** – имя функции из внешней библиотеки, **_summ** – имя, которое будет использовать интерпретатор, а строка **'float pNum1,float pNum2'** определяет список параметров функции. Естественно имена параметров должны совпадать с теми, которые использованы в теле импортируемой функции. В случае неудачи функция _ImportFunc() возвращает 0.

Важно: метод GetCurrentFuncName() в теле функции из внешней библиотеки нужен для того, чтобы узнать у интерпретатора имя функции, которое он сопоставил функции из внешней библиотеки.

Важно: внутреннее имя функции, используемое ELI, может быть любым, но необходимо следить, чтобы оно не совпадало с именем существующей функции.

Важно: имя функции, которая экспортируется DLL, может меняться в зависимости от того, как компилятор декорирует имена при использовании спецификатора __stdcall. Например, компилятор Embarcadero C++ Builder экспортирует функцию void __stdcall Foo(void*) в виде «Foo», а MinGW в виде «Foo@4». Если функция интерпретатора _ImportFunc() сообщает о неудаче, откорректируйте имя экспортной функции в том месте, где вызывается _ImportFunc().

Теперь можно использовать импортированную функцию в скрипте.

```
//-----------------------------------------------------------------------------------------------------------
$res = _summ(2.2, 3.4);
//-----------------------------------------------------------------------------------------------------------
```

После чего можно освободить библиотеку, когда она больше не нужна.

```
//-----------------------------------------------------------------------------------------------------------
_FreeLib($hinst);
//-----------------------------------------------------------------------------------------------------------
```

Важно: все функции, импортированные в интерпретатор, будут удалены из стека при вызове _FreeLib().

Все внешние библиотеки, подключенные к интерпретатору, будут автоматически освобождены при отключении библиотеки ELI.

Декорации

Декорацией в языке ELI называется метка, декорирующая собой некий набор конструкций языка, от переменных до целых выражений. В каком-то смысле декорация работает по принципу директивы препроцессора #define в C++. При трансляции строки, находящиеся в ней декорации будут обработаны перед всеми последующими действиями и вместо них в строку будут подставлены декорированные конструкции.

```
//-----------------------------------------------------------------------------------------------------------
$x = 10;
?ref = $x;
//-----------------------------------------------------------------------------------------------------------
```

Декорация ?ref скрывает собой конструкцию, которая состоит из переменной $x. В дальнейшем, во всех транслируемых строках вместо ?ref будет подставлена переменная $x. В результате трансляции таких строк

```
//-------------------------------------------------------------------------------------------------------------
$x = 10;
?ref = $x;
$y = ?ref + 5;
//-------------------------------------------------------------------------------------------------------------
```

переменная $y будет равна 15. Интерпретатор обработает правую часть выражения, подставит вместо декорации скрытую конструкцию, после чего вычислит значение выражения.

Декорации удобно использовать для хранения шаблонных конструкций вычисления, например, если значение переменной вычисляется по сложной формуле.

```
//-------------------------------------------------------------------------------------------------------------
?hip = $a * $a + $b * $b;

$a = 3;
$b = 4;
$sqc = ?hip;
$sqx = ?hip;
//-------------------------------------------------------------------------------------------------------------
```

Такая форма записи позволяет значительно сократить код, сделать его более читаемым. Следует, однако, помнить, что интерпретатор не следит за корректностью декорируемой конструкции. Если в приведенном выше примере переменные $a или $b не были предварительно определены, интерпретатор выдаст ошибку трансляции.

Поскольку декорации содержат имена, а не значения переменных, их удобно использовать в тех случаях, когда требуется работать с самой переменной, а не ее значением, например в случае функции _free(). Кроме того, декорации позволяют передавать набор аргументов, поэтому вполне законна такая конструкция:

```
//-------------------------------------------------------------------------------------------------------------
$a = hello;
$b = ' ';
$c = world;

?ref = $a, $b, $c;

#procedure Hello($x, $y, $z)
{
  _writeout($x $y $z); _writeout(#endl);
}
```

```
:Hello(?ref);
//-------------------------------------------------------------------------------------------------------------------
```

В результате на экран будет выведено «hello world».

Декорация также может использоваться для создания псевдонимов объектов или свойств, однако такую декорацию нельзя передать процедуре. Можно декорировать даже имена методов и функций, но использовать такие декорации можно только в правой части выражения. Интерпретатор позволяет применять имя декорации в левой части выражения только в одном случае: при описании декорации.

Такая конструкция допустима:

```
//-------------------------------------------------------------------------------------------------------------------
?f = _random;

$a = ?f(10);

#class A
{
  #method Foo(){#return 10;}
}

&cl.Create(A, '');

?f = &cl.Foo();

$b = ?f;
//-------------------------------------------------------------------------------------------------------------------
```

А такая – нет:

```
//-------------------------------------------------------------------------------------------------------------------
?f = _writeout;

?f(10);
//-------------------------------------------------------------------------------------------------------------------
```

Кроме того, следует помнить, что декорации могут скрывать только законченные конструкции и не поддерживают декорирование фрагментов кода. Например, нельзя задекорировать описание класса или объявление переменной. Декорирование процедур также лишено смысла, так как процедуры не используются в правой части выражения.