Назва: Extern Logic Interpreter (Інтерпретатор Зовнішньої Логіки).

Принцип роботи: порядкова інтерпретація команд.

Форма: динамічно зв'язана бібліотека (dll).

Призначення

Extern Logic Interpreter (надалі ELI) був розроблений щоб винести логічні маніпуляції за межі коду самої аплікації. Його безпосереднє призначення – максимально абстрагувати програму від логічних операцій, які можна було б з легкістю модифікувати без перекомпіляції основної програми. В ідеалі аплікація, що використовує ELI, може складатись тільки з уособлених механізми взаємодії із оточенням (наприклад, функції для роботи із файлами, виводу інформації на екран тощо), а також спеціальні функції-обгортки, що нададуть ELI доступ до цих механізмів. Зв'язування всього цього у певний алгоритм буде виконувати сам інтерпретатор.

Механізм роботи

ELI являє собою порядковий інтерпретатор, що використовує команди за синтаксисом подібні до мов програмування високого рівня, такими як, наприклад, C++. У більшості випадків трансляція рядків виконується послідовно, виключенням є оголошення користувацьких процедур, умов та циклів, тіла яких парсяться окремо, до запуску трансляції основного скрипту.

Одиницею трансляції вважається скрипт. Скрипт складається з рядків, що розділені символом «;». ELI може отримати текст скрипту безпосередньо від основної аплікації, або завантажити із зовнішнього файлу. Одиниць трансляції може бути декілька, їх код додається до трансляції за допомогою директиви #include або функції Run().

Перед трансляцією інтерпретатор виконує підготовку скрипту: виділяє у тексті строкові константи та фрагменти коду, що містяться у фігурних дужках «{}». Фрагменти коду замінюються у тексті скрипту на спеціальні ідентифікатори, а їх тіла потрапляють у спеціальний стек. Такі фрагменти називаються відкладеними, їх трансляція починається тільки за безпосереднім викликом.

Важливо: у файлі може бути тільки один скрипт.
Важливо: файл скрипту не повинен включати в себе порожні рядки в кінці.

Текст скрипту може містити коментарі. Закоментований рядок повинен починатися двома символами прямого слеша «//» та закінчуватися крапкой з комою «;».

ELI надає користувачу наступний функціонал:

Директиви: прямі команди інтерпретатору.
Змінні: внутрішні змінні інтерпретатора, існують на протязі роботи скрипту. Область видимості: поточний контекст.
Прості та складні математичні операції з числами.

Цикли.

Умови.

Функції: вказівники на функції-обгортки з основної аплікації. Дозволяють використовувати алгоритми, що містяться в основній аплікації. Можуть передавати результати своєї роботи з основної аплікації у скрипт. Область видимості: глобальна, тобто до відключення бібліотеки.

Процедури: фрагменти коду, що викликаються (та транслюються) лише при безпосередньому звернені. В протилежність функціям не повертають значень. Область видимості: глобальна.

Об'єкти: абстракції, що представляють собою набір типізованих даних та дозволяють працювати з ними, як з одним цілим, використовуючи властивості та методи. Область видимості: глобальна.

Стек параметрів: прошарок між основною аплікацією та ELI, використовується для обміну даними. Являє собою динамічно змінювану множину типізованих структур, що характеризують поняття «параметр».

Вивід стека повідомлень: помилки та службові повідомлення, що виникають у процесі трансляції, зберігаються у спеціальну строкову змінну, яка може бути передана в основну аплікацію за допомогою відповідної функції інтерпретатора.

Вивід стека змінних: вміст стека змінних у форматованому вигляді зберігається у строкову змінну.

Вивід стека функцій: збереження у строкову змінну вміст стека функцій-обгорток. Зберігаються не тільки описані у бібліотеці інтерпретатора функції, але й ті, що оголошені в основній аплікації.

Вивід стека параметрів.

Вивід стека об’єктів.

Лог трансляції: опція, що задається під час запуску трансляції скрипту. Вмикає логування строк у файл, туди ж зберігає повідомлення про помилки. Логування сповільнює процес трансляції.

Лог роботи інтерпретатора: опція задається за допомогою спеціальної функції інтерпретатора, або за допомогою методу ELI::SetDebug(). Вмикає логування дій самого ELI, спрямовує вивід або у файл, або у STDOUT. Логування значно сповільнює роботу інтерпретатора.

Реалізація

ELI написаний на C++, використовує контейнери std::vector<std::wstring> для збереження рядків скрипту, а також спеціальні класи, що описують стеки змінних, параметрів, функцій, процедур, об’єктів та окремих фрагментів коду. Екземпляри цих класів включені до класу **ELI**. Кодування рядків – ANSI. Внутрішня мова ELI по суті є обгорткою над C++ кодом. Кожен рядок за допомогою спеціального алгоритму інтерпретується у набір команд C++, після чого виконується. Аплікація, за допомогою спеціального методу класу ELI, передає інтерпретатору текст скрипту (або ім’я файлу, який потрібно транслювати), а також необов’язковий набір вхідних параметрів. ELI виконує інтерпретацію рядків, після чого повертає результат роботи (тип **const wchar_t***), якщо це потрібно. У разі помилки повертає константну строку «-err-».

Інтеграція

Основний код ELI побудований у вигляд динамічно зв’язаної бібліотеки, яку необхідно підключити до основної аплікації. Щоб працювати із функціоналом ELI, доведеться інтегрувати об’єкт класу ELI з бібліотеки інтерпретатора. Для цього використовується абстрактний інтерфейс, що оголошений у заголовковому файлі **eli_interface.h,** що розповсюджується разом з інтерпретатором.

Файл містить інтерфейсний клас **ELI_INTERFACE**, що є батьківським для оголошеного у dll класу **ELI**. Також у файлі оголошені дві функції-фабрики, за допомогою яких проходить створення та знищення інтерфейсів.

```
//---------------------------------------------------------------------------------------------------------------
__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);
//---------------------------------------------------------------------------------------------------------------
```

Функції повертають 1 у разі успішного виконання, або 0 в разі помилки.

Після підключення файлу eli_interface.h до проекту основної аплікації необхідно визначити вказівники на функції-фабрики та ініціалізувати їх.

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

ELI_INTERFACE *eIface;

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

Після цього ініціалізувати об’єкт-інтерфейс за допомогою визначеної раніше функції.

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

Тепер достатньо використати вказівник **eIface**, щоб отримати доступ до методів класу **ELI**, що описані всередині бібліотеки. Таким чином у аплікації можна використовувати декілька екземплярів ELI, які будуть діяти незалежно один від одного.

Перелік методів класу ELI_INTERFACE

const wchar_t * __stdcall GetVersion() – повертає поточну версію ELI.

const wchar_t * __stdcall ShowVarStack() – повертає вміст стека змінних у текстовому вигляді.

const wchar_t * __stdcall ShowObjStack() – повертає вміст стека об’єктів у текстовому вигляді.

const wchar_t * __stdcall ShowClassStack() – повертає вміст стека класів у текстовому вигляді.

const wchar_t * __stdcall ShowProcStack() – повертає вміст стека процедур у текстовому вигляді.

const wchar_t * __stdcall ShowFuncStack() – повертає вміст стека функцій у текстовому вигляді.

const wchar_t * __stdcall ShowParamStack() – повертає вміст стека параметрів у текстовому вигляді.

const wchar_t * __stdcall ShowFragmentStack() – повертає вміст стека відкладених фрагментів коду.

const wchar_t * __stdcall ShowInfoMessages() – повертає перелік повідомлень інтерпретатора.

const wchar_t * __stdcall RunScript(const wchar_t *imptext, const wchar_t *parameter, bool log) – запускає трансляцію скрипту. Аргумент log є прапорцем використання лога. Аргумент parameter може містити декілька значень, розділених символом «|».

const wchar_t * __stdcall RunScriptFromFile(const wchar_t *filepath, const wchar_t *parameter, bool log) – запускає трансляцію з файлу. Аргумент log є прапорцем використання лога. Аргумент parameter може містити декілька значень, розділених символом «|». Шлях до файлу може бути як абсолютним, так і умовним (.\<ім’я_файлу>). У цьому разі пошук ведеться у директорії де розташована dll інтерпретатора.

void __stdcall SetDebug(bool enable_dbg, bool in_file) – вмикає та вимикає запис діагностичного логу роботи самого інтерпретатора, а також вказує куди саме треба його писати.

bool __stdcall DebugEnabled() – перевіряє, чи використовується режим логування.

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

void __stdcall DeleteFunction(const wchar_t *name) – видаляє функцію зі стеку.

void __stdcall CallFunction(const wchar_t *name) – викликає функцію.

wchar_t * __stdcall GetFunctionResult(const wchar_t *name) – перетворює результат функції зі стеку ELI у строку та повертає вказівник на неї. У разі помилки повертає NULL.

void __stdcall SetFunctionResult(const wchar_t *name, const wchar_t* result) – записує результат функції у стек ELI.

void __stdcall SetParam(const wchar_t *name, const wchar_t *new_val) – додає у стек новий параметр, або змінює значення існуючого.

int __stdcall GetParamToInt(const wchar_t *name) – повертає значення параметру, як integer.

float __stdcall GetParamToFloat(const wchar_t *name) – повертає значення параметру, як float.

const wchar_t * __stdcall GetParamToStr(const wchar_t *name) – повертає значення параметру як const wchar_t*. У разі помилки повертає NULL.

const wchar_t * __stdcall GetCurrentFuncName() – повертає ім’я поточної функції-обгортки, яку ELI обробляє у дану мить. Може використовуватись для того, щоб повідомити функції-обгортці із зовнішньої бібліотеки, що була підключена під час роботи скрипту, яке внутрішнє ім’я надав їй ELI.

void __stdcall AddToLog(const wchar_t *msg) – додає запис в перелік повідомлень інтерпретатора.

Оголошення функції-обгортки

Для того щоб інтерпретатор міг використовувати код основної аплікації, цей код потрібно загорнути в спеціальні функції. Функції-обгортки не повинні бути членами класу, або оголошуватись у класі зі специфікатором **static**.

Важливо: тип результату та аргументів функції-обгортки строго регламентований, тому що для роботи з цими функціями використовується вказівник, визначений у файлі eli_interface.h:

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

Аргумент void* використовується, аби передати аплікації вказівник на екземпляр об’єкта ELI, що викликав функцію. Це дозволяє різним екземплярам інтерпретатора звертатись до функцій основної аплікації незалежно один від одного.

```
//-----------------------------------------------------------------------------------------------------------------
```

```
void __stdcall foo(void *p)
{
  //приводимо вказівник p до типу ELI_INTERFACE
  ELI_INTERFACE *ELI = (ELI_INTERFACE*)p;

  int x = ELI->GetParamToInt("pInd"); //отримуємо параметр зі стеку dll

  /*якийсь код основної аплікації*/

  ELI->SeFunctionResult("_foo", "0"); //встановлення потрібного результату у стек
}
//---------------------------------------------------------------------------------------------------------------
```

У цьому прикладі зі стеку зчитується параметр pInd та перетворюється у тип integer. Після цього виконується код основної аплікації, а результат, за допомогою інтерфейсу, передається до бібліотеки інтерпретатора, яке ELI передасть скрипту.

Важливо: використання методу ELI_INTERFACE::SetFunctionResult() в кінці тіла функції є обов’язковим. Кожна функція-обгортка повинна повертати результат у стек, інтерпретатор буде очікувати на це. Тип результату завжди є строковим, інтерпретатор сам перетворить його у потрібний тип, якщо це буде необхідно.

Після оголошення потрібно додати функцію до стеку.

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

«_foo» це ім’я функції, яке буде використовувати ELI. Символ «_» є обов’язковим, з його допомогою інтерпретатор визначає, що далі у тексті скрипту розташоване ім’я та аргументи функції. Ім’я функції нечутливе до регістру.

Строка «num pInd» описує список аргументів функції та їх тип. Якщо аргументів декілька, потрібно вказувати їх через кому. Пробіли ставляться лише між типом аргументу та його ім’ям. Один аргумент не виділяється комою. Імена аргументів не чутливі до регістру та не є строго регламентованими.

Тепер, коли інтерпретатор виявить в тілі скрипту вираз типу «_foo(12)», він звернеться до стеку функцій, знайде там функцію _foo(), перевірить кількість та тип аргументів, що передає їй скрипт, та викличе функцію-обгортку foo() із основної аплікації, та передасть їй вказівник на свій екземпляр. Після чого отримає результат та поверне його у рядок скрипта, у якому відбувся виклик функції.

Крім того, завжди можна передати скрипту будь-які дані у будь-якому місці коду аплікації, використавши стек параметрів.

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

Метод ELI_INTERFACE::SetParam() додає новий параметр в стек тільки якщо той не містить елемента з таким ім’ям. У випадку збігу імен, інтерпретатор оновить значення існуючого параметра. Така поведінка використовується для економії ресурсів. У тілі скрипту отримати параметр зі стеку можна за допомогою відповідної вбудованої функції. Вхідні параметри скрипту також заносяться до стеку параметрів з перевизначеними іменами, що мають вигляд типу INPRM<порядковий номер>.

Синтаксис внутрішньої мови

Типи даних

Внутрішня мова ELI оперує двома типами даних.
num – числовий тип.
sym – символьний тип.

Числовий тип використовується для роботи як з цілими так і з дійсними числами. За замовчуванням всі числові значення розглядаються інтерпретатором як дійсні. Роздільником цілої та дробової частини є крапка «.». Точність числа: 3 знаки після коми. Перетворення до цілочислового вигляду виконується за допомогою вбудованих функцій _round() та _int(), або автоматично, якщо того потребує алгоритм. Наприклад, якщо аргумент функції має тип int, то змінна типу num, яка була передана функції, буде автоматично перетворена у цілочисловий тип.

Внутрішня мова ELI є мовою з нестрогою відповідністю типів. Якщо у виразі строго не вказано, що потрібно використовувати числовий тип, значення буде розглянуте інтерпретатором як текст. Так, наприклад, строковій змінній можна надати значення 21, при цьому воно буде перетворене на текст. Проте зворотня операція, наприклад, встановлення числової змінної у значення '22', призведе до помилки та зупинки трансляції.

Змінні

Оголошення змінної виконується за допомогою запису типу:

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

Де $n – ім’я змінної, num – тип. Після цього йде значеннння, яким ініціалізується змінна. Відсутність значення призводить до ініціалізіції значенням за замовчуванням. У даному випадку, після трансляції рядка, у стек буде додано числову змінну $n, що дорівнює 0.00, та строкову $s, що є порожньою строкою, тобто «». Змінна може бути ініціалізована константним значенням, скалярним виразом, іншою змінною, властивістю об’єкта та результатом функції або методу об’єкта. Строкова змінна може бути ініціалізована значеннями числових змінних, при цьому вони будуть перетворені на текст.

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

Тут змінна $s буде ініціалізована строковим значенням «100.000», яке є результатом конкатенації строкової константи «10» та приведеного до строкового типу значення числової змінної $n, тобто «0.000». Окрім того, конструкція $str = sym <arg1> + <arg2> сприймає вираз у правій частині як строку «2+2», проте результатом конструкції $str = 2 + 2 буде «22», тобто конкатенація.

Символ «$» є обов'язковою частиною імені, це прапорець, який говорить ELI, що далі розташоване ім'я змінної. Дозволяється ініціалізація однієї змінної на рядок.

Тип даних у оголошення змінної можна опустити та скористуватись конструкцією типу $var = <значення>, у такому разі для змінної буде обраний тип даних на основі обчислення правої частини виразу.

Для встановлення значення змінної використовується наступна конструкція:

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

У правій частині виразу можуть бути: константа, допустимий для цього типу даних скалярний вираз (добуток, множення, конкатенація тощо), властивість або метод об'єкта та результат функції. Причому у даному випадку змінній $s буде надане значення «sometext», тобто інтерпретатор опустить пробіл. Утім, якщо помістити значення в одинарні лапки, інтерпретатор вважатиме цей фрагмент коду за константу, тож встановить для змінної значення «some text».

Строкові константи відстежуються інтерпретатором на стадії аналізу тексту скрипту та розміщаються у стеку змінних під спеціальним ім'ям. В подальшому, коли транслюється рядок, що містить таку змінну, ELI розміщає замість неї значення зі стеку. Строкові константи зберігаються у стеку в одному екземплярі. Якщо у коді скрипту декілька разів зустрічається одна й та сама строкова константа, інтерпретатор у всіх випадках підставить значення одного відповідного елемента стека.

Важливо: треба бути обережним із використанням у строкових константах символу «$», парсер ELI зреагує на нього, як на символ-маркер змінної.

Строкові змінні, окрім конкатенації (позначається символом «+»), підтримують пряме послідовне об'єднання значень. Взагалі, при роботі зі строковим типом, весь вміст правої частини виразу розглядається інтерпретатором як текст. Будь-які символи (окрім спеціальних), строкові константи, значення функцій, методів, властивостей та змінних будуть приведені до типу sym та включенні у одну строку. Включення в строку відбувається послідовно у тому порядку, в якому були розташовані значення у виразі. Пробіли (якщо вони знаходяться не у строкових константах) будуть проігноровані. Будь-які символи математичних операцій (окрім «+») вважаються звичайними символами.

Операції

ELI підтримує усі базові математичні операції для числового типу («+», «-», «*», «/»), включно з оператором пріоритету «()». Крім того для чисел доступні операції порівняння, такі як:

= встановлення значення;
== дорівнює;
> більше;
< менше;
>= більше або дорівнює;
<= менше або дорівнює;
!= не дорівнює;

Також до числових змінних можна застосувати операцію інкременту («++») та декременту («--»).

Для строкового типу допустима тільки конкатенація «+» та встановлення значення «=».

Умови

```
//-------------------------------------------------------------------------------------------------------------------
if (<умова>)
  {
    <перелік дій>
  }
else if (<умова>)
  {
    <перелік дій>
  }
else
  {
    <перелік дій>
  }
//-------------------------------------------------------------------------------------------------------------------
```

ELI підтримує розгалужені умови (if - else if - else), а також вкладеність умов.

Синтаксис конструкції наступний:

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

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

```
    $i = 0;
  }
else
  {
    $i = 1;
  }
//-------------------------------------------------------------------------------------------------------------
```

У якості параметрів виразу, що проходить перевірку, можуть бути використані: число, числова змінна або властивість об’єкту, а також функції та методи об’єктів, результат яких може бути приведений до числового типу. При обчислені, дробова частина значень також враховується, тому у наведеному вище прикладі $i буде більше ніж 5. Ліва і права частина виразу можуть порівнюватись за допомогою будь-якої операції порівняння. Якщо у виразі відсутній символ операції порівняння, вираз буде перевірений на істинність. Вираз вважається істинним, якщо результат його обчислення більше ніж 0.

Конструкція select

```
//-------------------------------------------------------------------------------------------------------------
select (<параметр>)
{
  when <значення1> then {<перелік дій>}
  when <значення2> then {<перелік дій>}
  ...
}
//-------------------------------------------------------------------------------------------------------------
```

Альтернативою розгалуженим умовам if-else слугує конструкція select. Суть її полягає у перевірці переданого вхідного параметру на відповідність значенням, що містяться у виконавчому блоці. Перевірка здійснюється за допомогою спеціального службового слова **when**, а опис дій, які потрібно виконати, якщо виконується рівність, розташований після службового слова **then**.

У якості параметра конструкції select може виступати: змінна, результат функції, властивість об’єкту або результат його методу. Скалярні вирази не допускаються. Значення, з якими порівнюється параметр select, повинні бути константами. Синтаксис дозволяє змішувати числові та строкові константи у межах конструкції, таким чином параметр переданий до select можна порівняти як із числом 2, так і зі строковою константою «два». Звісно, числову змінну не можна порівняти зі строкою, тож в цьому випадку інтерпретатор просто відкине цей блок when...then та перейде до наступного.

У виконавчому блоці конструкції select, окрім блоків-перевірок when...then можуть міститися будь-які інші дії, які будуть виконані після всіх блоків-перевірок.

```
//-------------------------------------------------------------------------------------------------------------
$res = _foo(); //$res отримує певне значення;

select ($res)
{
```

```
  when 10 then {$res = $res + 2;}
  when 21 then {_return(2);}

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

Область видимості виконавчого блоку конструкції – глобальна. Це означає, що всі операції зі змінною $res з попереднього прикладу, за умови істинності першого перевірочного блоку, матимуть вплив на результати виконання наступних перевірочних блоків.

Цикли

Інтерпретатор надає користувачу три типа циклів: **for**, **while** та **count** (останній є спрощеною реалізацією for). Цикли можуть бути вкладеними. У якості параметрів умов циклів (окрім count) можуть бути: числа, числові змінні або властивості об’єктів, а також функції чи методи об’єктів, що повертають результат, що може бути інтерпретований як число. Змінні, що оголошені у циклі, будуть ініціалізуватись на кожній ітерації.

Цикл for

```
//-------------------------------------------------------------------------------------------------------------
for (<початкове значення>, <умова виконання>, <крок>)
{<якісь дії>}
//-------------------------------------------------------------------------------------------------------------
```

Виконує код, що міститься між фігурними дужками, доки операція порівняння між першим та другим аргументами умови є істинною. Третій параметр являє собою крок, на який змінюється значення першого параметру умови. Крок може бути як цілим так і дробовим числом, а знаки «+» чи «-» вказують на спосіб зміни першого параметру. Другий параметр складається зі знаку операції порівняння та числового значення, що окреслює верхній поріг умови. Допустимі операції порівняння: «>», «<», «>=» та «<=». Перший параметр може задаватись числовою константою, результатом функції або методу, а також властивістю об’єкта або змінною, що мають числовий тип. Якщо використовуються змінна чи властивість об’єкта, вони повинні бути ініціалізовані до початку циклу. Крім того, їх значення буде змінюватися після кожної ітерації, тому використання властивості об’єкта, можливо, є не найкращою ідеєю.

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

for (0, <= 10, -1)
{<певні дії>}
//-------------------------------------------------------------------------------------------------------------
```

Цикл while

```
//-----------------------------------------------------------------------------------------------------------------
while(<умова>)
{<певні дії>}
//-----------------------------------------------------------------------------------------------------------------
```

Виконує код, що міститься між фігурними дужками, доки операція порівняння між першим та другим параметром є істинною. В протилежність попередньому циклу, while не містить кроку, що змінює параметри умови. Умова перевіряється на істину перед кожною ітерацією. Якщо умова неістинна ще до першої ітерації – тіло циклу не виконується. Обидва параметри умови можуть бути числовими константами та змінними, результатами функцій або методів об’єктів та властивостями об’єктів, значення яких може бути інтерпретоване як число. Допустимі операції порівняння: «>», «<», «>=», «<=», «!=» та «==».

```
//-----------------------------------------------------------------------------------------------------------------
$i = 1;
while($i != 100)
{$i = _random(100);}
//-----------------------------------------------------------------------------------------------------------------
```

Цикл count

```
//-----------------------------------------------------------------------------------------------------------------
count(<лічильник>)
{<певні дії>}
//-----------------------------------------------------------------------------------------------------------------
```

Спрощена версія циклу for. Єдиний параметр вказує на кількість ітерацій виконання тіла циклу. Параметр може бути числовою константою, результатом функції або метода та числовою властивістю об’єкту чи змінною, яка повинна ініціалізуватись певним значенням до початку виконання циклу. Параметр обов’язково повинен бути позитивним значенням.

Директиви

Директиви це службові команди інтерпретатора, що описують певні дії, що є обов’язковими для виконання. У ELI використовуються наступні директиви:

#begin – відмічає початок скрипту. Обов’язкова.

#end – відмічає кінець скрипту. Обов’язкова.

#include <ім’я файлу> – долучає код, що міститься у зовнішньому файлі, до скрипту. Усі змінні, оголошені у вкладеному файлі, ініціалізуються в поточному активному стеку.

#exit – примусове завершення скрипту.

#procedure <ім’я процедури>(<аргументи>){**<тіло процедури>**} – оголошує користувацьку процедуру.

#drop procedure <ім’я процедури> – видаляє користувацьку процедуру.

#make <ім’я змінної> {<певний код>} – передає змінній ідентифікатор відкладеного фрагмента кода. В подальшому цей ідентифікатор можна передати іншій змінній або властивості об’єкта.

#run <ім’я змінної> – виконує фрагмент коду, ідентифікатор якого міститься у змінній. Усі змінні, що були створені під час виконання цього фрагменту, будуть розміщені у стеку, який є активним для контексту де була викликана директива #run. Після трансляції, фрагмент буде видалено зі стеку фрагментів.

#class <ім’я класу> {тіло класу} – створює новий клас об’єктів.

#modify class <ім’я класу> {тіло класу} – змінює існуючий клас об’єктів.

#property <ім’я властивості>=<значення> – додає нову приватну властивість в обраний клас. Застосовується лише всередині коду, що описує тіло класу.

#public property <ім’я властивості>=<значення> – додає нову публічну властивість в обраний клас. Застосовується лише всередині коду, що описує тіло класу.

#method <ім’я методу>(<аргументи>){**<тіло методу>**} – додає новий приватний метод в обраний клас. Застосовується лише всередині коду, що описує тіло класу.

#public method <ім’я методу>(<аргументи>){**<тіло методу>**} – додає новий публічний метод в обраний клас. Застосовується лише всередині коду, що описує тіло класу.

#drop property <ім’я властивості> – видаляє властивість з класу.

#drop method <ім’я метода> – видаляє метод з класу.

#drop class <ім’я класу> – видаляє клас об’єктів.

#return <значення> – повертає результат виконання методу класу.

#protect {<певний код>} – виконує захищений фрагмент коду. Код, що міститься у цьому фрагменті, буде виконаний у будь-якому разі та не зупинить трансляцію, навіть через критичне виключення. Усі помилки тим не менш будуть додані у лог трансляції.

#trigger <умова> {<певний код>} – оголошує тригер. Тригер це відкладений фрагмент коду, який буде виконано, якщо умова, що заявлена у тригері, буде істиною. Тригер оголошується один раз та перевіряється на спрацювання після трансляції кожного

рядка скрипту. Виконання тригеру ніяк не впливає на трансляцію основного скрипту, навіть якщо під час трансляції тіла тригера були знайдені помилки. або викинуто виключення.

#drop trigger <умова> – видаляє тригер.

#set {<директива>} – налаштовує інтерпретатор ELI. У тілі директиви використовуються декілька службових директив.

#cnum | #!cnum – вмикає/вимикає парсинг числових констант. Доступно лише у тілі директиви #set. За замовчуванням ввімкнено.

#csym | #!csym – вмикає/вимикає парсинг символьних констант. Доступно лише у тілі директиви #set. За замовчуванням ввімкнено.

#keepobjects | #!keepobjects – вказує зберігати/не зберігати вміст стеку класів після трансляції скрипту. Доступно лише у тілі директиви #set. За замовчуванням ввімкнено.

#keepclasses | #!keepclasses – вказує зберігати/не зберігати вміст стеку об'єктів після трансляції скрипту. Доступно лише у тілі директиви #set. За замовчуванням ввімкнено.

Функції

ELI може використовувати будь-які функції, що описані у користувацькій аплікації, за умови, що вони оголошені у стеку функцій. Крім того до складу ELI включені декілька вбудованих функцій:

_random(num pArea) – повертає псевдовипадкове значення з діапазону pArea.

_round(num pNumber, num pPrecision) – повертає результат округлення значення аргументу pNumber. Аргумент pPrecision вказує на точність округлення (від 0 до 2 знаків після коми).

_int(num pNumber) – приводить значення аргументу pNumber до цілочислового типу.

_strlen(sym pStr) – повертає довжину строки.

_streq(sym pStr1, sym pStr2) – порівнює строки, повертає 1 якщо строки рівні, 0 якщо ні. Враховує регістр.

_istreq(sym pStr1, sym pStr2) – порівнює строки, повертає 1 якщо строки рівні, 0 якщо ні. Не враховує регістр.

_substr(sym pTargetStr, num pPos, num pCount) – повертає субстроку.

_return(sym pReturnVal) – завершує скрипт та встановлює результат виконання.

_throw(sym pException) – завершення трансляції із додаванням до логу користувацького виключення із довільним текстом.

_free(sym pVarName) – видаляє зі стека змінну, повертає 1 якщо успішно, в іншому випадку повертає 0.

_LoadObjStack(sym pFilePath, num pClear) – завантажує вміст стеку об’єктів з файлу, повертає 1 якщо успішно, 0 – в іншому випадку. Якщо pClear > 0 стек об’єктів очищується перед завантаженням. Шлях до файлу може бути як абсолютний так умовний («.\ <ім’я файлу>»). У цьому випадку пошук ведеться в директорії де розташовано dll інтерпретатора.

_SaveObjStack(sym pFilePath) – зберігає вміст стеку об’єктів у файл, повертає 1 якщо успішно, 0 – в іншому випадку. Шлях до файлу може бути як абсолютний так умовний («.\ <ім’я файлу>»). У цьому випадку пошук ведеться в директорії де розташовано dll інтерпретатора.

_SaveObjects(sym pFilePath, sym pCathegory) – зберігає у файл всі об’єкти зі стеку за вказаною категорією. Шлях до файлу може бути як абсолютний так умовний («.\<ім’я файлу>»). У цьому випадку пошук ведеться в директорії де розташовано dll інтерпретатора.

_CompactObjStack() – стискає стек, видаляє з нього записи, що відмічені як видалені, тобто ті, для яких поле Keep == 0.

_RemoveObjects(sym pCathegory) – видаляє зі стеку всі об’єкти із вказаною категорією.

_ClearObjStack() – очищує стек об’єктів.

_Run(sym pVarName) – виконує код, що міститься у змінній з ім’ям pVarName. Відповідає послідовному виклику директив **#make** та **#run**. Використовується коли код для виконання генерується під час трансляції скрипту, наприклад, якщо інша функція (_LoadFileToVar() або ReadIn() тощо) повертає текст, який потрібно транслювати. Якщо код, який потрібно транслювати, вказується безпосередньо у тілі скрипту, слід використовувати директиви #make та #run. Код виконується в рамках поточного контексту.

_GetParamAsNum(sym pParam) – повертає значення параметру pParam зі стека параметрів перетворене в число.

_GetParamAsStr(sym pParam) – повертає значення параметру pParam зі стека параметрів перетворене в строку.

_SetParam(sym pParam, sym pVal) – додає в стек параметр з ім’ям pParam та значенням pVal. Якщо параметр з таким ім’ям уже присутній у стеку, його значення буде перезаписане.

_LoadFileToVar(sym pFile, sym pTarget) – завантажує вміст текстового файлу у змінну з ім’ям pTarget. Шлях до файлу може бути як абсолютний так умовний («.\<ім’я файлу>»). У цьому випадку пошук ведеться в директорії де розташовано dll інтерпретатора.

_SaveVarToFile(sym pTarget, sym pFile) – зберігає значення змінної у вказаний файл. Шлях до файлу може бути як абсолютний так умовний («.\<ім’я файлу>»). У цьому випадку пошук ведеться в директорії де розташовано dll інтерпретатора. Якщо шлях до файлу не існує, буде створений новий файл.

_SaveFragmentToFile(sym pTarget, sym pFile) – зберігає відкладений фрагмент з ідентифікатором, що міститься у змінній pTarget, у вказаний файл. Шлях до файлу може бути як абсолютний так умовний («.\<ім’я файлу>»). У цьому випадку пошук ведеться в директорії де розташовано dll інтерпретатора. Якщо шлях до файлу не існує, буде створений новий файл.

Важливо: функція _SaveFragmentToFile() зберігає фрагмент «як є», не розгортаючи фрагменти, ідентифікатори яких містяться у тексті.

_GetConfig(sym pFile, sym pLine) – завантажує значення параметру pLine з файлу pFile. Шлях до файлу може бути як абсолютний так умовний («.\<ім’я файлу>»). У цьому випадку пошук ведеться в директорії де розташовано dll інтерпретатора. Файл повинен мати структуру <параметр>=<значення>.

_SaveState() – зберігає поточний стан усіх стеків ELI в файл state.log у робочій директорії ELI.

_SaveVarStack(num pLevel) – зберігає стан стека змінних у файл varstack.log у робочій директорії ELI. Якщо аргумент pLevel дорівнює 0 зберігається вміст глобального стека, якщо 1 – локального (тобто того, що використовується у даний час).

_WriteOut(sym pStr) – направляє у стандартний потік виводу вміст аргументу pStr. Крім звичайного набору допустимих строкових значень, функції можна передати шість зарезервованих слів:

#varstack – вивід стека змінних. Використовується поточний активний стек.
#funcstack – вивід стека функцій.
#prmstack – вивід стека параметрів.
#objstack – вивід стека об’єктів.
#clstack – вивід стека класів.
#procstack – вивід стека процедур.
#frgstack – вивід стека відкладених фрагментів кода.
#**endl** – вивід символу кінця строки (перенос каретки).

_ReadIn(sym pVar) – зчитує символи зі стандартного потоку вводу та записує їх в змінну pVar. Максимальна кількість символів для вводу: 4096.

_System(sym pCmd) – виконує команду Windows, що міститься в аргументі pCmd. Аналог C++ функції system().

_LastError() – повертає опис останньої зафіксованої помилки.

_ConnectLib(sym pPath) – підключає до інтерпретатора зовнішню бібліотеку dll. Повертає дескриптор підключеної бібліотеки. У разі невдачі повертає -1. Шлях до файлу може бути як абсолютний так і умовний («.\<ім'я файлу>»). У цьому випадку пошук ведеться в директорії де розташовано dll інтерпретатора.

_FreeLib(num pHandle) – вивільнює підключену зовнішню бібліотеку.

ImportFunc(num pHandle, sym pExtName, sym pInName, sym pArgList) – імпортує із підключеної бібліотеки, що має дескриптор pHandle, функцію-обгортку з ім'ям pExtName та додає її у стек інтерпретатора під ім'ям pInName та списком аргументів pArgList. Аргументами pInName и pArgList повинні бути константні строки, оскільки в pInName використовується символ маркера функції «», а pArgList містить пробіли, які будуть проігноровані без використання константної строки.

_DebugIntoFile() – вмикає логування дій інтерпретатора, направляє вивід у файл у робочій директорії ELI.

_DebugIntoScreen() – вмикає логування дій інтерпретатора, направляє вивід в стандартний поток виводу.

_StopDebug() – вимикає логування дій інтерпретатора.

_sleep(num pMsec) – призупиняє трансляцію на pMsec мілісекунд. Аналог функції Sleep() у C++.

Аргументи функцій представлені тими типами даних мови ELI :
num – числовий.
sym – строковий.

Тип параметра вказує лише на механізм інтерпретації значення, яке буде передано функції в тілі скрипту, самі параметри зберігаються у стеку без вказання типу. У самих функціях-обгортках використовуються методи класу ELI, на кшталт GetParamToInt() для того, щоб отримати зі стека параметр з потрібним типом.

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

Аргументами функції можуть бути: змінні, константи, скалярні вирази, властивості та методи об'єктів, а також інші функції.

Процедури

Процедури не повертають значень, зате їх реалізація повністю залежить від користувача. Крім того, на відміну від функцій, алгоритм яких жорстко обмежений їх описом у коді бібліотеки ELI (або в коді користувацької аплікації), процедури можуть бути створені,

використані та видалені під час виконання скрипту. Процедура, що оголошена в рамках одного контексту (наприклад, в тілі іншої процедури), може бути використана у будь-якому іншому, до тих пір поки не буде завершена робота бібліотеки ELI. Ім’я процедури – унікальне та не повинне збігатись з ім’ям функції зі стеку. Усі змінні, що оголошені у тілі процедури, видимі тільки в ній і будуть знищені після того, як процедура завершить роботу. В тілі процедури не можна використовувати змінні, що оголошені не в її тілі, тому що вони розміщені в іншому стеку змінних.

Оголошення процедури:

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

Звернення:

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

Видалення зі стеку:

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

У процедури може бути довільна кількість аргументів. Аргументи ініціалізуються у локальному стеку змінних процедури як змінні типу sym. При здійснені виклику процедури, значення, що передані їй в якості аргументів, будуть записані у відповідні змінні. Передача аргументів здійснюється по значенню. В якості аргументів процедури можна використовувати: константи, змінні, скалярні вирази, властивості об’єктів та результати функцій чи методів.

Користувацькі процедури зберігаються в спеціальному стеку. При оголошенні процедури, у ньому створюються два елементи з категорією «procedure» та ім’ям, що було вказане при оголошенні. Один елемент містить список аргументів, другий – посилання-ідентифікатор на тіло процедури, що збережене у стеку фрагментів коду.

Об’єкти

У мові ELI «об’єкт» є абстракцією, що уособлює певну множину записів зі спеціального стеку. Для інтерпретатора кожен елемент цього стеку – це набір типизованих полів, що описані у наступній структурі:

```
//-----------------------------------------------------------------------------------------------------------------
struct RESOURCE
{
  UINT Index;                       //унікальний індекс
  std::wstring ObjectCathegory;  //категорія об'єкта-власника
  std::wstring ObjectID;            //ID об'єкта-власника
  std::wstring PropertyID;          //ID властивості
  std::wstring Value;                //значення
  std::wstring KeepInStack;       //вказує чи зберігати ресурс в стеку після компакта
  std::wstring SaveInFile;         //вказує чи зберігати ресурс в файл на диску
};
//-----------------------------------------------------------------------------------------------------------------
```

Стек об'єктів описаний як std::vector<RESOURCE> та включений у спеціальний клас RESOURCESTACK. Таким чином, для інтерпретатора об'єктом є набір елементів відповідного стеку, для яких поле ObjectID є однаковим. Поле PropertyID визначає властивість об'єкту, а поле Value – значення цієї властивості. Ключовою властивістю ELI при роботі з об'єктами є можливість додавати та видаляти властивості прямо «на ходу». В протилежність, набір методів строго обмежений. Кожний об'єкт має наступні методи:

Create(sym cathegory, sym ctor_args) – створює новий об'єкт із вказаною категорією. Аргумент ctor_args містить перелік аргументів для конструктора класу.

Exist() – перевіряє чи присутній у стеку такий об'єкт, повертає 1, якщо існує хоча б один елемент стеку об'єктів, для якого ObjectID відповідає імені об'єкта, що викликав даний метод.

Have(sym prop_name) – перевіряє чи існує вказана властивість у обєкта.

Add(sym prop_name, sym val) – додає властивість до об'єкта.
Хоча аргумент val методу Add() оголошений як sym, його можна використовувати аби передати числове значення. ELI сам виконає перетворення типів даних.

Remove(sym prop_name) – видаляє властивість з м'ям prop_name.

Destroy() – знищує об'єкт.

Keep(sym prop_name, sym istrue) – вказує чи треба зберігати властивість у стеку. Аргумент istrue може приймати значення 1 чи 0.

Save(sym prop_name, sym istrue) – вказує чи треба зберігати властивість у файл. Аргумент istrue може приймати значення 1 чи 0.

Execute(sym prop_name) – виконує відкладений фрагмент коду, ідентифікатор якого міститься у властивості prop_name. Ідентифікатор повинен бути спочатку створений за допомогою директиви #make. Під час виклику цього методу створюється тимчасова процедура з одним аргументом $this, у якому міститься ім'я об'єкта.

Важливо: хоча метод **Execute** виконує таку ж функцію, що і директива **#run** та функція **_Run()**, між ними є різниця. Директива #run та функція _Run() працюють в рамках поточного контексту, в той час як метод Execute створює новий контекст, тож код, що розгортається під час виклику метода, виконується в рамках цього, нового контексту.

Show() – виводить на екран всі властивості об’єкта.

GetName() – повертає ім'я об’єкта.

ExportIn(sym pPropNames, sym pPropVals) – експортує властивості об’єкта у два об’єкта-переліка із вказаними іменами. Кожна властивість об’єкту-переліку pPropNames буде мати числовий індекс, а значенням цієї властивості буде ім’я властивості об’єкта, що викликав метод. У об’єкта pPropVals значення властивостей будуть містити значення властивостей об’єкта, що викликав метод. Крім того останньою властивістю обох списків буде Count, що містить кількість індексних елементів списку.

Наприклад:

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

Результатом виконання &x.Show() буде:

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

Результатом виконання &y.Show() буде:

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

Загальний синтаксис при роботі з об’єктами наступний:

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

Перший рядок створює об’єкт з іменем &Hero та категорією NPC, другий додає об’єкту числову властивість Strength зі значенням 100, третій – знищує об’єкт. У створеного об’єкта за замовчуванням вже є дві властивості: Owner зі значенням «<none>» та ObjectName зі значенням, що відповідає імені об'єкта з рядка скрипту («Hero» у даному випадку).

Звертатись до властивостей об’єктів можна за допомогою службового оператора «.», так само, як це відбувається у багатьох мовах програмування при роботі з полями структур. Встановлення значення властивості відбувається за допомогою оператора «=».

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

Ім’я об’єкту є унікальним, чутливим до регістру і може включати в себе тільки літери, цифри та символ «_». Ім’я об’єкту може визначатись значенням змінної (чи навіть декількох) – така конструкція називається псевдонімом.

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

$part1 = Bohdan;
$part2 = Hmelnitsky;
&$part1$part2.Create(NPC, '');
//-------------------------------------------------------------------------------------------------------------------
```

Процедурам можна передавати ім’я об’єкту в якості параметра. Для цього потрібно у виклику процедури опустити в аргументі спецсимвол &, а в тілі процедури використати конструкцію &<ім’я аргументу процедури> в якості псевдоніма об’єкту.

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

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

Псевдонім властивості, так само як псевдонім об’єкту, може бути складений зі значень змінних. Проте варто бути уважним при використанні числових змінних у псевдонімах властивостей чи об’єктів. Справа в тому, що значення змінних, явно визначених як числа, зберігаються у стеку разом із десятковим знаком, тож саме у такому вигляді вони будуть вставлені у псевдонім.

Також слід пам’ятати, що при встановлені значення властивості ELI спочатку намагатиметься привести праву частину виразу до числового типу. Якщо це неможливо, тоді буде використаний строковий тип.

Об’єкти класів

Вище були описані принципи роботи з простими об’єктами, набір властивостей яких встановлюється користувачем індивідуально для кожного екземпляра. Крім того, набір методів для простих об’єктів строго обмежений стандартними методами інтерпретатора. По суті ці об’єкти можна назвати структурами. Однак ELI підтримує також можливість створення типізованих об’єктів по визначеному шаблону. Такий шаблон називається класом.

Набір властивостей класу визначається користувачем при оголошенні. В подальшому, при створенні об’єкту даного класу, всі властивості, вказані при оголошенні, будуть додані до екземпляру об’єкта автоматично. Крім того клас дозволяє визначати не тільки властивості, але й визначені користувачем методи, що вигідно виділяє об’єкт класу поміж простих об’єктів.

Приклад простого класу, що визначає 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;}
}
//-------------------------------------------------------------------------------------------------------------------
```

Тепер після використання стандартного методу Create() з аргументом List буде створений об’єкт, що включає в себе властивість Next зі значенням 0.000 та чотири методи. Після кожного виклику методу AddItem() до властивостей об’єкта буде додана ще одна із вказаним значенням. Звісно, до об’єктів класів можна застосовувати всі вбудовані методи, що доступні простим об’єктам. Ім’я класу чутливе до регістру, тому при виконанні &obj_1.Create(List, '') та &obj_2.Create(list, '') буде створено два різних об’єкта. Ім’я класу є унікальним, тому неможливо оголосити два класи з однаковими іменами, але можна переоголосити клас, попередньо видаливши його директивою

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

або додати чи видалити властивості та методи

```
//-------------------------------------------------------------------------------------------------------------------
#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;
  }
}
//-------------------------------------------------------------------------------------------------------------------
```

Важливо: усім властивостям, під час опису класу, слід задати значення за замовчуванням.

Важливо: зміни, що були зроблені за допомогою директиви #modify class, впливають тільки на об'єкти класу, створені після використання директиви. Об'єкти, що були створені раніше, залишаються без змін.

Важливо: якщо метод класу приймає участь у виразі, що підлягає обчисленню, то він повинен повертати певний результат за допомогою директиви #return. В протилежному випадку інтерпретатор не зможе коректно сформувати вираз.

Конструктор класу

Часто необхідно, аби властивості об'єкта класу були визначені одразу після його створення. Для цього, при оголошенні класу, можна визначити метод, ім'я якого збігається з ім'ям класу, саме його інтерпретатор і вважатиме за конструктор. У тілі конструктора можна визначати та виконувати дії над будь-якими членами класу, але повертати результат – не можна. Конструктор оголошується так само, як і будь-який метод класу. Список аргументів конструктора передається у другому аргументі вбудованого методу Create().

Важливо: конструктор обов'язково повинен оголошуватись як публічний, в іншому випадку метод Create() не зможе викликати його.

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

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

Якщо аргументів декілька, їх потрібно передавати через строкову константу.

```
//-------------------------------------------------------------------------------------------------------------------
#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 класу

ELI розглядає всі об’єкти як набір даних, що міститься у одиниці трансляції і зазвичай так воно і є, тому для знищення об’єкту достатньо видалити всі його поля зі стеку об’єктів за допомогою вбудованого методу Destroy(). Проте, деякі з таких об’єктів можуть містити вказівники або дескриптори, що посилаються на адреси у купі (heap). Наприклад, якщо метод об’єкту інкапсулює функцію зі сторонньої dll або хост-аплікації, у якій створюється об’єкт за допомогою оператора new. Звісно хорошим тоном для розробника бібліотеки (хост-аплікації) буде створення механізму очищення пам’яті таких об’єктів у купі, але цей механізм ще треба запустити з контексту ELI. Для цього до функціоналу класів було додано деструктор.

Деструктор оголошується так само, як і будь-який метод класу. Аргументів не має. Не спадкується. Деструктор буде автоматично викликаний під час виклику методу Destroy().

Важливо: деструктор обов'язково повинен оголошуватись як публічний, в іншому випадку метод Destroy() не зможе викликати його.

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

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

Приватні та публічні члени класу

Класи можуть мати як приватні (закриті) так і публічні (відкриті) методи та властивості. За замовчуванням, директиви #property або #method оголошують член класу як приватний (private). Щоб оголосити публічний член, необхідно скористатись директивою #public property чи #public method.

Крім того, ELI підтримує один з принципів ООП: інкапсуляцію. Це означає, що приватні члени класу доступні лише у його методах.

Успадкування класів

Класи можуть успадковувати властивості та методи інших класів. Численне наслідування не підтримується. Успадковуються лише публічні члени. Конструктор також успадковується, проте його ім'я успадковується від батьківського класу і не є конструктором за замовчуванням для нащадка. Тим не менш успадкований конструктор може викликатись як будь-який інший метод. Наймолодший в ієрархії клас-нащадок успадковує конструктори всіх її членів.

Клас-нащадок може визначити свій член з ім'ям, відповідним до імені успадкованого члена, при цьому перевага надається члену, який було визначено, а не успадковано.

Усі успадковані члени за замовчуванням оголошуються як публічні.

```
//-------------------------------------------------------------------------------------------------------------
#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;}
}
//-------------------------------------------------------------------------------------------------------------
```

У цьому прикладі клас 3DPoint не має свого конструктора, проте успадковує конструктор від класу Point, так само, як і усі інші члени. Таким чином, для того, щоб повністю визначити точку у трьох координатах, об'єкт класу 3DPoint може або викликати успадкований конструктор Point(), або використати успадковані методи SetX() и SetY(), після чого викликати свій приватний метод SetZ().

Вкладеність об'єктів

Об'єкти класів можуть використовувати у якості властивостей інші об'єкти класів. Для цього під час опису класу потрібно оголосити властивість, використовуючи наступну конструкцію:

```
//-------------------------------------------------------------------------------------------------------------
```

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

Де A це ім'я конструктора класу, а в дужках вказано список аргументів конструктора. Якщо клас не має конструктора (або його виклик з якихось причин не потрібен), слід використовувати таку конструкцію:

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

При кожному створенні об'єкта класу B, інтерпретатор створить об'єкт класу A із автоматично згенерованим ім'ям та надасть властивості Prop1 ім'я цього об'єкта. Після цього можна звертатись до членів вкладеного об'єкта класу A через властивість Prop1. Наприклад:

```
//-----------------------------------------------------------------------------------------------------------------
#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; //помилка – властивість Pos є приватною;
&point.Y.Set(20); //правильно;
//-----------------------------------------------------------------------------------------------------------------
```

Автоматично згенерований об'єкт відрізняється від користувацького тим, що його службова властивість Owner містить ім'я об'єкта-власника. При знищенні об'єкта-власника, усі зв'язані з ним вкладені об'єкти також будуть знищені.

Існує ще один спосіб створення властивості-об'єкта: використання вбудованого методу Add() та передачі йому у якості другого аргумента конструкції типу #class Class(<аргументи конструктора>). Додана методом Add() властивість буде публічною.

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

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

Важливо: через певні властивості парсера, для передачі декількох аргументів конструктора у даному випадку слід використовувати строкову константу.

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

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

Розширення функціоналу за допомогою зовнішніх бібліотек

Звісно, ELI не в змозі надати весь спектр можливостей, які спадуть (чи можуть спасти) на думку користувачу. Саме на цей випадок у ньому передбачена можливість розширення функціоналу за допомогою підключення зовнішніх бібліотек та використання їхнього коду в своїх скриптах. Конкретніше мова йде про використання функцій-обгорток, описаних у зовнішніх бібліотеках, так само, як це передбачено для користувацької аплікації. Достатньо загорнути код в функцію-обгортку при проектуванні бібліотеки, та імпортувати її в ELI під час роботи скрипту, після чого ця функція буде доступна для виклику аж до закінчення роботи інтерпретатора, або до вивільнення зовнішньої бібліотеки. Звісно, для того, щоб бібліотека була сумісна з ELI, при її проектуванні необхідно включити у проект заголовковий файл **eli_interface.h**, що містить віртуальний інтерфейс.

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

Цей код під’єднає до інтерпретатору бібліотеку testextdll.dll з директорії, у якій розташовано бібліотеку ELI, та передасть змінній $hinst її дескриптор. Припустимо, що у бібліотеці є функція

```
//---------------------------------------------------------------------------------------------------------------------
__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);
}
//---------------------------------------------------------------------------------------------------------------------
```

яка сумує два числа з десятковою крапкою та повертає результат. Аби надати інтерпретатору можливість викликати цю функцію, необхідно включити у текст скрипту наступний код:

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

Тут **$hinst** – це дескриптор зовнішньої бібліотеки, **Summ** – ім’я функції у зовнішній бібліотеці, **_summ** – ім’я, яке буде використовувати інтерпретатор, а рядок **'num pNum1,num pNum2'** визначає список параметрів функції. Звісно ж імена параметрів повинні збігатись з тими, які використовуються в тілі функції, що імпортується. У випадку невдачі імпорту, функція _ImportFunc() повертає значення, що менше 1, в іншому випадку повертає 1.

Важливо: метод GetCurrentFuncName() в тілі функції із зовнішньої бібліотеки потрібен аби дізнатись у інтерпретатора ім’я функції, яке він призначив функції з зовнішньої бібліотеки.

Важливо: внутрішнє ім’я функції, що використовує ELI, може бути яким завгодно, та необхідно слідкувати, щоб воно не збігалось з іменем функції, що вже існує у стеку.

Важливо: ім’я функції, що експортується з DLL, може змінюватись в залежності від того, як компілятор декорує імена при використанні специфікатора __stdcall. Наприклад, компілятор Embarcadero C++ Builder експортує функцію void __stdcall Foo(void*) у вигляді «Foo», а MinGW у вигляді «Foo@4». Якщо функція інтерпретатора _ImportFunc() повідомляє про невдачу, відкоригуйте ім’я експортної функції у місці виклику _ImportFunc().

Тепер можна використати імпортовану функцію у скрипті.

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

Після чого можна звільнити бібліотеку, якщо вона більше не потрібна.

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

Важливо: усі функції, що були імпортовані в інтерпретатор, будуть видалені зі стеку після виклику _FreeLib().

Всі зовнішні бібліотеки, підключені до інтерпретатора, будуть автоматично звільнені при відключенні бібліотеки ELI.

Декорації

Декорацією у мові ELI називається мітка, що декорує собою певний набір конструкцій мови, від змінних до цілих виразів. В певному сенсі декорація працює за принципом директиви препроцессора #define в C++. Під час трансляції рядка, декорації, що містяться в ньому, будуть оброблені перед усіма наступними діями та замість них в рядок будуть підставлені задекоровані конструкції.

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

Декорація ?ref замінює собою конструкцію, що складається зі змінної $x. В подальшому, у всіх рядках, що транслюються, замість ?ref буде підставлена змінна $x. У результаті трансляції таких рядків:

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

змінна $y дорівнюватиме 15. Інтерпретатор обробить праву частину виразу, підставить замість декорації приховану конструкцію, після чого обчислить значення виразу.

Декорації зручно використовувати для збереження шаблонних конструкцій обчислення, наприклад, якщо значення змінної обчислюється за складною формулою.

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

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

Така форма запису дозволяє значно скоротити код, зробити його більш читабельним. Слід, однак, пам'ятати, що інтерпретатор не слідкує за коректністю задекорованої конструкції. Якщо у наведеному вище прикладі змінні $a чи $b не були попередньо визначені, інтерпретатор видасть помилку трансляції.

Оскільки декорації містять імена, а не значення змінних, їх зручно використовувати в тих випадках, коли потрібно працювати із самою змінною, а не її значенням, наприклад у випадку функції _free(). Крім того декорації дозволяють передавати набір аргументів, тому така конструкція є законною:

```
//-------------------------------------------------------------------------------------------------------------------
$a = здоровенькі;
$b = ' ';
$c = були!;

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

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

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

В результаті на екран буде видано «здоровенькі були!».

Декорація також може використовуватися для створення псевдонімів об'єктів або властивостей, однак таку декорацію не можна передати процедурі. Дозволяється декорувати навіть імена методів та функцій, проте використовувати такі декорації можна тільки у правій частині виразу. Інтерпретатор дозволяє застосовувати ім'я декорації і лівій частині виразу лише в одному випадку: під час опису декорації.

Така конструкція допустима:

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

$a = ?f(10);

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

&cl.Create(A, '');

?f = &cl.Foo();
```

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

А така – ні:

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

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

Крім того, слід пам'ятати, що декорації можуть приховувати лише завершені конструкції та не підтримують декорування фрагментів коду. Наприклад, не можна задекорувати опис класу або оголошення змінної. Декорування процедур також не має сенсу, тому що процедури не використовуються у правій частині виразу.