Scriptengine „Purgá“

Dokumentation

Version 0.47

www.virtual-maxim.de/purga

Inhaltsverzeichnis

Einleitung

Purga (rus: Schneesturm, Blizzard) ist eine Scriptengine mit einfacher Schnittstelle zu C++-Programmen.

Merkmale & Möglichkeiten:

- die Sprache ist case-sensitive, d.h. es wird zwischen Groß- und Kleinschreibung unterschieden.
- Anweisungen werden mit Semikolon abgeschlossen!
- 5 Variablentypen: int, float, string, bool, array

- Variablen aus Purga in C++ exportieren
- Einzeilige und mehrzeilige Kommentare
- Inkrementieren(var++;) und Dekrementieren(var--;)
- Scripts als String übergeben
- C++ Funktionen(stdcall, cdecl) in Purga aufrufen
- C++ Variablen in Purga verwenden
- Purga Variablen nach C++ exportieren
- Standardinitialisierung von Variablen
- if (elseif, else), while, for
- Cache-Methode. Scripts können vor dem Ausführen „vorkompiliert“ werden.
- Rechnen mit Variablen + - * / %
- Logische Operatoren: and or
- Vergleichsoperatoren: < <= > >= == !=
- Vordefinierte Konstanten
- Deklaration beinhalten immer eine Initialisierung (Standardinitialisierung)
- dynamische Arrays

To Do – Liste

- Rekursion
- Überladen von Funktionen
- Bessere Fehlererkennung
- C++ Klassen in Purga verwenden
- Purga-Scripts in anderen Scripts einbinden (include in C++)
- Logische Negation – Operator
- continue – Schlüsselwörter
- foreach – Schleife für Arrays
- mehrdimensionale Arrays
- Escape-Sequenzen für Unicode-Zeichen, binäre und hexadezimale Zahlen
- Referenzen
- Code in Module aufteilen
- Absturzstabilität erhöhen
- arrays als Rückgabetyp von Funktionen und als Parameter
- lambda-Terme

Sprachelemente

Kommentare

Einzeilige Kommentare werden mit zwei Schrägstrichen // eigeleitet.
Beispiel: // Das ist ein einzeiliger Kommentar

Mehrzeilige Kommentare werden durch einen Schrägstrich mit einem Stern eingeleitet /* und durch die gleichen Zeichen in ungedrehter Reihenfolge geschlossen */.
Beispiel:

```
/* Das ist ein
 Kommentar über
 mehrere Zeilen */
```

Variablen

Variablen müssen vor der Verwendung deklariert werden. Eine Initialisierung ist nicht nötig, da Variablen immer mit Standardwerten initisiert werden.
Es werden 4 Datentypen unterstützt (siehe Tabelle).

Dateitypen:

Datentyp	*Speicher*	*Wertebereich*
int	4Byte	–2,147,483,648 bis 2,147,483,647
float	8Byte (double in c++)	1.7E +/- 308 (15 Nachkommastellen)
bool	1Byte	true / false bzw. 1/0
string	2Byte pro Zeichen	Unicode-Zeichen

Variablen deklarieren & initialisieren:

Datentyp Variablenname;
Datentyp Variablenname = Initialisierungswert;

Datentyp Variablenname, Variablenname = Wert, Variablenname;

Standardinitialisierung:

Wird bei der Deklaration kein Wert angegeben, so ist int = 0, float = 0.0 und string = “”.

Variablen definieren:

Variablenname = Wert;

Variablennamen:

Variablennamen beginnen mit einem Buchstaben und können aus Buchstaben, Zahlen und dem Unterstrich bestehen.

Inkrementieren und Dekrementieren:

Variable++;
Variable--;

Beide Operatoren müssen hinter der Variable stehen (postfix). Eine prefix-Variante, wie etwa in C++ gibt es nicht.

--Variable; // Fehler
++Variable; // Fehler

Zudem müssen die Variablen immer alleine stehen. Eine Benutzung innerhalb einen mathematischen Terms ist nicht erlaubt.

Rechnen mit Variablen:

Es werden 4 elementare Rechenarten unterstützt: + - / *.
Dabei ist zu beachten, dass es auch Stringoperatoren gibt und mache Operation nicht definirt sind und zur einen Fehlermeldung führen z.B. String / String. Um Fehler zu vermeiden, gibt es eine Tabelle in Kapitel Operatoren, welche alle Möglichkeiten aufzeigt.

Konstanten

Deklaration & Initialisierung:

Konstanten werden mit Schlüsselwort const eingeleitet und müssen bei der Deklaration gleich Initialisiert werden.

const Datentyp Konstantenname;

Referenzen [Planung]

Deklaration & Initialisierung:

Die Deklaration erfolgt genauso wie bei Variablen mit Schlüsselwort *ref* vor dem Datentypen. Eine Referenz kann nur deklariert werden, wenn sie auch gleich Initialisiert wird.

ref Datentyp Referenzname = Variable;

Die Variable auf die die Referenz zeigt, darf keine Konstante oder andere Referenz sein.

Eine Referenz kann konstant sein, d.h. man hat nur Lesezugriff auf die Daten. Dies wird durch das Schlüsselwort *const* realisiert.

const ref Referenzname = Variable;

Arrays [Planung / in Arbeit]

Arrays sind dynamische (assoziative) Felder. Am ehesten sind Arrays mit std::map aus STL zu vergleichen. Ein Array kann Daten verschiedener Datentypen verwalten.

Deklaration:

Deklaration wird durch das Schlüsselwort *array* eingeleitet.

array name;

Bei dieser Deklaration wird ein leeres Array erstellt.

Initialisierung:

Name = { Wert, String, ...};

Diese Initialisierungsform kann zusammen mit Deklaration erfolgen.

array name = { Wert, String, ...};

Zugriff:

Wie in C++ per Index:

Name[0] = 4;
Name[1] = „hallo“;

Die Indexzahl darf nur Positiv oder Null sein.
Ist das Element beim Schreiben noch nicht vorhanden, so wird es automatisch angelegt.

Ist das Element beim Lesen nicht vorhanden, so wird eine Fehlermeldung ausgegeben und das Ausführen wird unterbrochen.

Escape-Sequenzen

Espace-Sequenzen werden wie in C/C++ einfach in einen String eingeführt.

`\n`	Zeilenumbruch (von englisch *new line*)
`\t`	Horizontaler Tabulator (von englisch *(horizontal) tabulator*)
\b	Rückschritt (von englisch *backspace*)
\f	Seitenvorschub (von englisch *formfeed*)
\r	Wagenrücklauf (von englisch *(carriage) return*)

Operatoren:

Arithmetische Operationen:

Rechnen mit Variablen: Ergebnisse bei verschiedenen Datentypen				
Variable 1	**Operator**	**Variable 2**	**Ergebnis**	**Auto-Datatype-Casting**
int	+	int	int	
		float	float	
		bool	FEHLER	
		string	string	
	-	int	int	
		float	float	
		bool	FEHLER	
		string	FEHLER	
	*	int	int	
		float	float	
		bool	FEHLER	
		string	string	String in int-facher Kopie
	/	int	int	
		float	int	nur ganzer Anteil als Ergebnis
		bool	FEHLER	
		string	FEHLER	

float	+	int	float	
		float	float	
		bool	FEHLER	
		string	string	
	-	int	float	
		float	float	
		bool	FEHLER	
		string	FEHLER	
	*	int	float	
		float	float	
		bool	FEHLER	
		string	FEHLER	
	/	int	float	
		float	float	
		bool	FEHLER	
		string	FEHLER	
bool	+ - * / - Operationen sind nicht erlaubt.			
string	+	int	string	
		float	string	
		bool	string	
		string	string	
	-	int	FEHLER	
		float	FEHLER	
		bool	FEHLER	
		string	FEHLER	
	*	int	string	String in int-facher Kopie
		float	FEHLER	
		bool	FEHLER	
		string	FEHLER	
	/	int	FEHLER	
		float	FEHLER	
		bool	FEHLER	
		string	FEHLER	

Vergleichsoperatoren:

< kleiner
<= kleiner gleich
> größer
>= größer gleich
== Gleichheit
!= Ungleichheit

Beim Vergleichen mit Strings, außer auf Gleichheit, wird immer die Länge des Strings als Integer genommen.

Logische Operatoren:

and – Logisches UND
or – Logisches ODER

Vorrangregeln (Präzedenzregeln):

Diese Regeln gelten bei der Auswertung eines mathematischen Ausdruckes oder einer Bedingung (z.B. für if).

Wie in der Mathematik:
- Geklammerte Ausdrücke zuerst
- Ungeklammerte Ausdrücke gemäß vier Präzedenzklassen:
  1) ! (NOT)
  2) Multiplikationsoperatoren
  3) Additionsoperatoren
  4) Vergleichsoperatoren
- bei gleicher Präzedenz erfolgt Abarbeitung von links nach rechts

Vorrang-Tabelle			
Rang	Operator in Purga	Name	Beispiel
1	( ) []	Klammern Indizierung	(a + b) array[2]
2	@	Funktionsaufruf	@summe(1,2)
3	-	Minusvorzeichen	-4
4	* /	Multiplikation Division	a*b a/b
5	+ -	Addition Subtraktion	1 + b a - b
6	< <= > >=	Vergleichsoperatoren	a > 5
7	==, equal !=	Gleichheitsoperatoren	a == b
8	and	Logisches AND	a > 3 and b
9	or	Logisches OR	a or b
10	=	Zuweisungsoperator	a = b

Ein Wertevergleich erfolgt durch zwei Gleichheitszeichen (==) oder durch das Schlüsselwort equal (die Wahl ist frei).

Ein Vergleich zwischen Wahrheitswerten und Zahlen ist verboten.

IF (ELSEIF, ELSE)

```
if( Bedingung )
{
  Anweisung;
}
elseif( Bedingung)
{
 Anweisung;
}
else
{
Anweisung;
}
```

elseif und else-Teile sind optional.

Die geschweiften Klammern müssen immer angegeben werden.

WHILE

```
while( Bedingung)
{
 Anweisung;
}
```

FOR

```
for(Initialisierung; Bedingung; Anweisung)
{
Anweisung;
}
```

Variablen, die in dem Kopf der Schleife deklariert wurden, sind nur innerhalb der Schleife verfügbar.

FOR EACH [Planung]

```
foreach( [array name] as [iterator name] )
{
```

}

Wird verwendet um ein Array zu durchlaufenen.
Es werden nur wirklich belegte Elemente ausgegeben. Wenn man also ein Array mit Werten arr[0] = 3 und arr[8] = 5 hat so ergibt die Ausgabe nur 3 und 5.

break

break erlaubt eine eine while/for Schleife zu verlassen. Dabei wird nur die nächsthöhere Schleife beendet.

Funktionen [in Arbeit]

function [<datatype>] functionname ([[*const*] [*ref*] <datatype> argument], …)
{

    [*return* value;]
}

Die Funktion wird durch das Schlüsselwort *function* eingeleitet. Danach kommt der Datentyp des Rückgabewertes der Funktion. Der Rückgabetyp wird nur dann angegeben, wenn die Funktion auch einen Wert zurück gibt.
Die Parameterliste ist optional. Die Werte werden entweder als Kopie oder als Referenz übergeben. Die Referenz wird durch das Schlüsselwort *ref* vor dem Datentyp des Parameters gekennzeichnet. Eine Referenz kann durch das Schlüsselwort *const* konstant gemacht werden. Das Schlüsselwort *const* kann nur im Zusammenhang mit einer Referenz verwendet werden.
Wenn eine Funktion einen Wert zurück gibt, dann wird dies durch die *return* – Anweisung realisiert.

Beispiele:

```
function hallo()
{
  printf(“hallo“);
}

function int sum(int a, int b)
{
```

```
  return a+b;
}

function sum(int a, int b, ref int res)
{
 res = a+b;
}
```

Schnittstelle: Purga -> C++

Variablen von Purga nach C++ exportieren:

Um auf globale Script-Variablen in C++ zuzugreifen gibt es überladene Methoden:

```
// Eine globale Purga-Variable in C++ exportieren
bool GetVariable(const wchar_t* var_name, int& variable);
bool GetVariable(const wchar_t* var_name, double& variable);
bool GetVariable(const wchar_t* var_name, std::wstring& variable);
bool GetVariable(const wchar_t* var_name, bool& variable);
```

Purga:

```
int purga_zahl = 0;
```

C++:

```
int cpp_zahl = 0;
purga.GetVariable(L“purga_zahl“, cpp_zahl);
```

Globale Variablen bleiben nach dem Ausführen des Scripts bestehen und können in anderen Scripten verwendet werden.

Schnittstelle: C++ -> Purga

Variablen von C++ nach Purga exportieren:

Ab Version 0.40 DEAKTIVIERT!!! Aktivierung evtl. in einer späten Version.

Man kann ganz einfach C++-Variablen in Purga-Scripten verwenden. Dazu muss die C++-Variable registriert werden.

```
// Variablen von C++ nach Purga-Script exportieren
// Diese können dann in Scripts ganz normal verwendet werden
```

```
bool RegisterVariable(const wchar_t* var_name, int &var);
bool RegisterVariable(const wchar_t* var_name, double &var);
bool RegisterVariable(const wchar_t* var_name, std::wstring &var);
bool RegisterVariable(const wchar_t* var_name, bool &var);
```

Wenn man den Wert der Variable in Purga-Script verändert, so wird er automatisch auch in C++ verändert.
Die registrierten Variablen sind in Purga als globale Variablen deklariert.

C++ Funktionen in Purga aufrufen (TEST!):

Zuerst muss die Funktion registriert werden.

```
bool Purga::RegisterFunction(const wchar_t* FullNameOfFunction, void* FunctionP);
```

Dabei ist FullNameOfFunction der Funktionskopf als String. Bei den Parametern müssen nur die Datentypen angegeben werden. FunctionP ist der Zeiger auf die Funktion.

Ein Beispiel:

C++:

```
int rechne(int z1, int z2)
{
	int ergebnis = z1 + z2;
	std::cout<<"Summe von z1 und z2 betraegt: "<< ergebnis <<std::endl;
	return ergebnis;
}

pg.RegisterFunction(L"int rechne(int,int)", (void*)&rechne);
```

Purga:

```
int res = rechne(5,8);
print("res =" + res);
```

Ist der Parameter eine Referenz oder ein Zeiger, so stellt man nach dem Datentyp ein &- oder ein * - Zeichen dahinter.

Ein Beispiel:

C++:

```
void tausche(int &a, int &b)
{
        int t = a;
        a = b;
        b = t;
}
pg.RegisterFunction(L"void tausche(int*,int&)", (void*)&tausche);
```

Purga:

```
int a = 3;
int b = 5;
print("a =" + a + " b=" + b + "\n");
tausche(a,b);
print("a =" + a + " b=" + b + "\n");
```

Hinweis: es werden nur Datentypen int, double und (const)wchar_t* unterstützt(BEIM EXPORTIEREN NUR string, float oder int angeben).

Rückgabewerte gibt es nur bei Funktionen, die einen int-, double-, bool-Wert oder einen gültigen Zeiger auf wchar_t-Array zurückgeben.

Es sind keine Leerzeichen zwischen den einzelnen Werten beim Funktionsaufruf zugelassen.

DLL-Datein laden / Purga-Bibliotheken erstellen [in Planug / Test]

Um eine Bibliothek zu erstellen, die weitergegeben werden kann, muss ein Script erstellt werden, der die benötigten Funktionen definiert oder diese aus einer DLL einbindet. Das letztere geschieht mit den Befehlen: LoadLibrary, ImportLibraryFunction, FreeLibrary und erstmal nur unter Windows.

Cache-Methode

Um Ausführungsgeschwindigkeit eines Scriptes zu steigern sollte man den Script vorkompilieren. Dazu gibt es CacheScript-Methode. Der Code wird in Bytecode übersetzt und für spätere Ausführungen in einer Liste abgespeichert.

Wenn man ein Script ausführt ohne, dass dieser sich im Cache befindet, so wird er beim ersten Ausführen automatisch in Cache abgelegt, d.h. folgende Aufrufe des gleichen Scripts erfolgen schneller.

Interne Funktionen

Interne Funktionen kann man ohne weites in allen Scripts verwenen.

Mathematische Funktionen

float cos(float rad) – Berechnet Kosinus eines Winkels in Radiant.

float sin(float rad) – Berechnet Sinus eines Winkels in Radiant.

float tan(float rad) – Berechnet Tangens eines Winkels in Radiant.

float rad_to_grad(float rad) – Rechnet Radiant in Grad um.

float grad_to_rad(float grad) – Rechnet Grad in Radiant um.

Sonstige

void print(string str) – Gibt einen String in dem Konsolenfenster aus. Hab man eine Zahl so kann man sie davor in einen String umwandeln, in dem man sie mit einem leeren String addiert.
Beispiel:

```
int v = 4;
print(""+v);
```

int ftoi(float) – Konventiert einen float- in einen int-Wert. Wird benötigt weil eine automatische Konvertierung nicht vorgesehen ist.

Vordefinierte Konstanten

Purga besitzt mehrere vordefinierte Konstanten. Diese sind global deklariert – können also überall im Code verwendet werden.

Integer-Konstanten:

MAX_INT = 2147483647
MIN_INT = -2147483648

PURGA_VERSION_NUMBER = Aktuelle Purga-Version

Fließkomma-Konstanten:

PI = 3.14159265358979323846
PI_INV = 0.31830988618379067154

String-Konstanten:

PURGA_VERSION = Aktuelle Purga Version

Beschränkungen

Um Geschwindigkeit der Engine nicht unnötig zu belasten, müssen an manchen Stellen einige Beschränkungen in Kauf genommen werden.

- Die maximale Anzahl an Argumenten für C++-Funktionen beträgt 20
- Eine Codezeile im Script darf höchstens 512 Zeichen lang sein.
- Variablennamen dürfen maximal 64 Zeichen lang sein.
- Fließkomma-Genauigkeit beträgt 10 Nachkommastellen

Schlüsselwörter

int, float, string, void, array, vector, false, true, if, while, else, elseif, string, for, bool, and, or, const, function, return, ref, equal, include, class, public, private, lambda