# Brückencode-Chronik oder ein C++/CLI-Erfahrungsbericht

Ahoi Landratte! Ich bin Käpt’n Blackbyte – definitiv kein normaler Software-Entwickler, der nur gucken will, ob er so einen Blog-Eintrag auf einem professionellen Internet-Hafen veröffentlichen kann.

Ich erzähle dir von meiner Zeit auf den stürmischen Gewässern des CLI/C++-Meers. Das liegt zwischen dem sicheren Hafen von C#, wo die GUIs blinken und man sich keine Sorgen um verschollene Goldstücke (Memory Leaks) machen muss, und der wilden See von C++, auf der legendäre Schiffe wie OpenSSL, VTK oder GLM schon seit Jahrzehnten kreuzen.

Das hier wird kein Seemannsgarn, sondern eine echte Chronik darüber, wie man zwischen beiden Welten navigiert, auch wenn die Wellen mal bis zum Mast reichen. Also spitz die Lauscher, Plankenlatscher, und lerne aus meinen Fehlern

## Binnenschifffahrt mit Platform Invocation Services

Wenn du Kanalfrosch nur ein paar Golddukaten transportieren willst, dann halt dich an die Piratenweisheit „Ein Kuss auf die Segel hält den Wind günstig!“ (oder auch KISS – Keep it simple stupid) dann nutze P/Invoke (Platform Invocation Services). Die C++ function wird direkt als unmanged Code aufgerufen, aber das funktioniert nur für simple Datentypen, ist dann aber leichter als Behälter voller Tee über Bord zu werfen.

using System.Runtime.InteropServices;

class Program{

[DllImport("mylib.dll", CallingConvention = CallingConvention.Cdecl)]

public static extern int AddNumbers(int a, int b);

static void Main() {

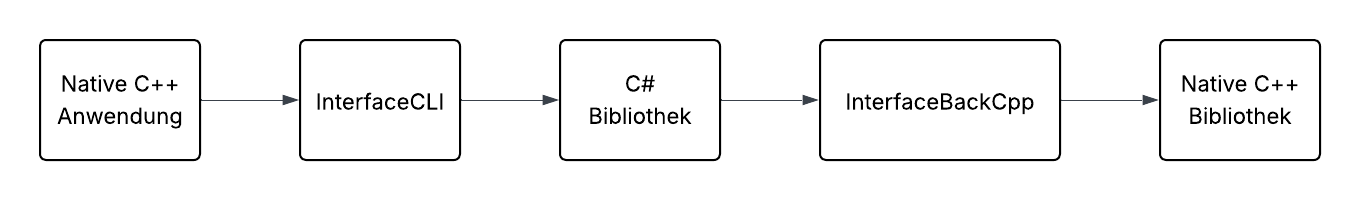
int result = AddNumbers(5, 7);

Console.WriteLine($"Ergebnis: {result}");

}}

### Handel mit komplexen zusammengesetzten Schätzen

Okay Möwen-Sitzplatz, was ist aber nun, wenn du eine Schatzkiste hast, in der verschiedener wertvoller Krempel ist und du die gesamte Kiste gleichzeitig transportieren willst, statt alle Objekte separat? Nun dafür gibt es CLI/C++ ein Slang von Microsoft, der C++ erweitert und genau die Brücke zwischen ausführbaren C++-Code und .Net-Code für dich bereitstellt. (Damit funktioniert es auch mit F# oder VB, aber ein alter Pirat hält seine Geschichten einfach also rede ich nur von C#). Also soll unsere Handelsroute wie folgt aussehen (hier ist alles aufgezeichnet <https://github.com/jan-kasper-t2i/CLIExperienceBlogExampleCode>):



Jeder Halt ist ein eigenes Projekt, wobei InterfaceCLI und InterfaceBackCpp, den Microsoft C++ Compiler mit dem Schalter /clr nutzen, womit wir in diesen die CLI-Syntax verwenden können.

In unsere nativen C++ Anwendung, können wir komplexe Daten erzeugen (ironischerweise sind die Daten als Beispiel wieder simpel gehalten).

ValuableItem::ValuableItem(std::wstring name, int value){\_name = name; \_value = value;}  
std::wstring ValuableItem::GetName() { return \_name;}  
int ValuableItem::GetValue() { return \_value; }  
  
Treasure::Treasure(ValuableItem& item1, ValuableItem& item2,ValuableItem& item3)  
{  
 contents.push\_back(item1);  
 contents.push\_back(item2);  
 contents.push\_back(item3);  
}  
std::vector<ValuableItem>& Treasure::GetContents(){return contents;}

Wir erstellen ein paar Beispieldaten und senden sie an das InterfaceCLI. Da wir von Native C++ aufrufen, muss der verwendete Header C++ sein und keine CLI-Bestandteile enthalten. Was es aber enthält ist die Möglichkeit die Klasse in der Bibliothek zu finden ( Und nein Plankenschrubber ich werde dir nicht erklären, wie declspec das genau bewerkstelligt, ich will mich auf meine Geschichte konzentrieren!). Und ein Objekt welche die Zugriffe auf die C# Funktionalitäten erlaubt.

class CSharpInterface;  
class \_\_declspec(dllexport) CppCliWrapper  
{  
public:  
 virtual void SendTreasure(Treasure& treasure);  
private:  
 CSharpInterface\* \_managedState;  
};

Und nun nimm beide Augenklappen ab und schau dir an, wie wir CLI-Syntax benutzen.

Über Marshalingkönnen wir unmanged native C++ Datentypen, wie wstring in managed Datentypen wie String konvertieren. Mit gcnew erschaffen wir sogenannte Handles (Mit ^ deklarier), die auf dem Heap lokalisiert sind und von der Garbage Collection verwaltet werden. Der Weg zurück von C# zu C++ ist ähnlich, nur das man den Speicher der erschaffenen C++ Objekte natürlich selbst verwalten muss. Und wo wir grade bei Speicherverwaltung sind, mit dem C# unsafe Feature können wir auch Pointer durchreichen und einiges an Kopieren einsparen. Großartig bei großen Datenmengen, oder auch bei kleinen wenn man unbedingt mal wieder Pointer-Arithmetik nutzen und schwere zu wartenden Code schreiben will.

void CppCliWrapper::SendTreasure(Treasure& treasure)  
{  
 ITreasure^ managedTreasure = TreasureConverter::ToITreasure(treasure);  
 \_managedState->CSharpLib->ReceiveTreasure(managedTreasure);  
}

ITreasure^ TreasureConverter::ToITreasure(Treasure& treasure)  
{  
 std::vector<ValuableItem>& items = treasure.GetContents();  
 String^ item1Name = msclr::interop::marshal\_as<String^>(items[0].GetName());  
 int item1Value = items[0].GetValue();  
 String^ item2Name = msclr::interop::marshal\_as<String^>(items[1].GetName());  
 int item2Value = items[1].GetValue();  
 String^ item3Name = msclr::interop::marshal\_as<String^>(items[2].GetName());  
 int item3Value = items[2].GetValue();  
  
 ManagedValuableItem^ item1 = gcnew ManagedValuableItem(item1Name, item1Value);  
 ManagedValuableItem^ item2 = gcnew ManagedValuableItem(item2Name, item2Value);  
 ManagedValuableItem^ item3 = gcnew ManagedValuableItem(item3Name, item3Value);  
   
 Collections::Generic::List<IValuableItem^>^ managedValuableItemList = gcnew Collections::Generic::List<IValuableItem^>();  
 managedValuableItemList->Add(item1);  
 managedValuableItemList->Add(item2);  
 managedValuableItemList->Add(item3);  
 ManagedTreasure^ managedTreasure = gcnew ManagedTreasure(managedValuableItemList);  
   
 return managedTreasure;  
}

### Die Weisheiten eines alten Seebären

Und nun die wirklichen Perlen in den Muscheln meines langen Geschwätzes, eine Auflistung an Erkenntnissen, die erst beim Umsetzen kamen und nicht schon beim Lesen von Dokumentation (Und da gibt es besser verständlichere und vollständigere als dieses Genörgel eines alten Seemannes!)

* Microsoft weiß, was sie tun. Oder alle anderen wissen es nicht bei ihrem selbstentwickelten CLI/C++-Dialekt. Rider funktioniert generell, aber nach Änderungen an den Projekt-Dateien gibt es manchmal obskure Fehler, welche Visual Studio problemfrei auflöst.
* Achte genau auf die unterstützen Marshalling, std::**w**string ist problemfrei, std::string überhaupt nicht.
* Beim Interface von C# zu C++, sei sparsam mit C# feature. Primär-Konstruktoren oder records können zu Problemen führen.
* Eine strikte Trennung zwischen CLI/C++-Header und normalen C++ Headern ist nötig! Native C++ Projekte können ausschließlich **komplett** native C++ Header einziehen, die Implementierung kann dann CLI sein. Kein Header-Only-Ansatz möglich.
* Nutze das CLI-Projekt als Adapter-Schicht, logikfrei
* Gute Idee: 2 Konstruktoren, einer nimmt managed Datentypen, einer unmanged Datentypen entgegen.
* Überlege dir ein Namensschema, frühzeitig. Häufig existiert der Datentype einmal für C++, einmal für C# und einmal als CLI-Adapter.
* Nicht alles an interface sollte 1:1 übersetzt werden. In C# ist Vector3 eine gute Lösung, in C++ übersetzt sich das in 3 integer, statt einen eigenen Datentyp zu erschaffen.
* Verwirrende Compiler-Fehler nach Schnittstellen-Erstellung? Überprüfe, ob alle ^ vorhanden sind, das ist mir häufiger passiert als ein fehlendes Semikolon, als ich angefangen habe zu programmieren.
* Die marshalling Header gehören **GANZ** nach oben! Keine Ausnahme!
* Fehlerbehandlung! Ein ErrorCallback wird irgendwann benötigt.

Mein ehrliches subjektives Fazit zu CLI/C++ ist: Die Syntax ist gut leserlich aber schlecht schreibbar und es gibt immer wieder kleiner Stolpersteine in Funktionalität als auch Tooling. Meinem Verständnis hat es sehr geholfen CLI als eigene Programmiersprache zu begreifen, statt C++ mit Zusatzfunktionen. Das Kopieren von Daten hat sich in unserem Fall (<https://t2informatik.de/en/software-development/references/dr-sennewald-medizintechnik-gmbh>) als keinerlei Performance Problem herausgestellt. Aber am Ende löst die Sprache genau das Problem, wofür sie erschaffen worden ist. Sie erlaubt eine schmale und stabile Schnittstelle zwischen C++ und C# zu erstellen. Und nun segelt Käpt’n Blackbyte wieder los und erkundet andere neue Technologien.