Gert Jan Timmerman
is trainer/consultant bij Info Support.
gertt@infosupport.com, www.infosupport.com

Managed extensions for C++

.NET-APPLICATIES SCHRIJVEN IN C++

Eén van de belangrijkste eigenschappen van Microsoft .NET is dat het programmeertaal-onafhankelijk is. Dit betekent dat we programma's voor .NET in verschillende talen kunnen schrijven, zoals C#, Visual Basic.NET en JScript. Ook C++ is één van de talen waarin .NET-programma's geschreven kunnen worden. Toch zijn er belangrijke verschillen tussen het schrijven van .NET-programmatuur in C++ en in de andere .NET-talen. In dit artikel wordt allereerst aangegeven hoe we in C++ .NET-applicaties kunnen schrijven, daarna wordt ingegaan op bovengenoemde verschillen. Ten slotte worden de voor- en nadelen van het gebruik van 'Managed extensions for C++' afgewogen.

Wanneer we een traditioneel C++-programma compileren, genereert de compiler native code, ook wel unmanaged code genoemd. Dit houdt in dat de gegenereerde executable instructies bevat die direct door de processor kunnen worden uitgevoerd. Als we bij het compileren daarentegen een vlag meegeven (/clr), genereert de compiler een .NET-assembly, die IL (Intermediate Language) instructies bevat. Deze IL-instructies worden niet direct door de processor uitgevoerd, maar worden runtime Just-In-Time (JIT) gecompileerd tot native code. Deze assembly met IL-instructies noemen we managed code, omdat de Common Language Runtime (CLR) gedurende de executie van dit programma aan de touwtjes trekt. We kunnen dus een bestaand C++-programma (zonder dat we daar enige wijziging in hoeven aan te brengen) compileren tot een .NET-assembly. Dit houdt nog niet in dat de in zo'n programma gedeclareerde classes kunnen worden uitgewisseld met .NET-applicaties die in andere talen zijn geschreven. Daarvoor hebben we niet alleen managed code nodig, maar ook managed data. Voor C++-programma's die geen gebruik maken van het .NET Framework zijn er dus twee mogelijkheden: compileren tot managed code of tot unmanaged code. C++-programma's die wel gebruik maken van het .NET Framework kunnen alleen naar managed code gecompileerd worden.

Ansi C++

C++ is sinds enige jaren een ANSI-standaard. Dit houdt in dat de syntax en semantiek van C++ vastliggen en dat hierin niet zomaar wijzigingen kunnen worden aangebracht. Bij de meeste bestaande talen die Microsoft geschikt heeft gemaakt voor .NET, zoals Visual Basic en JScript, heeft men aanzienlijke aanpassingen moeten doen, maar bij C++ was dit niet zomaar mogelijk. Vandaar dat Microsoft, conform de ANSI-standaard, alleen uitbreidingen aan de taal heeft toegevoegd; uitbreidingen die herkenbaar zijn aan de keywords die beginnen met een dubbele _ (underscore). Hierdoor kunnen bestaande C++-programma's zonder problemen worden gecompileerd met de compiler die onderdeel is van het .NET Framework. Er is dus geen sprake van een nieuwe taal C++.NET, maar van een verzameling uitbreidingen op de standaard, genaamd Managed extensions for C++.

Namespaces en datatypes

In codevoorbeeld 1 is een C++-programma te zien dat gebruikt maakt van het .NET Framework om 'Hello World' in een console-window te schrijven. Wat allereerst opvalt is de #using directive, die nodig is om type-informatie uit een .NET-assembly (in dit geval mscorlib.dll) aan de compiler bekend te maken. De rest van de code van dit programma is gewone C++-code, omdat de .NET namespaces en classes volgens de bestaande C++-syntax te gebruiken zijn.

Omdat de .NET-datatypes naast de gewone C++-datatypes kunnen worden gebruikt, is daar een mapping tussen gemaakt. Het opvallendst bij die mapping is het C++-type char dat overeenkomt met het .NET-type SByte en niet met het .NET-type Char. De achterliggende reden is dat een Char in .NET een Unicode-character is (2 bytes dus) en in C++ een gewoon character van

één byte. Een ander opvallend geval is het C++-datatype long dat overeenkomt met het .NET-type Int32 en niet met Int64. Ten slotte de strings. In C++ kennen we twee soorten string literals: de string die bestaat uit ASCII-characters en die wordt weergegeven tussen dubbele quotes ("gewone string literal") en de string die bestaat uit Unicode-characters en die wordt voorafgegaan door een L (L"Unicode string literal"). De Managed Extensions voegen hier een derde string literal aan toe, namelijk de .NET String (een object van het type System::String). Deze wordt voorafgegaan door een S (S"een .NET String literal"). In codevoorbeeld 2 is hetzelfde programma nog eens opgenomen, maar nu met gebruik van een .NET String. Het voordeel hiervan is dat er runtime geen conversie hoeft te worden gedaan, maar dat de compiler onmiddellijk een string literal van het juiste type als parameter aan de WriteLine()-functie kan meegeven.

Managed data

Als we zelf een class definiëren in C++ kunnen we, dankzij de Managed Extensions, zelf kiezen of we willen dat dit een traditionele class is, waarbij we zelf verantwoordelijk zijn voor het opruimen van eventuele geheugenruimte. We kunnen ook kiezen voor een garbage-collected class, waarbij de CLR voor de geheugenruimte verantwoordelijk is. Wanneer we het keyword __gc gebruiken, geven we aan dat objecten van deze class alleen op de managed heap kunnen worden gecreëerd, en dus niet op de gewone (unmanaged) heap, maar ook niet op de stack (als lokale variabele) of in het globale datasegment (als globale variabele). We kunnen dus alleen via managed pointers (pointers naar de managed heap) met deze objecten werken. In codevoorbeeld 3 zien we hoe zo'n class kan worden gedeclareerd en gebruikt.

Ook al is de programmeur niet verantwoordelijk voor het opruimen van de geheugenruimte van een __gc class, toch is het wel mogelijk om delete te gebruiken op een pointer naar een object van een __gc class. Dit resulteert niet in het opruimen van geheugenruimte, maar zorgt dat de destructor wordt aangeroepen. Het aanroepen van een destructor kan in C++ dus op een door de programmeur bepaald moment gebeuren. Classes die gedeclareerd zijn met __gc kunnen typische .NET-features gebruiken, zoals: static constructors, delegates, events, properties en attributen, maar ze mogen geen gebruik maken van een aantal standaard C++-kenmerken, zoals default arguments, friend functies en classes, const member-functies en C++-operator-functies.

```
#using <mscorlib.dll>

using namespace System;

int main()
{
   Console::WriteLine("Hello World!");
   return 0;
}
```

Code voorbeeld 1

```
#using <mscorlib.dll>

using namespace System;

int main()
{
   Console::WriteLine(S"Hello World!");
   return 0;
}
```

Code voorbeeld 2

```
#using <mscorlib.dll>

using namespace System;

public __gc class MyClass
{
    public:
        int Number;
        String *Greeting;
};

int main()
{
   MyClass *pMyClass = new MyClass;
   pMyClass->Number = 5;
   pMyClass->Greeting = S"Hello Wordl";

   Console::WriteLine(pMyClass->Greeting);

   return 0;
}
```

Code voorbeeld 3

Managed pointers en references

Doordat objecten van een __gc class uitsluitend op de managed heap kunnen worden gecreëerd, wijzen pointers en references die naar deze objecten verwijzen, ook naar delen van deze managed heap. Deze pointers en references noemen we dan ook managed pointers en references. Omdat de Garbage Collector (GC) verantwoordelijk is

voor het opruimen van de managed heap en daarbij ook objecten kan verplaatsen, moeten we bij het gebruik van managed pointers en references goed oppassen. Bij het verplaatsen van objecten wijzigen uiteraard ook de pointers en references. Daarom is het werken met managed pointers en references aan een aantal restricties onderhevig. Zo mag op managed pointers geen pointer-aritmetiek worden uitgevoerd, omdat de GC anders de pointer niet zou kunnen wijzigen als het object wordt verplaatst. Een managed pointer mag ook niet zomaar aan een unmanaged pointer worden toegekend, omdat de unmanaged pointer niet onder het beheer van de GC valt. Als het object zou worden verplaatst, zou de unmanaged pointer niet aangepast worden. Om een managed pointer toch (als unmanaged pointer) door te kunnen geven aan een unmanaged functie, bestaat de mogelijkheid een pointer te pinnen. Dat wil zeggen dat gedurende een beperkte tijd het object niet kan worden verplaatst door de GC. Hiervoor gebruiken we het keyword __pin. In codevoorbeeld 4 zien we een voorbeeld van het gebruik van een pinning pointer; zolang deze pinning pointer naar het object wijst, kan het object niet worden verplaatst.

Value Types

Naast __gc classes kunnen we in C++ ook __value classes declareren. Een object van een __value class kan gedeclareerd worden als lokale variabele (op de stack), als globale variabele (in het globale datasegment), op de unmanaged heap of als member van een __gc type (op de managed heap). Bij een toekenning tussen variabelen van een __value type wordt een kopie gemaakt van de inhoud van de variabele. Bij __gc classes is het niet mogelijk op deze manier een kopie te maken van een object, omdat bij objecten van een __gc class alleen met pointers of met references kan worden gewerkt, niet met de objecten zelf. De geheugenruimte van een object van een __value type wordt niet beheerd door de GC, dus deze objecten worden niet door de GC verplaatst in het geheugen. Daarom kan het adres van een object van een __value type in een gewone (__nogc) pointer gestopt worden. Als een object van een __value type moet worden doorgegeven aan een functie die een object op de managed heap verwacht, moet het object eerst worden geboxt (__box()). Hierdoor wordt een soort wrapper op de managed heap gecreëerd waarin een kopie van het object gemaakt wordt. In codevoorbeeld 5 zien we hoe een object van een __value type wordt geboxt en vervolgens ge-unboxt. Bij het unboxen wordt een pointer verkregen naar de kopie van het __value object dat op de managed heap is gemaakt. Met behulp van deze pointer kan vervolgens de waarde weer worden gekopieerd naar een gewone variabele van het __value type.

```
#include <stdio.h>

#using <mscorlib.dll>

public __gc class MyClass
{
    public:
        int SomeMember;

    // some other members

};

int main()
{
   MyClass *pMyClass = new MyClass;

   MyClass __pin * pinMyClass;

   pinMyClass = pMyClass;
   scanf("%d",(int *)&pinMyClass->SomeMember);
   printf("Het ingevoerde getal is: %d", pMyClass->SomeMember);

   return 0;
}
```

Code voorbeeld 4

Interactie met unmanaged code

Wanneer we .NET-applicaties schrijven, zullen we zo nu en dan toch nog tegen de noodzaak aanlopen om functies aan te roepen uit traditionele DLL's (unmanaged code). Dit kan zich bijvoorbeeld voordoen als we een Win32-functie direct willen aanroepen omdat de betreffende functionaliteit niet in het .NET Framework aanwezig is. Ook kan het gebeuren dat we een bestaande DLL hebben (van een third-party) die bepaalde noodzakelijke functionaliteit aanbiedt. In C++ hebben we hiervoor twee mogelijkheden: 'Platform Invoke' en 'It Just Works' (IJW). Gebruikers van andere .NET-talen (Visual Basic.NET, C# enzovoort) hebben hierin geen keuze. Zij kunnen alleen Platform Invoke gebruiken. Platform Invoke is de algemene manier in .NET om unmanaged code aan te roepen. Hiervoor wordt het attribuut DllImportAttribute gebruikt. Met behulp van dit attribuut kan de programmeur aangeven welke functie uit welke DLL hij wil aanroepen. De CLR zorgt vervolgens voor het marshalen van de parameters en de returnwaarde. De programmeur declareert de functie met formele parameters van .NET-types en roept vervolgens de functie aan met actuele parameters van deze types. De CLR converteert deze parameters - indien nodig - naar de datatypes die de aangeroepen functie verwacht, en pint de variabelen eventueel in het geheugen, zodat de GC ze niet kan verplaatsen terwijl de unmanaged functie wordt uitgevoerd. Nadat de functie is uitgevoerd,

worden de returnwaarde en eventuele outputparameters op dezelfde manier weer geconverteerd naar .NET-datatypes. Een voorbeeld van het gebruik van Platform Invoke is te zien in codevoorbeeld 6.

Uitsluitend in C++ is er nog een tweede mogelijkheid om unmanaged code aan te roepen, namelijk IJW. Met behulp van IJW kunnen we niet alleen functies in een DLL aanroepen, maar ook in een static library. IJW maakt gebruik van het feit dat de prototypes van functies in static library's en in DLL's meestal in C of C++ gedefinieerd zijn. Doordat er in C++ een mapping bestaat tussen native C++-datatypes en .NET-datatypes, kan de programmeur dus zelf zorgen dat de meegegeven parameters van het juiste datatype zijn. Dit betekent dat de CLR niet hoeft te converteren, waardoor de aanroep van de unmanaged functie efficiënter wordt. Bij de aanroep van één enkele functie is dit verschil niet erg belangrijk, omdat de conversies die de CLR niet hoeft te doen toch op een of andere manier door de programmeur zelf gedaan moeten worden. De parameters moeten immers in het juiste formaat worden aangeboden. Wanneer dezelfde functie een groot aantal malen wordt aangeroepen of een groep functies die dezelfde parameters gebruikt, kan IJW een stuk efficiënter zijn. De programmeur hoeft in dit geval de conversie maar één keer te doen en kan daarna de betreffende functies een groot aantal malen aanroepen, zonder dat er bij elke functie-call opnieuw geconverteerd hoeft te worden. In dit geval zou het gebruik van Platform Invoke ertoe leiden dat er wel bij elke call opnieuw door de CLR geconverteerd moet worden. Een voorbeeld van het gebruik van IJW is te zien in codevoorbeeld 7.

Voor- en nadelen van het gebruik van C++

Wanneer we de keuze moeten maken of we onze .NET-applicaties in C++ of bijvoorbeeld in C# gaan schrijven, is het natuurlijk van belang een goede afweging te maken tussen voor- en nadelen. Allereerst de nadelen van het gebruik van C++. Omdat C++ een bestaande gestandaardiseerde taal is, heeft Microsoft toevoegingen aan de taal moeten doen om de taal geschikt te maken voor .NET. Het nadeel hiervan is dat er voor bijna alles twee mogelijkheden zijn: we kunnen gewone (__nogc) classes definiëren die zich gedragen zoals we dat gewend zijn van C++-classes, met alle bestaande mogelijkheden. We kunnen ook __gc classes definiëren, .NET-classes die totaal andere features hebben dan normale classes. Eigenlijk hebben we dus te maken met twee talen binnen één taal: traditioneel C++ en de Managed Extensions. Mede hierdoor moeten veel dingen die in andere talen impliciet gaan (boxen, pinnen enzovoort) in C++ expliciet aangegeven worden, waardoor het programma minder leesbaar wordt. Ook het subtiele onderscheid tussen __gc en __nogc pointers en references maakt het programmeren met de Managed Extensions een stuk ingewikkelder.

```
#using <mscorlib.dll>

using namespace System;

int main()
{
   int i = 5;
   ValueType *boxedValue = __box(i);

   Console::WriteLine(S"De waarde van i is: {0}", boxedValue);
   int __box *pInside = dynamic_cast<int __box *>(boxedValue);

   if(pInside)
      i = *pInside;

   return 0;
}
```

Code voorbeeld 5

```
#using <mscorlib.dll>

using namespace System;
using namespace System::Runtime::InteropServices;

[DllImport("User32.dll")]
int MessageBox(IntPtr hWnd, String *Message, String *Title,
unsigned int type);

int main()
{
   MessageBox(IntPtr(0),S"Hello World",S"Greeting", 0);
   return 0;
}
```

Code voorbeeld 6

```
#using <mscorlib.dll>
#include <windows.h>

using namespace System;

int main()
{
   MessageBox(NULL,"Hello World","Greeting", MB_OK);
   return 0;
}
```

Code voorbeeld 7

Het voordeel van het gebruik van C++ zit vooral in de compatibiliteit met bestaande C++-programmatuur. Wanneer we een aantal bestaande C++-applicaties hebben die we ook in .NET willen kunnen gebruiken, dan hoeven we niet ineens de hele applicatie te herschrijven. We kunnen de bestaande applicatie compileren tot IL-code en vervolgens voor nieuwe delen van de applicatie of stukken die moeten worden herschreven, de Managed Extensions gebruiken. Zo kunnen we een bestaande applicatie geleidelijk migreren naar .NET. Een bijkomend voordeel is dat we bestaande C++-kennis kunnen blijven gebruiken, hoewel de programmeurs natuurlijk wel de Managed Extensions zullen moeten leren. Verder kan performancewinst een reden zijn om voor C++ te kiezen. Met name als er veel interactie met unmanaged code nodig is, kan het gebruik van IJW belangrijke voordelen bieden.

Gefaseerde migratie

Omdat C++ een bestaande gestandaardiseerde taal is, is de taal in tegenstelling tot veel andere .NET-talen, niet aangepast voor het gebruik binnen .NET, maar alleen uitgebreid. Dankzij deze uitbreidingen kan men in C++ zowel traditionele applicaties schrijven als .NET-applicaties, maar deze mogelijkheden kunnen ook in één applicatie worden gecombineerd. Dit maakt C++ met name geschikt om bestaande applicaties langzamerhand te migreren naar .NET. De belangrijkste nadelen hierbij zijn dat de toch al ingewikkelde taal nog wat complexer geworden is en, mede door een groot aantal nieuwe keywords, ook wat minder leesbaar.

nuttige internetadressen:

www.microsoft.com/net/
msdn.microsoft.com/visualc/
www.gotdotnet.com/team/cplusplus/
http://www.microsoft.com/netherlands/vstudio

(vervolg van pagina 57)

We zijn overtuigd dat proprietary software en Open Source software lange tijd samen kunnen bestaan, en we verwachten niet dat de ene kant de andere zal tegenwerken. Ik heb het idee dat DotGNU-mensen een religieuzere en radicalere positie innemen. Ze zijn vooral druk doende Microsoft en de concurrentie te vernietigen. Voor ons is Mono een middel om een goede technologie te ontwikkelen voor Unix/Linux en ons eigen ontwikkelplatform te verbeteren. Onze licentie staat hen toe om code van onze codebase te gebruiken, maar we kunnen veranderingen van hen niet terug 'mergen', omdat dat ons zou dwingen de GPL te gebruiken voor onze class-libraries; dus de samenwerking is bijna nihil."

Welke delen van Mono zijn al 'GPL-ed' en welke niet? Wat is de reden van de verschillende licenties van de verschillende delen?

Miguel de Icaza: "De compiler is GPL, de class-libraries zijn MIT X11 en de runtime is LGPL. Eenvoudigweg betekent dit dat je Mono zonder enige restrictie kunt gebruiken om alle soorten applicaties te bouwen. Sommigen hebben de runtime gelicentieerd zodat ze proprietary verbeteringen kunnen aanbrengen, of ze hebben de compiler gelicentieerd zodat ze delen kunnen hergebruiken in een proprietary product. Wat mij betreft is dat geen probleem. De class-libraries hebben nauwelijks licentievoorwaarden, zodat er ook geen licentieproblemen kunnen ontstaan. De zorg die sommigen hadden met betrekking tot de librarie van GPL, GPL met excepties en LGPL is dat het een samenwerkende inspanning betreft, zodat elke schrijver een ietwat andere interpretatie kan hebben van de licentie, en hoe deze zich verhoudt tot code die op een objectgeoriënteerde manier is 'afgeleid' van code die met deze voorwaarden is gelicentieerd. Om problemen met de interpretatie te vermijden willen wij glashelder zijn. Het bouwen van proprietary applicaties met Mono is prima, wij kozen voor de X11 MIT licentie."

Microsoft beweegt zich naar een volledig 'gemanaged' operating systeem. Longhorn is de volgende grote stap in dit proces. Ziet u een gelijkwaardig proces in de Open Source Community? Een 'gemanagede' Linux of BSD met Mono in het centrum?

Miguel De Icaza: "Microsoft heeft een voordeel heeft dat Linux niet bezit. Microsoft kan een koers uitzetten, zodat iedereen een dezelfde kant opgaat. Ik ben ervan overtuigd dat Mono voor de toekomst van Linux de juiste manier is om applicaties te ontwikkelen, maar de enige manier om door ontwikkelaars te worden geadopteerd is wanneer ze gelijkgestemd zijn. Mono zal zich dus moeten bewijzen voordat het alom wordt geadopteerd. Het is voor de open source community ook lastiger om in korte tijd een architectuur te ontwerpen voor heel grote projecten. Zaken gaan in slow-motion. Zodra een doel is gezet, gaan dingen wel vooruit, maar dit proces vergt vele iteraties. Ik denk dat Novell/Ximian in staat zullen zijn de voordelen van deze technologie te tonen, maar het zal langer duren dan het Microsoft kostte om in een volledig 'gemanagede' wereld te stappen."

Wat zijn de grote 'gaten' in het Mono 'ecosysteem' die u graag zou willen opvullen (docs, gebruiksvriendelijke IDE, enzovoort), en waarbij onze lezers u zouden kunnen ondersteunen?

Miguel de Icaza: "Mono uitproberen en bug-reports sturen is op dit ogenblik de belangrijkste taak. We hebben al een boel software getest, maar nog niet alles. Dus het is voor developers een prima tijd om hun eigen software te testen en alle problemen die ze mochten tegenkomen aan ons te rapporteren. We hebben documentatie nodig voor zowel de .NET class-libraries als onze eigen class-libraries (Gtk, Gnome, Mozilla bindings). En ten slotte, het nieuwe .NET Framework biedt het meeste plezier aan hen die betrokken willen raken bij Mono. Het nieuwe framework heeft vele nieuwe classes en functionaliteit geïntroduceerd, en het is erg leuk deze zelf te implementeren."

Nuttige internetadressen

http://www.go-mono.org/
http://www.ecma-international.org/