Gert Jan Timmerman is werkzaam als trainer/consultant bij Info Support.

Visual C++ en .NET

DE KEUZE VOOR EEN LEESBAARDER TAAL

Toen Microsoft in 2002 de eerste versie van .NET uitbracht, was een onderdeel daarvan de Managed Extensions for C++. Omdat er een ANSI-standaard voor C++ bestond, koos Microsoft ervoor zich hieraan te houden. Dat hield in dat men geen wijzigingen kon aanbrengen in de bestaande taalelementen en dat nieuwe keywords die aan de taal werden toegevoegd, moesten beginnen met een dubbele underscore (bijvoorbeeld __gc). Het resultaat hiervan was een taal die eigenlijk twee talen in zich verenigde: ANSI-C++ en C++ voor .NET.

Deze taal had dus ongeveer alles dubbel: twee soorten classes, twee soorten pointers, twee soorten arrays, enzovoort. Het grootste nadeel hiervan was dat programma's die geschreven waren met behulp van de Managed Extensions for C++ bijna onleesbaar waren. Aan de ene kant was het al gauw een mengelmoes van ANSI- en .NET-elementen, aan de andere kant wemelde het van allerlei keywords die met een dubbele underscore begonnen en die effecten teweegbrachten die in de meeste andere .NET-talen impliciet zijn (bijvoorbeeld boxen). Al met al was het een lofwaardig streven van Microsoft om zich netjes aan de ANSI-standaard te houden, maar het resultaat was daarmee bijna onwerkbaar geworden.
Daarom heeft Microsoft in Visual Studio 2005 een nieuwe poging gewaagd om C++ geschikt te maken voor .NET. Men heeft de ANSI-standaard laten varen en de vrijheid genomen om bestaande keywords een nieuwe betekenis te geven - afhankelijk van de context waarin dat keyword gebruikt wordt - en om nieuwe keywords toe te voegen die niet met een dubbele underscore beginnen; dit om de leesbaarheid te verbeteren. Ook heeft men ervoor gekozen verschijnselen die in andere .NET-talen impliciet zijn ook in C++ impliciet te laten gebeuren. Daarmee lijkt C++ in Visual Studio 2005 veel meer op een 'gewone' .NET-taal en is daarmee bruikbaarder geworden. Dit artikel (inclusief de codevoorbeelden) is gebaseerd op de Community Technology Preview van mei 2004.

Managed code en native code

Een ander nadeel van de Managed Extensions for C++ was dat hiermee alleen assemblies konden worden gemaakt die naast MSIL ook native code bevatten en die, mede daardoor, niet verifieerbaar waren (unsafe). Dit ontstaat doordat functies in een C++-programma dat met de /clr-vlag is gecompileerd twee entry-points krijgen: een managed entry-point (MSIL) en een unmanaged entry-point (native code). Wanneer het adres van een functie in een variabele wordt opgeslagen, wordt het adres van het unmanaged entry-point gebruikt, omdat van tevoren niet is vast te stellen vanuit welke code dit adres zal worden gebruikt om de functie aan te roepen. Dat leidt ertoe dat er veel managed/unmanaged overgangen zijn in een C++-programma dat met de /clr-vlag is gecompileerd.
Een oplossing voor dit probleem is om een C++-programma te compileren tot pure MSIL, dus zonder unmanaged code. Dit houdt in dat er dus ook geen native libraries kunnen worden gebruikt, maar uitsluitend MSIL-assemblies. Visual C++ 2005 heeft hiervoor een compiler-optie, namelijk /clr:pure. Wanneer deze optie wordt meegegeven, wordt er alleen MSIL gegenereerd, zonder native code. Hiervoor is een speciale versie van de CRT-library beschikbaar, die ook naar pure MSIL gecompileerd is. Uiteraard kan men, wanneer deze optie wordt gebruikt, geen inline assembly-code toepassen.
Er is ook een optie /clr:safe. Wanneer /clr:safe wordt gespecificeerd, is de gegenereerde code niet alleen pure MSIL, maar ook nog verifiable. Daarom kunnen in dat geval geen (unmanaged) pointers of native datatypes worden gebruikt. Omdat de CRT-libary (uiteraard) niet verifiable is, kan deze library niet gebruikt worden in een assembly die wordt gecompileerd met /clr:safe.

Hello World

Om een indruk te krijgen hoe een Visual C++ 2005 programma eruit ziet, is in codevoorbeeld 1 een HelloWorld-programma opgenomen. Opvallend is (met name voor wie ervaring heeft

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

Codevoorbeeld 1.
Hello World

```
using namespace System;

public ref class Managed
{
   public public:
      int Increment();
   private private:
      int value;
};

int Managed::Increment()
{
   return ++value;
}

int main()
{
   Managed ^m = gcnew Managed;
   Console::WriteLine("Waarde: {0}",m->Increment());
   m = nullptr;
   return 0;
}
```

Codevoorbeeld 2.
Declaratie en gebruik van een managed class

```
using namespace System;

enum class Color
{
   Red,
   Green,
   Blue,
};

interface class IA
{
     void f(int);
};

ref class Base abstract : IA
{
   public public:
   virtual void f(int) abstract;
};

ref class Derived : Base
{
   public public:
   literal int high = 999;
   static initonly Derived ^singleton = gcnew Derived;
   void f(int i) override
   {
      Console::WriteLine("f in Derived: {0}",i);
   }
   property Color BackGround
   {
      Color get()
      {
         return background;
      }
      void set(Color value)
      {
         background = value;
      }
   }
   d->BackGround = Color::Blue;
   return 0;
}
```

Codevoorbeeld 3.
Gebruik van nieuwe keywords

met de Managed Extensions for C++) dat er geen #using-directive nodig is, om de functionaliteit van mscorlib.dll te importeren. Dit is impliciet wanneer er gecompileerd wordt met de /clr-vlag. Verder valt op dat niet meer expliciet hoeft te worden aangegeven dat de string "Hello World" een CLR-string is. De compiler leidt zelf af uit de context of er een constante van het type System::String nodig is of eventueel een impliciete conversie. Het programma uit codevoorbeeld 1 kan ook met de /clr:safe-vlag worden gecompileerd, zodat er verifieerbare code wordt gegenereerd.

Managed en unmanaged data

Net als in de Managed Extensions for C++ bestaat ook in Visual C++ 2005 het onderscheid tussen managed en unmanaged data. Unmanaged data zijn de normale classes en structs in C++ die we al jaren kennen. Managed data zijn classes en structs die vallen onder de regels van de CTS (Common Type System) en die worden beheerd (gemanaged) door de CLR. Codevoorbeeld 2 bevat een voorbeeld van de declaratie en het gebruik van een dergelijke class.
Het keyword ref geeft aan dat het om een managed class gaat. Binnen een managed class wordt met twee access specifiers (public, private) aangegeven wat de accessibility van de members is. De meest beperkte specifier geeft de externe accessibility aan, de ruimste specifier geeft de interne accessibility (binnen de assembly) aan. Het is niet toegestaan een variabele van een managed class te declareren. Ook een pointer naar een object van een managed class is niet toegestaan. Uitsluitend een handle naar een object van een managed class kan worden gedeclareerd. Deze handle wordt aangegeven met een ^. Het creëren van een object van een managed class gebeurt door middel van de operator gcnew waarmee wordt aangegeven dat het object op de managed heap moet worden gecreëerd. De members van een object van een managed class worden benaderd via de handle met de operator ->. Er is een speciale waarde, namelijk nullptr die aangeeft dat een handle geen referentie naar een object bevat. Verder valt op dat het boxen (het creëren van een object-wrapper om een value-type) nu impliciet gebeurt. Dit verhoogt de leesbaarheid van de code aanzienlijk. Codevoorbeeld 3 toont een voorbeeld met interfaces, abstract base classes, enum, properties, override, initonly (readonly) en literal (const). Vele nieuwe keywords worden in dit voorbeeld gebruikt en de ervaren .NET-ontwikkelaar herkent ongetwijfeld de achterliggende principes.

Generics

Visual C++ 2005 kent ook CLR Generics, naast de bestaande templates voor unmanaged classes. Codevoorbeeld 4 bevat hier een voorbeeld van. CLR Generics hebben (globaal) dezelfde voordelen als C++-templates, namelijk het hergebruik van generieke classes en methods. Er is echter ook een aantal belangrijke verschillen tussen C++-templates en CLR Generics. Bij C++-templates wordt de code van de geïnstantieerde classes compile-time gegenereerd. Dit kan soms zeer grote executables opleveren. Bij CLR Generics wordt de code pas runtime (tijdens JIT-compilatie) gegenereerd. Bovendien wordt alleen nieuwe code gegenereerd voor instantiaties met een value-type als type-parameter. Wanneer een reference-type als type-parameter wordt meegegeven, wordt geen nieuwe code gegenereerd: voor alle reference-types wordt dezelfde code gebruikt.

```
using namespace System;
using namespace stdcli::language;

generic <typename T>
ref class DynamicArray
{
  public public:
    DynamicArray(int size)
    {
      if (size > 0)
      {
        this->size = size;
        table = gcnew array<T>(size);
      }
      else
        throw gcnew Exception("Illegal size");
    }
    int Add(T newelement)
    {
      if (count < size)
      {
        table[count] = newelement;
        return count++;
      }
      else
      {
        size = resize();
        table[count] = newelement;
        return count++;
      }
```

Codevoorbeeld 4. **vervolg op volgende bladzijde**

```
    }
    property T default[]
    {
      T get(int index)
      {
        if (index < count)
          return table[index];
        else
          throw gcnew Exception("Index out of bound");
      }
      void set(int index,T value)
      {
        if (index < count)
          table[index] = value;
        else
          throw gcnew Exception("Index out of bound");
      }
    }
  private private:
    int resize()
    {
      array<T> ^tmptable = gcnew array<T>(size*2);
      Array::Copy(table,tmptable,count);
      table = tmptable;
      return size*2;
    }
  array<T> ^table;
  int size;
  int count;
};

int main()
{
  DynamicArray<int> ^dai = gcnew DynamicArray<int>(10);
  for (int i=0;i<200;i++)
  {
    dai->Add(i);
  }
  for (int i=0;i<200;i++)
  {
    Console::WriteLine(dai[i]);
  }
  DynamicArray<String^> ^das = gcnew DynamicArray<String^>(20);
  das->Add("abc");
  Console::WriteLine(das[0]);
  return 0;
}
```

Codevoorbeeld 4.
Voorbeeld van CLR Generics

Uiteraard zijn er ook beperkingen ten opzichte van de templates uit C++. Zo is het niet mogelijk een class af te leiden (inheritance) van een template-parameter. Ook het gebruik van static members (data-members en function-members) van een template-parameter is niet toegestaan. Bovendien is het gebruik van instance-members sterk beperkt. Alleen die instance-members, waarvan de compiler weet dat ze op het betreffende type zijn gedefinieerd, kunnen worden gebruikt. In principe zijn dat dus alleen de members van class Object. Wanneer andere members moeten worden gebruikt, moet er een constraint op de template-parameter worden gedefinieerd waarbij wordt aangegeven welke interface de betreffende parameter minimaal moet implementeren. Pas dan kunnen de members van die interface op de template-parameter gebruikt worden. Een voorbeeld daarvan is te vinden in codevoorbeeld 5.
Hier wordt op de class DynamicArray een method BinarySearch() geïmplementeerd. Er is slechts een deel van de code gegeven, de rest van de code is gelijk aan de code in codevoorbeeld 4. Om

```
using namespace System;
using namespace stdcli::language;

generic <typename T>
where T : IComparable<T>
ref class DynamicArray
{

    /* some code deleted */

    int BinarySearch(T arg)
    {
      int low=0;
      int high=count-1;
      if (!sorted)
      {
        Array::Sort(table,0,count);
        sorted = true;
      }
      while (low <= high)
      {
        int mid = (low+high)/2;
        int result = arg->CompareTo(table[mid]);
        if (result == 0)
          return mid;
        else if (result < 0)
          high = mid - 1;
        else
          low = mid + 1;
      }
      return -1;

    }
  private private:

    bool sorted;
    array<T> ^table;
    int size;
    int count;
};

int main()
{
  DynamicArray<int> ^dai = gcnew DynamicArray<int>(10);
  for (int i=200;i>0;i--)
  {
    dai->Add(i);
  }
  for (int i=0;i<200;i++)
  {
    Console::WriteLine(dai[i]);
  }
  int res = dai->BinarySearch(10);
  if (res >=0)
    Console::WriteLine("{0},{1}",res,dai[res]);
  return 0;
}
```

Codevoorbeeld 5.
CLR Generics met constraints

binair te kunnen zoeken in het betreffende array moet er een vergelijking gedaan kunnen worden tussen het zoekargument en de elementen van de array. De method CompareTo() kan echter alleen worden aangeroepen op het array-element - waarvan het type de template-parameter is - als bij de template-parameter als constraint wordt toegevoegd dat de betreffende class de IComparable-interface implementeert. Deze constraint wordt aangegeven met het keyword where. Als men een instantie van de class DynamicArray<> maakt, controleert de compiler of de class die als type-parameter meegegeven wordt inderdaad de interface IComparable implementeert.
Een alternatief voor Generics is het gebruiken van het type Object als (generiek) elementtype. Vóór Visual Studio 2005 was dit de enige manier. In zo'n geval wordt binnen de class DynamicArray een array van type Object gebruikt om de elementen in op te slaan. Het voordeel van het gebruik van Generics ten opzichte van deze oplossing zit voor value-types met name in een performancewinst, omdat er niet meer geboxt hoeft te worden. Voor reference-types is het voordeel vooral dat de compiler betere type-controles kan uitvoeren, omdat bij de compiler bekend is wat het element-type is. Codevoorbeeld 4 laat ook zien hoe Visual C++ 2005 omgaat met CLR-arrays. Hiervoor wordt een generic type stdcli::language::array<T> gebruikt. Een array van dit type voldoet aan de eisen die .NET aan een array stelt, zoals de controle op boundaries, enzovoort. Ook de methods van de class System.Array kunnen op zo'n array gebruikt worden; bijvoorbeeld Array::Copy() zoals in het codevoorbeeld. Ook het gebruik van een indexer (default indexed property) wordt in codevoorbeeld 4 getoond. Door de declaratie van de member property T default[] is het mogelijk objecten van de class DynamicArray met behulp van [] te indexeren. De toepassing hiervan wordt getoond in functie main().

Interoperability

Visual C++ biedt nog steeds de meest flexibele mogelijkheden van alle .NET-talen op het gebied van interoperability. Naast de gebruikelijke DllImport - die de meeste andere talen ook bieden- waarmee functies uit een DLL kunnen worden aangeroepen, is er de mogelijkheid om functies aan te roepen uit een static library of via een import-library. Hierbij maakt Visual C++ gebruik van het feit dat de compiler ook native-code kan genereren. Deze mogelijkheid is dan ook alleen beschikbaar voor mixed assemblies, dat wil zeggen assemblies die gecompileerd zijn met de /clr-vlag, en dus niet met /clr:pure of /clr:safe. Assemblies die Pure MSIL bevatten - die dus gecompileerd zijn met /clr:pure - kunnen geen gebruik maken van de mogelijkheid om native entry-points aan te roepen. Wel kunnen ze gebruik maken van native datatypes, waardoor functies in een DLL die parameters verwachten die niet eenvoudig in .NET-datatypes weer te geven zijn toch gemakkelijk kunnen worden aangeroepen met de juiste parameters.

Een nieuwe weg

Met Visual C++ 2005 heeft Microsoft het pad verlaten dat geleid heeft tot de Managed Extensions for C++. Men heeft nu gekozen voor een veel leesbaarder taal waarin C++-programmeurs die kennis hebben van .NET vrij eenvoudig .NET-applicaties kunnen bouwen. De prijs die men hiervoor betaalt, is het verlaten van de ANSI C++ standaard. Dat is natuurlijk jammer, want standaarden zijn er niet voor niets, maar het alternatief was ook niet aantrekkelijk. De Managed Extension for C++ waren weliswaar ANSI-compliant, maar het was een onleesbare taal geworden die erg lastig te programmeren was. Naast de veranderingen in de taal zelf zijn er ook mogelijkheden gekomen om Pure MSIL-assemblies te bouwen en zelfs om verifieerbare code op te leveren. Dat maakt het mogelijk Visual C++ ook te gebruiken in situaties waarin verifieerbaarheid een absolute eis is.

Nuttige internetadressen

VC++ Dev Center: http://msdn.microsoft.com/visualc/
Bèta: http://lab.msdn.microsoft.com/vs2005/
RSS feed: http://msdn.microsoft.com/visualc/rss.xml
Webcast: http://msdn.microsoft.com/msdntv/episode.aspx?xml=episodes/en/20040708visualctm/manifest.xml

Gert Jan Timmerman is werkzaam als trainer/consultant bij Info Support www.info-support.nl. Hij is gespecialiseerd in Visual C++ en interoperability tussen .NET en andere omgevingen. Hij is de auteur van een aantal artikelen over deze onderwerpen in eerdere nummers van .NET Magazine.