世界はオブジェクトの海に浮かぶ

.NET Framework で楽しむ オブジェクト指向

第20回 C++/CLIの有効な使いみち

επιστημη
えぴすてーめー

C++/CLIの魅力

10月14日に新宿マイクロソフトの一部屋をお借りしてわんくま同盟主催の勉強会が催されました（次回は12月に大阪／東京でやりますんでぜひともいらしてくださいな）。この勉強会の席で僕もひとコマ、一時間のセッションを引き受けました。ネタは「C++/CLIカクテル・レシピ」。

レベル >>> Level

1 2 3 4 5

言語 >>> Language

- C#
- C/C++
- C++/CLI

ツール >>> Tool

- .NET Framework 2.0 SDK
- Visual Studio 2005 Professional

C++/CLIはC++にいくつかの独自文法を導入することで.NETのManagedオブジェクトを扱えるようにしたもので、いわば“.NET対応C++”です。C++/CLIの売りは.NET対応というだけじゃなく、既存のC/C++文法を必要に応じて“混ぜて”使うこともできるところ。従来のC/C++ライブラリやWin32 APIも自由に呼び出せます。なのでC#やVB.NETなどの純粋.NET言語では少なからず困難／面倒なことを簡単に実現できるわけで、そこがC++/CLIの大きな魅力となっています。

C++/CLIの.NET拡張部分だけを使ってもアプリケーションの実装は可能です。が、それだったらハナっからC#やVB.NETで書いたほうがよっぽど素直だし楽なわけで、既存のC/C++と必要に応じてミックスして使えるところがC++/CLIの持ち味だと思ってます（Tech・Ed 2006で僕が5分間だけ喋ったのも既存のCライブラリにC++/CLIで薄い皮をかぶせ、.NETライブラリに仕立てるお話でした）。いくつかのシチュエーションを考え、C++/CLIの面白くて有効な使いみちを考えよう、というのがセッションのお題「C++/CLIカクテル・レシピ」ってわけです。

で、このセッションの準備のためにかなり時間を費やすことになり、なかなか今月の原稿に着手できません。それじゃいっそのこと勉強会のネタを使っちまえー、ということで、今回のお題は「C++/CLIの有効な使いみち」。さっそくいってみましょう。

C++/CLIのための下準備 ～文字列の扱い

Visual Studio 2005のC++コンパイラ「Visual C++ 8.0」はコンパイル時のオプションに「/clr」を付加することで.NETに対応します。C++/CLIで導入された拡張文法を受け付けてくれるわけね。

```
int main() {
   int value = 123;
   printf("%d¥n", value);   // C
   std::cout << value <<std::endl;   // C++
   System::Console::WriteLine(value);   // C++/CLI
   return 0;
}
```

このようにC、C++、C++/CLIを自由に“混ぜ書き”できるのが大きな特徴です。さらに従来のCライブラリだけでなく、C++ライブラリやWin32 API、そして.NET Frameworkライブラリが使い放題。プログラマの手駒がぐっと増えることとなりました。

C++/CLIでは従来のメモリ管理に従った（malloc/freeあるいはnew/delete）Nativeオブジェクトと.NETの管理下にあるManagedオブジェクトをひとつのコードの中で混在させることができます。ですがこれは従来のNativeオブジェクトに、さらにManagedオブジェクトが追加されたということであり、NativeとManagedの双方を区別なく扱えるということではありません。両者はやはり“別物”なんです。

```
#include <iostream>
#include <string>

int main() {
   std::string native = "Native文字列";
   System::String^ managed = "Managed文字列";

   // これはOK
   std::cout << native << std::endl;
   System::Console::WriteLine(managed);

   // これはNG（コンパイルエラー）
   std::cout << managed << std::endl;
   System::Console::WriteLine(native);

   return 0;
}
```

なので単なる共存ではなく、NativeとManagedを混在／融合させたいなら、必要に応じて両者間の適切な変換を行なわなければなりません。int、longなどの数値型についてはNativeもManagedも区別はありません。が、文字と文字列は異なります。

まず、Managedでの文字／文字列はUnicodeですからManaged文字に対応するNativeな型はwchar_tです（charではありません）。

```
System::String^ clistr = L"Managed文字列";
char ch = clistr[7];          // マチガイ
wchar_t wch = clistr[7];   // OK。'文'で一文字
```

同様にManaged文字列に対応するのはchar*ではなくwchar_t*となります。なのでSystem::String^からwchar_t*を取り出すにはSystem::StringのメソッドToCharArrayを用いて、

```
wchar_t* wptr = &(clistr->ToCharArray()[0]);
```

とする……。残念でした。ToCharArray()によって得られるwchar_tの列はManaged配列であり.NETのメモリ管理下にあります。ですから何らかのタイミングでこの領域が水面下で移動し、wptrに格納したポインタ値に意味がなくなるかもしれません。使っている間はメモリの移動を抑止、つまりピンで留めておかなければなりません。

```
pin_ptr<wchar_t> wptr = &(clistr->ToCharArray()[0]);
std::wstring wstr = wptr;   // Native文字列にコピー
```

pin_ptrはその使命を終える（スコープから抜ける）とピンが外れ、メモリの移動が許されます。

今までさんざん使ってきたNativeマルチバイト文字列char*（std::string）との変換はワイド文字列wchar_t*（std::wstring）より面倒で、UnicodeとANSIとの変換が必要になります（リスト1）。そんなわけでManaged文字列System::String^とNative文字列との間を行ったり来たりする必要があるなら、Native文字列にはstd::stringより、簡単でUnicode／ANSI変換の必要もない（ので速い）std::wstringを使ったほうがよろしいかと。そのときワイドストリームを利用するなら、ストリームの利用に先立って適切にlocale設定しておいてください（リスト2）。

余談ですが、System::String^からchar*への変換にsprintfが使えるみたいです。