アドバンスド Windows プログラミング

Microsoftテクノロジーをめぐる冒険

第 10 回

P/Invoke～メモリとポインタ編

こだか かおる
KODAKA, Kaoru

はじめに

先日、といっても昨年の話ですが、Visual Studioを利用しているユーザーを対象に、新しいユーザーコミュニティが誕生しました。その名もVSUG（Visual Studio User Group）です。今まで、Microsoft製品を対象としたユーザーコミュニティはいくつかありましたが、どれもVisual BasicやC#などの開発言語、あるいは.NET FrameworkやASP.NETといったプラットフォームを対象としたもので、「Visual Studio」という括りのユーザーコミュニティはVSUGがはじめてといってよいでしょう。

私も、.NET Frameworkフォーラムのボードリーダーとして参加させていただいています。まだまだ生まれたばかりのコミュニティで、今後どのように成長していくかは参加者次第。Visual Studioに興味がある方なら、どなたでも参加でき、参加の際に会費などはかかりません。VSUGのWebサイト（http://vsug.jp/）から会員登録が行なえますので、まだ登録していない皆さんは、ぜひ参加してください。

Level
1 2 3 4 5

Technology Tools
- ◯ Visual Basic
- ✓ Visual C#
- ◯ Visual C++
- ◯ SQL Server
- ◯ Oracle
- ◯ Access
- ◯ ASP.NET
- ✓ Other:
  - ↓
  - Visual Studio 2005
  - Win32 API

P/Invokeで.NET Frameworkの限界を超える

.NET Frameworkの世界は広大で、できないことはないような気すらしてきます。しかし、そうは問屋がおろしません。.NET Frameworkですべて完結できればよいのですが、実際はなかなかうまくいかないものです。となると、.NET Framework以外の手段を借りてくることになりますが、その方法のひとつがP/Invokeです。

P/Invokeとは、要するにWin32（64）APIの呼び出しのことです。なお、64ビットについては、環境が用意できなかったということもあり、この場では説明しません。ということで、今回はWin32 APIに限定した話となりますが、どうかご容赦を。

.NET Framework関連のドキュメントや参考書を見ても、表立ったP/Invokeについての説明はありません。それは、「今後はすべて.NET Frameworkで解決していこう」というMicrosoftの目論見があるからでしょう。

P/Invokeを使う上では、Cの知識が必須です。それはつまり、Win32 APIがCで作られ、Cを対象にしたものだということを意味しています。そういったバックグラウンドがないと、P/Invokeを使いこなすことはできません。それ

では、まずC言語の本を買ってきてください……というわけにもいきませんので、今回の記事では、最低限のCの話題も盛り込みつつ、いかに.NET FrameworkからP/Invokeを使っていくか、そのあたりを紹介していきたいと思います。

メモリについて

Visual BasicやC#といった高級言語では、メモリについてあまり意識する必要がありません。せいぜい、ガベージコレクタに関連してオブジェクトの生成と破棄、さらにはアンマネージドなリソースについて意識する程度です。しかし、Cのような低級言語ではそうはいきません。すべてがアンマネージドであり、メモリの確保と解放には、プログラマがきちんと責任を持つ必要があります。

なお、ここでいう「低級」「高級」という言葉は、機械語からの距離を示しています。すなわち、機械語から近ければ（機械語に似ていれば）「低級」、機械語から遠ければ遠いほど「高級」ということになります。必ずしもCがVisual BasicやC#に比べて劣っているという意味ではありませんので、誤解しないようにしてください。また、この文脈によれば、ExcelやWordのマクロ（VBA）は、Visual BasicやC#から比べ、さらに高級という位置づけになります。

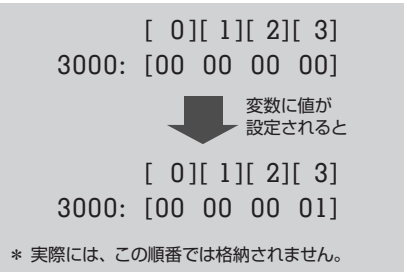
話を戻します。P/Invokeを使う際、一番ハマるのがメモリとポインタに関することではないでしょうか。DllImportでどう宣言をすればいいのか、どのAPIを使えばよいか、といったことは、慣れと検索スキルの問題です。ということで、ここで言い切ってしまいますが、

「メモリとポインタがわかれば、Win32APIは怖くない！」

のです。

まずは、メモリについてきちんと理解しておきましょう。メモリは目で見えないため、なかなかイメージしづらい部分があります。わかりやすい例えとしては、バイト型の配列です。配列の要素を示すインデックスがアドレスになります。最初の要素が0番地で、そこから「1」ずつ増えていきます（図1）。変数が宣言されたとき、メモリ上にその領域が確保されます。たとえば、int型は4バイトですので、4バイト分の領域です。

図1：メモリ



＊ 実際には、この順番では格納されません。

メモリの確保と解放

メモリは単なる入れ物でしかありません。つまり、メモリを確保したからといって、型などの属性が付与されるわけではないのです。「型」それぞれの大きさ分のメモリ領域が確保されるに過ぎません。それを、プログラム言語が「型」として扱うことになるわけです。ここの部分を間違えないようにしておいてください。疑っている人がいるかもしれませんので、実験用に簡単なサンプルプログラムを作ってみました（図2）。

図2：メモリ取得サンプル

このサンプルプログラムでは、型とメモリが独立したものであることを示すために、int型で書き込んだ値をいろいろな型で取得しなおしています（リスト1）。

それでは、サンプルプログラムを見ていきましょう。まずは、IntPtr型の変数を用意して、メモリを確保します。.NET Frameworkの世界でアンマネージドメモリを明示的に確保するには、Marshal.AllocXXX()メソッドを利用します。ここでは、Marshal.AllocHGlobal()メソッドを利用することにしましょう。どこにどのような仕組みを利用してメモリを確保するかによって、利用するメソッドが異なります。

今回は、Marshal.AllocHGlobal()メソッドを利用して、4バイトのメモリを確保しました。このメソッドにより、IntPtr型の値が返ってきます。この値は、確保したメモリのアドレスになり