特2集

誰も語らなかった Visual C++ .NET

C/C++のスタンダード開発環境の詳細

Visual C++ユーザーのみなさん、どのバージョンを使ってます？　もしかしたら、まだVC++6.0を使っていませんか？　すでにVC++6.0が発売されてから5年半。VC++は着実に進化しています。本稿では、Visual C++ .NET 2003に追加されている新機能の中から「これは！」という機能を一気に紹介してしまいます。VC++6.0やそれ以前のバージョンをご使用の方は必見です。

マイクロソフト株式会社
デベロッパーマーケティング本部
デベロッパー製品部マネージャ
田中 達彦
TANAKA, Tatsuhiko

Visual C++ .NET 2003

Visual C++ .NET 2003（以下VC++.NET2003）には、以下の2つの側面があります。

①.NET Frameworkに対応したアプリケーション作成
②VC++6.0と同様にWin32 APIベースのアプリケーション作成

この両方を組み合わせたアプリケーションも作ることができますが、今回はその中から②のWin32 APIベースのアプリケーションを作る側面にスポットライトを当てます。そして、.NET Frameworkを使用していない方に向けて、VC++.NET 2003に乗り換えるべき理由を解説してゆきます。

とくにコンパイラ系の新機能は、VC++6.0のプロジェクトファイルをVC++.NET2003で開き、コンパイラオプションを指定してリビルドするだけで使用することができます。非常に簡単に試すことができるので、ぜひ検討してください。

もちろん、Windowsの次期バージョンである“Longhorn”に備えて、.NET Framework対応アプリケーションを構築するためにも、VC++.NET2003に乗り換えたほうが良いのは言うまでもありません。

製品構成

本稿ではVC++.NET2003という名称を使用します。バージョン2002以降、製品名としての「VC++」という名称は、Standardと呼ばれるエディションにしかついていません。ProfessionalとEnterpriseの両エディションは、Visual StudioブランドまたはMSDNブランドとして販売されています。

Standardには「コンパイラの最適化の機能」がついていませんが、価格が安いため、学習やホビー用として適しています。最適化されていないアプリケーションの実行速度は、最適化されているアプリケーションと比較してかなり劣るので、商用アプリケーションや他の人に使ってもらうフリーソフトウェアの作成にはProfessional以上のエディションをお勧めします。

なお、Visual Studio .NETの評価版にも最適化の機能はついていません。

今回解説する機能は、Professional以上のエディションであればすべて使用することができます。

コンパイラの進化

VC++のコンパイラは、常に業界のトップレベルの性能を誇っています。最適化の性能や、コード削減のための工夫など、開発者の負担を軽減しながら最大の性能を持つアプリケーションを構築できるように設計されています。

進化した最適化

VC++.NET2003では、Pentium 4またはAMD Athlonプロセッサ用にコードを最適化することができるようになりました。コンパイラオプションとして「/G7」を指定することに

リスト1：/G6オプションを指定

```
int _tmain(int argc, _TCHAR* argv[])
{
      int i;
      i = getchar();
  push    offset __iob (408040h)
  call    _filbuf (40102Dh)
      return i * 15;
  imul    eax,eax,0Fh    ; imul命令のlatencyは14
  add     esp,4
}
  ret
```

リスト2：/G7オプションを指定

```
int _tmain(int argc, _TCHAR* argv[])
{
      int i;
      i = getchar();
  push    offset __iob (408040h)
  call    _filbuf (401036h)
  mov     ecx,eax   ; mov命令のlatencyは0.5
      return i*15;
  shl     eax,4     ; shl命令のlatencyは4
  add     esp,4     ; スタックポインタを元に戻している。
                    ; 今回の演算には関係ないが、パイプライン
                    ; の関係でこの位置に配置されている
  sub     eax,ecx   ; sub命令のlatencyは0.5 (latencyは合計で5)
      // 上記の例では、単純に15を掛けるのではなく、iの値を
      // 16倍したものからiを引いている
      // imul命令を単純な命令に置き換えることにより、処理
      // 速度の向上を実現している
}
  ret
```

より、Pentium 4またはAthlon用にコードを最適化し、コードの内容によっては約20%の速度の向上を図ることができます。浮動小数点演算を多用しているアプリケーションなどでは、特にこの最適化の効力が発揮されます。

まずは、簡単なサンプルコードで比較してみましょう。

リスト1はコンパイラオプションに「/G6」を指定し、Pentium Pro/Pentium Ⅱ/Pentium Ⅲ用に最適化を行なったものです。対してリスト2は、「/G7」を指定し、Pentium 4用に最適化を行なったものです。コード中の“latency”は命令を実行するために必要なクロックサイクル数ですが、このサンプルコードのような例では、乗算を行なうためにimul命令を使用するよりも、「mov」「shl」「sub」という3つの命令を使用したほうが合計のlatencyの数値が小さくなり、処理が高速になります。

リスト2はある変数を15倍するときの例ですが、リスト3のように14倍するときや12倍するときなどにも独自の演算の展開を行ない、高速化を実現しています。

リスト3：/G7オプションを設定したときのその他の例

例1

```
    return i * 14;  // 14を掛ける場合は、一旦iに8を掛けた
                    // ものからiを引き（iを7倍する）、
                    // その値を2倍している
mov     ecx,eax     ; iの内容（eaxレジスタ）を
                    ; ecxレジスタに代入
lea     eax,[ecx*8] ; eaxレジスタにecx*8の値を代入
sub     eax,ecx     ; eaxレジスタからecxレジスタの
                    ; 値を引くことにより、eaxレジスタに
                    ; iの7倍の値が入ることになる
add     eax,eax     ; iの7倍にiの7倍を足すことにより、
                    ; iの14倍になる
```

例2

```
    return i * 12;  // 12を掛ける場合には、一旦iに2を
                    // 掛けたものにiを足し（iを3倍する）、
                    // その値を2倍し、さらに2倍している
lea     eax,[eax+eax*2] ; iの値が入っているeaxレジスタに
                        ; eaxレジスタを2倍したものを足し
                        ; (すなわちiの3倍)、その値をeax
                        ; レジスタに代入
add     eax,eax         ; iの3倍にiの3倍を足すことにより
                        ; eaxレジスタにiの6倍の値が入る
add     eax,eax         ; さらにiの6倍にiの6倍を足して
                        ; iの12倍にする
```

また、Pentium 4は、減算を行なう場合にdec命令を使用するよりもsub命令を使用したほうが速いため、“-1”の減算を行なうときにはsub命令を使用します。/G7オプションは最適化を行なう際にPentium 4特有の命令を使用しないので、/G7オプションを指定して作成したアプリケーションをPentium Ⅲ上などで動かすこともできます。

さらに、コンパイラオプションで「/arch:SSE」または「/arch:SSE2」を指定することにより、ストリーミングSIMD拡張命令であるSSEとSSE2に対応したコードを生成することもできます。

バッファのセキュリティチェック

VC++.NET2003のコンパイラには、セキュリティチェックの機能も追加されました。

バッファオーバーランを発生してしまうアプリケーションは、一見正しく動作しているように見えても、環境が変わったり、ユーザーが予期せぬ操作をした場合にアプリケーションが正常に動作しなくなる可能性があるなど、発見しにくい問題を抱えています。さらに、バッファオーバーランを発生させるコードは、往々にしてウィルスのターゲットになる場