Inline Assembly Code
Chapter 12

As you have seen, it can be quite tedious to write programs using assembly language compared to a high level language like Pascal or C++. However, assembly does give you the advantage of speed and direct access to the computer. There may be some operations that are impossible in the high-level language that requires the use of assembly (e.g., drivers). There may also be optimizations in assembly that will run much faster than the high-level language equivalent. This is because human programmers are often able to write more efficient code than the compiler.

With Microsoft Visual C++ 6.0, you can use the inline assembly to embed assembly-language instructions directly in your C and C++ source programs without extra assembly and link steps. The inline assembler is built into the compiler — you don't need a separate assembler such as MASM.

Important Note: Programs with inline assembler code are not fully portable to other hardware platforms. If you are designing for portability, avoid using inline assembler.

In Visual C++, the **__asm** directive is used to indicate that assembly begins. This can be played in a single statement or for a block of statements. The syntax is:

```
__asm statement
```

or

```
__asm {
        statement1
        statement2
        …
}
```

Note that there are *two* underline characters that precede the asm statement.

A few points warrant mention:

- Use // to denote comments rather than ;

- You cannot use data definition directives (e.g. db, dw, dd). You CAN access variables that are defined in C++ and normally available in the program's scope. Refer to the variables just like you would refer to an assembly variable. Be sure to note the sizes; e.g. something defined as char takes 1 byte, something defined as short takes 2 bytes, int and long are both 4 bytes, etc.

- You cannot use assembler operators like OFFSET. Instead use the LEA instruction (load effective address):

  LEA ebx, buffer

  Puts the 32 bit offset of the variable "buffer" into EBX

- Numeric constants may be defined in either assembler style or in C style. For example, 0FFh or 0xFF both mean the hex value "FF."

- When using inline assembly, the memory model is the FLAT memory model. Since Visual Studio doesn't run on 286 and lower machines, you also have the 386 instructions available in 32-bit protected mode.

- You cannot assume registers contain any particular values, but in general it is safe to use EAX, EBX, ECX, EDX, ESI, and EDI in your code without restoring them to their original values. This is because C++ doesn't expect these values to be preserved between statements (one reason why you can write more efficient assembly code than the C++ compiler!)

- You can use the LENGTH, SIZE, PTR, and TYPE operators.

- You can even call C++ function calls or jump to labels OUTSIDE your asm block.

- Write your apps as CONSOLE applications.

Here is a simple example:

```
// Compute num * 2^power
int power2(int num, int power)
{
  int returnval;
  __asm
  {
    mov eax, num       // Get first argument
    mov ecx, power     // Get second argument
    shl eax, cl        // EAX = EAX * ( 2 to the power of CL )
    mov returnval, eax
  }
  return returnval;
}
```

Notice how the inline assembly function has direct access to the parameters. Also notice that an INT takes up 4 bytes or 32 bits, so we need to use the extended registers like EAX to ensure that the sizes of the operands match up.

Here is a file encryption example, where we have written an assembly program to encrypt a stream of data using XOR

```
#include <iostream>
using namespace std;

// Prototype
void TranslateBuffer(char *buf, unsigned count, unsigned char encryptChar);

int main()
{
	char buf[100]="Attack at dawn!";

	TranslateBuffer(buf, strlen(buf), 'z');
	cout << "The encoded string is: " << buf << endl;
	TranslateBuffer(buf, strlen(buf), 'z');
	cout << "The decoded string is: " << buf << endl;
	return 0;
}

// Encode a string of data using XOR
void TranslateBuffer(char *buf, unsigned count, unsigned char encryptChar)
{
	__asm {
		mov esi, buf
		mov ecx, count
		mov al, encryptChar
L1:
		xor [esi], al
		inc esi
		loop L1
	}
}
```

As a side note, when you insert the __ASM directive the compiler adds 16 additional instructions to save and restore registers. Keep this in mind if you are really trying to optimize code! For example, if you use a loop, you may want to put the loop in assembly code rather than put your assembly code in a C++ loop.

It is also possible to link C++ programs directly with assembly object files, but we will skip this technique for this class.