# Calling C++ Functions

By Kip R. Irvine. *Last update: 11/25/2002*

This article is written primarily for users of Assembly Language for Intel-Based Computers, 4th Edition. You may copy and print the article, as long as you do not alter its content.

Chapter 12 does a reasonably good job of showing you how to call assembly language procedures from C++ programs. But what about calling C++ functions from assembly language? There are at least a couple of reasons for doing so:

- Input-output is more flexible under C++, with its rich iostream library. This is particularly true if you want to work with floating point numbers!
- C++ has extensive math libraries, which are not available for assembly language.

### Declaring a C++ Function

All C++ functions called from assembly language code must be defined with the **"C"** and **extern** modifiers. Here's the basic syntax:

```
extern "C" funcName( paramlist )
{ . . . }
```

Rather than modifying every function defintion, it's easier to create a list of function prototypes inside an extern "C" block:

```
extern "C" {
   void mySub();
   int Sub2( int x, int y );
   etc.
}
```

Then you can omit the **extern** and **"C"** modifiers from the function implementations.

## Multiplication Table Example

Let's write a simple application that prompts the user for an integer, multiplies it by ascending powers of 2 (from $2^1$ to $2^{10}$) using bit shifting, and redisplays each product with leading padded spaces. We will use C++ for the input-output. The assembly language program will contain calls to three functions written in C++.

**Assembly Language Module**

The assembly language module contains only one function, named **DisplayTable**. It calls a C++ function named **askForInteger** that inputs an integer from the user. It uses a loop to repeatedly shift an integer named **intVal** to the left and display it by calling **showInt**.

```
; ASM function called from C++
.586
.model flat,C

; External C++ functions:
askForInteger PROTO
showInt PROTO, value:SDWORD, outWidth:DWORD
newLine PROTO

OUT_WIDTH = 8
START_POWER = 10

.data
intVal DWORD ?

.code
;-----------------------------------------
DisplayTable PROC
;
; Input an integer n and display a
; multiplication table ranging from n * 2^1
; to n * 2^10.
;-----------------------------------------
   INVOKE askForInteger          ; call C++ function
    mov  intVal,eax              ; save the integer
    mov  ecx,START_POWER         ; loop counter

L1: push ecx                     ; save loop counter
    shl  intVal,1                ; multiply by 2
   INVOKE showInt,intVal,OUT_WIDTH
   INVOKE newLine                ; output CR/LF
   pop  ecx                      ; restore loop counter
   loop L1

   ret
DisplayTable ENDP
END
```

Note an important detail: ECX must be pushed and popped before calling **showInt** and **newLine** because compiled C++ code routinely alters the contents of general-purpose registers. Also, we assume that **askForInteger** returns its integer result in the EAX register. This is standard practice for C and C++ programs.

The ASM module contains three function prototypes that permit the connection to the external C++ functions. The order and sizes of parameters in the **showInt** prototype directly match the parameter list of the C++ function:

```
; External C++ functions:
askForInteger PROTO
showInt PROTO, value:SDWORD, outWidth:DWORD
newLine PROTO
```

You do not have to use the INVOKE statement, of course. The same effect could be produced by using PUSH and CALL instructions. This is how the call to **showInt** would look:

```
push OUT_WIDTH            ; push last argument first
push intVal
call showInt              ; call the function
add  esp,8                ; clean up stack
```

You must follow the C language calling convention, where arguments are pushed on the stack in reverse order, and the caller is responsible for removing the arguments from the stack after the call.

**C++ Program**

Now we're ready to look at the C++ program. It begins by executing **main**, as do all C++ programs. This ensures the execution of required C++ startup code. It contains function prototypes for the external assembly language procedure, as well as for the three functions being exported from this module:

```
// main.cpp

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

extern "C" {
   // declare external ASM procedure:
   void DisplayTable();

   // declare local C++ functions:
   int askForInteger();
   void showInt( int value, int width );
   void newLine();
}

// program entry point
void main()
{
   DisplayTable();              // call ASM procedure
```

```
}

// Prompt the user for an integer.

int askForInteger()
{
   int n;
   cout << "Enter an integer between 1 and 90,000: ";
   cin >> n;
   return n;
}

// Display a signed integer with a specified width.

void showInt( int value, int width )
{
   cout << setw(width) << value;
}

// Display a newline.

void newLine()
{
   cout << endl;
}
```

The code written for the three functions (**askForInteger, showInt,** and **newLine**) is standard C++ code.

**Building the Project**

If you need help in building combined C++/Assembly Language projects, there are two tutorials on the book's Web site: one for Visual C++ 6.0, and another for Visual C++.Net. Click on the *Integrated Development Environments* link from the home page.

## Calling C Library Functions

The C language has a rich set of functions named the *Standard C Library.* The same functions are available to all C++ programs, and by connection, to an assembly language module attached to a C++ program.

You must declare a prototype in your code for each C function you plan to call. The following prototypes are for **system**, **printf**, and **scanf**. The system function lets you pass an operating system command such as "cls" or "dir" that would normally be typed at the command

prompt. The **printf** function displays strings, integers, and reals in a variety of formats. The **scanf** function reads strings, integers, and reals from standard input:

```
system PROTO, pCommand:PTR BYTE
printf PROTO, pString:PTR BYTE
scanf  PROTO, pFormat:PTR BYTE,pBuffer:PTR BYTE
```

You can usually find the original C function prototypes by accessing the help system supplied with your C++ compiler. As is the case with MS-Windows functions, you have to translate the C prototypes into assembly language prototypes.

## Directory List Program

Let's write a short program that clears the screen, displays the current disk directory, and asks the user to enter a filename. (You might want to extend this program so it opens and displays the selected file.)

### C++ Stub Module

The C++ module simply contains a call to **asm_main**, so we can call it a *stub module*:

```
// main.cpp
// stub module: launches assembly language program

extern "C" void asm_main();      // asm startup proc

void main()
{
   asm_main();
}
```

### ASM Module

The ASM module contains the function prototypes, several strings, and a **fileName** variable. It calls the **system** function twice, passing it "cls" and "dir" commands. Then **printf** is called, displaying a prompt for a filename, and **scanf** is called so the user can input the name:

```
; ASM program launched from C++
.586
.model flat,C

; Standard C library functions:
system PROTO, pCommand:PTR BYTE
printf PROTO, pString:PTR BYTE
scanf  PROTO, pFormat:PTR BYTE, pBuffer:PTR BYTE

.data
str1 BYTE "cls",0
str2 BYTE "dir/w",0
```

```
str3 BYTE "Enter the name of a file: ",0
str4 BYTE "%s",0
fileName BYTE 60 DUP(0)

.code
asm_main PROC
    ; clear the screen, display disk directory
    INVOKE system, ADDR str1
    INVOKE system, ADDR str2

    ; ask for a filename
    INVOKE printf, ADDR str3
    INVOKE scanf, ADDR str4, ADDR fileName

    ; redisplay the filename
    INVOKE printf, ADDR fileName
    ret                   ; return to C++ main
asm_main ENDP
END
```

The **scanf** function requires two arguments: the first is a pointer to a format string ("%s"), and the second is a pointer to the input string variable (**fileName**). We will not try to explain standard C functions here, because there is ample documentation elsewhere. You may even want to read the standard "bible" of C, named **The C Programming Language** by Kernighan and Ritchie.

If you get good enough at calling C functions, you will no longer need the Irvine32 library!

*End of Article*