The __fortran calling convention isn't the calling convention used by FORTRAN

devblogs.microsoft.com/oldnewthing/20101222-00

December 22, 2010

Raymond Chen

Although the Microsoft C compiler supports a calling convention called `__fortran`, that's just what the calling convention is called; its relationship with the FORTRAN programming language is only coincidental. The `__fortran` keyword is now just an old-fashioned synonym for `__stdcall`.

Various FORTRAN compilers use different calling conventions; the one I describe here applies to the now-defunct Microsoft Fortran PowerStation.

Fortran Powerstation pushes parameters on the stack right-to-left, with callee-cleanup. (So far, this matches `__fortran` aka `__stdcall`.) Function names are converted to all-uppercase, with an underscore at the beginning and @n appended, where `n` is the number of bytes of parameters. (This still matches `__stdcall` aside from the uppercase conversion.)

As for how the parameters are passed, well, that's where things get weird. FORTRAN natively passes all parameters by reference. This is the source of a famous classic FORTRAN bug known as *constants aren't*.

```
      PROGRAM MYSTERY
      CALL MAGIC(1)
      PRINT *, 'According to the computer, 3 + 1 is ', ADDUP(3, 1)
      END
      FUNCTION ADDUP(I, J)
      ADDUP = I + J
      END
C     What does this subroutine actually do?
      SUBROUTINE MAGIC(I)
      I = 9
      RETURN
      END
```

(It's been a long time since I've written a FORTRAN program, so I may have gotten some of the details wrong, but any errors shouldn't detract from the fundamental issue.)

When you run this program, it says

```
According to the computer, 3 + 1 is 12
```

How did that happen? We called a function that adds two numbers together, and instead of getting 4, we get 12?

The reason is the subroutine `MAGIC`: We passed it the constant `1`, and since all FORTRAN parameters are passed by reference, the assignment `I = 9` *modifies the constant 1*. In C:

```
int One = 1;
int Three = 3;
int Nine = 9;
void Magic(int *i) { *i = Nine; }
int AddUp(int *i, int *j) { return *i + *j; }
void main()
{
 Magic(&One);
 printf("According to the computer, 3 + 1 is %d\n",
        AddUp(&Three, &One));
}
```

Since `Magic` modified the constant `One`, any further use of the constant 1 ends up using the value 9! (According to the FORTRAN standard, modifying a constant results in undefined behavior.)

Okay, back to calling conventions. Other significant differences between C and FORTRAN: In FORTRAN, array indices begin at 1, not 0, and arrays are stored in column-major order rather than row-major as in C.

`COMPLEX` variables in FORTRAN are stored as two floating point numbers (corresponding to the real and imaginary components).

Functions which return `COMPLEX` or `CHARACTER*(*)` are internally rewritten as subroutines where the location to store the return value is passed as a hidden first parameter. (This is analogous to how C returns large structures.)

The final commonly-encountered weirdness of FORTRAN is that `CHARACTER*n` data types (which are used to hold strings) are passed as *two* parameters: The address of the character buffer, followed by the size of the buffer (n). Note that FORTRAN `CHARACTER*n` variables are fixed-length; if you assign a string shorter than the buffer, it is padded with spaces. There is no null terminator.

Anyway, I sort of got carried away with the FORTRAN calling convention. It's definitely more complicated than just sticking `__fortran` in front of your function. But at least the `__fortran` keyword takes care of the part that can't be expressed in C. The rest you can manage on your own.

Raymond Chen

Follow