IBM XL Fortran for AIX

The Fortran 2008 BLOCK construct

By: Rafik Zurob

Level: Introductory

May 2012

Contents

Before you start

About this Tutorial

This demo demonstrates the use of the Fortran 2008 BLOCK construct for limiting the scope, and sometimes volatility, of local variables.

Objectives

- Use the IBM XL Fortran Compiler for AIX to compile Fortran source containing the Fortran 2008 BLOCK construct.
- Total time: 20 minutes

Prerequisites

- Basic Unix skills
- Basic Source code compile and build experience

System Requirements

http://www.ibm.com/software/awdtools/fortran/xlfortran/aix/sysreq

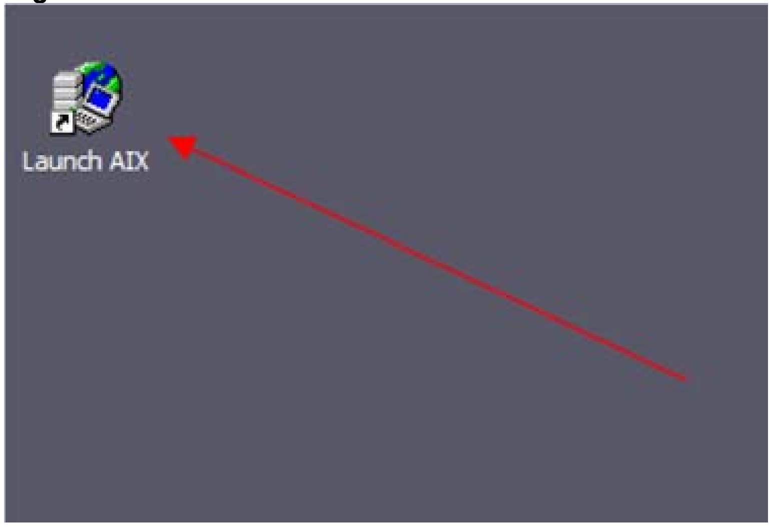
Glossary

IBM XL Fortran Compiler: The IBM® XL Fortran compiler offers advanced compiler and optimization technologies and is built on a common code base for easier porting of your applications between platforms. It complies with the latest Fortran international standards and industry specifications and supports a large array of common language features.

Getting Started

Start the Terminal Emulator to AIX System

Figure 1: Get Started

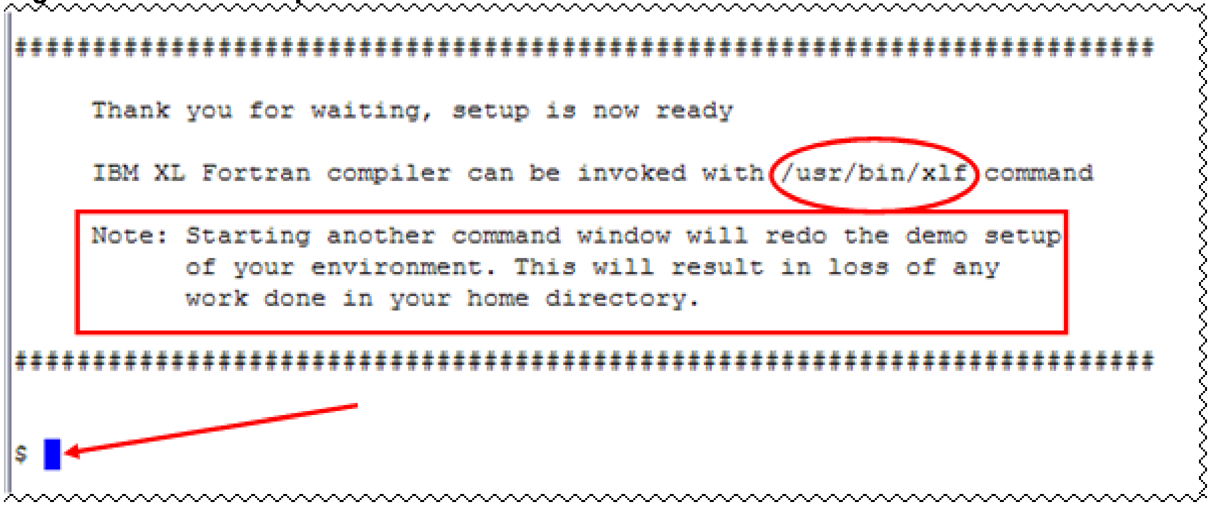


Double click the "Launch AIX" icon on the desktop (See Figure 1) to start the character terminal to AIX system.

Get Started with IBM XL Fortran Compiler

A successful login will result with the user presented with a menu of the demos hosted on the server. Type 16 and press Enter to select the "XL Fortran detecting uninitialized variables" demo.

Figure 2: Demo Prepared



```
##########################################################################

     Thank you for waiting, setup is now ready

     IBM XL Fortran compiler can be invoked with /usr/bin/xlf command

     Note: Starting another command window will redo the demo setup
           of your environment. This will result in loss of any
           work done in your home directory.

##########################################################################

$
```

On the terminal window you will see important information and the directory path to the compiler invocation command (See Figure 2 Demo Prepared *oval red*).

Note: Starting another command window will start the demonstration setup of your environment. This will result in loss of any work done in your home directory. This will impact any progress you have made on demo steps going forward.

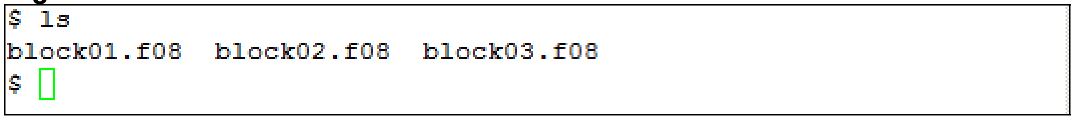
This demo does not require more than one terminal window. However, if you prefer more than one terminal window then you may open them before going forward.

The terminal window is now ready for commands. Your home directory contains the source code for three Fortran programs. Type the ls command to see the directory content (See Figure 3 Contents).

Command:

```
ls
```

Figure 3: Contents



```
$ ls
block01.f08  block02.f08  block03.f08
$ 
```

The Fortran programs are written in Fortran 2008.

Three F08 programs are provided here:

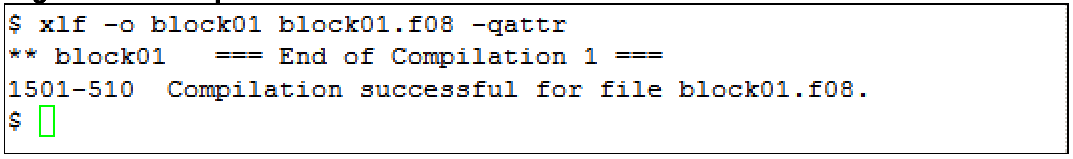
block01.f08	Simple usage of the BLOCK construct
block02.f08	BLOCK construct containing a specification expression
block03.f08	Using the BLOCK construct to reduce the performance impact of VOLATILE

Steps:

1. Compile and run block01.f08 program. (See Figure 4)

   Command:
   ```
   xlf –o block01 block01.f08 -qattr
   ```

   Figure 4: Compile block01.f08

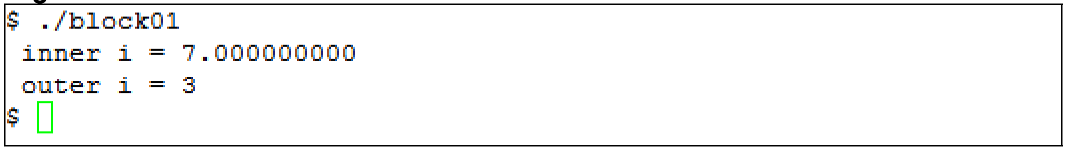
   

   ```
   $ xlf -o block01 block01.f08 -qattr
   ** block01   === End of Compilation 1 ===
   1501-510  Compilation successful for file block01.f08.
   $
   ```

   Run block01 program (See Figure 5 Run block01).

   Command:
   ```
   ./block01
   ```

   Figure 5: Run block01

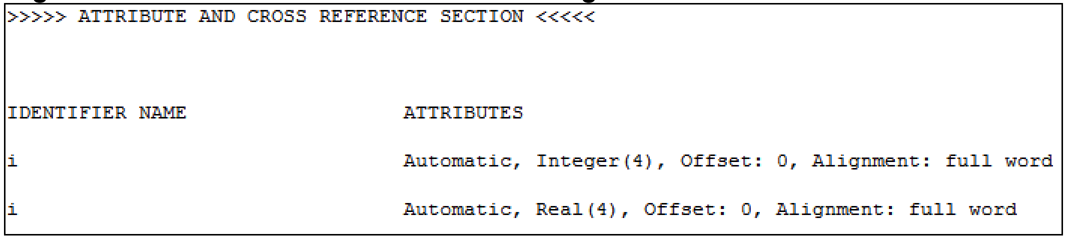
   

   ```
   $ ./block01
    inner i = 7.000000000
    outer i = 3
   $
   ```

   The output above illustrates that the value of local variable *i* in program block01 is not affected by the declaration of BLOCK variable *i*.

   Since we compiled block01.f08 above with the -qattr option, a block01.lst listing file was created. Looking into the attribute section of the listing file, you can see that we have two different *i* variables. (See Figure 6 for the attribute section of the listing file.)

   Figure 6: Attribute section of the listing file shows two variables called *i*

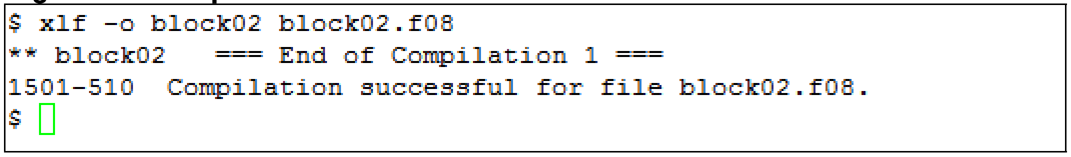
   ```
   >>>>> ATTRIBUTE AND CROSS REFERENCE SECTION <<<<<


   IDENTIFIER NAME                ATTRIBUTES

   i                              Automatic, Integer(4), Offset: 0, Alignment: full word

   i                              Automatic, Real(4), Offset: 0, Alignment: full word
   ```

2. Compile and run block02.f08. (See Figure 7)

   This program demonstrates two things:
   - BLOCK variables can depend on specification expressions. The upper bound of array *x* is not known at compile time. On entry to the block, *num_entries* will be evaluated and used for the declaration of *x*. This is only done on entry to the block, so changing *num_entries* in the block does not affect the upper bound of *x*.
   - You can give a name to your BLOCK construct. Furthermore, you can use this construct name to exit the BLOCK construct early.

   Compile block02.f08 program.

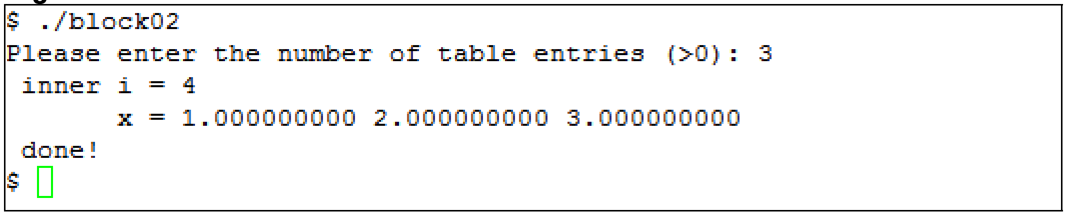
   Command:
   ```
   xlf -o block02 block02.f08
   ```

   Figure 7: compile block02.f08
   
   ```
   $ xlf -o block02 block02.f08
   ** block02   === End of Compilation 1 ===
   1501-510  Compilation successful for file block02.f08.
   $ 
   ```

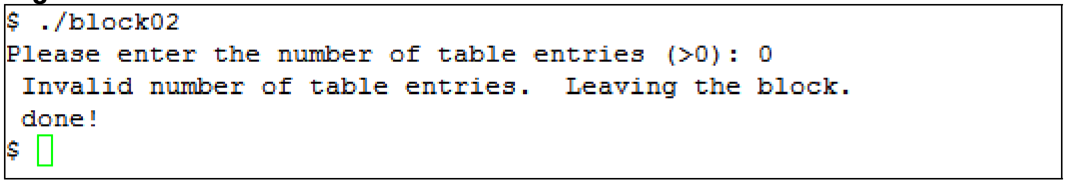
   Run block02 program (See Figure 8 Run block02).

   Command:
   ```
   ./block02
   ```

   Enter the number of entries. For example, type 3 and press enter. The program will create array *x* with upper bound 3. (See Figure 8 for the output with 0 entries)

   Figure 8: Run block02 with 3 entries
   
   ```
   $ ./block02
   Please enter the number of table entries (>0): 3
    inner i = 4
          x = 1.000000000 2.000000000 3.000000000
    done!
   $ 
   ```

   Now, run the program again with 0 entries. That is, type 0 and press enter. The program will use the EXIT statement to terminate the BLOCK early. (See Figure 9 for the output with 0 entries)

   Figure 9: Run block02 with 0 entries
   
   ```
   $ ./block02
   Please enter the number of table entries (>0): 0
    Invalid number of table entries.  Leaving the block.
    done!
   $ 
   ```

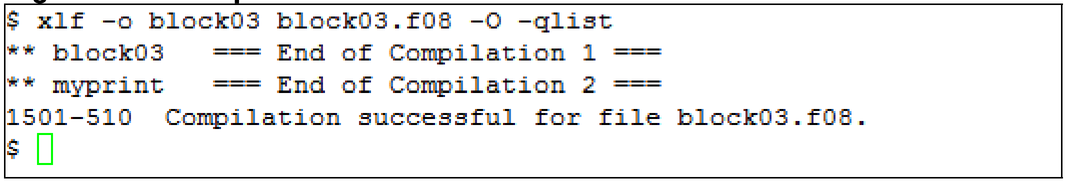
3. Compile block03.f08 and generate a listing file.

   This program has variables *x*, *y*, and *z*. The value of *x* needs to be stored into *y* and *z* every time program block02 changes *x*. Let's assume that the memory for variable *x* can be changed by a different thread in the middle part of the program, but not the top and bottom. To support this, the program uses a BLOCK construct to house the middle part of the program and gives *x* the VOLATILE attribute inside the block. (Note that we're not declaring a separate "inner" *x* as in previous examples. We're only giving *x* the VOLATILE attribute inside the block.)

   Compile block03.f08 with optimization and generate a listing file. (Figure 10)

   Command:
   ```
   xlf –o block03 block03.f08 -O -qlist
   ```

   Figure 10: Compile block03.f08



```
$ xlf -o block03 block03.f08 -O -qlist
** block03   === End of Compilation 1 ===
** myprint   === End of Compilation 2 ===
1501-510  Compilation successful for file block03.f08.
$
```

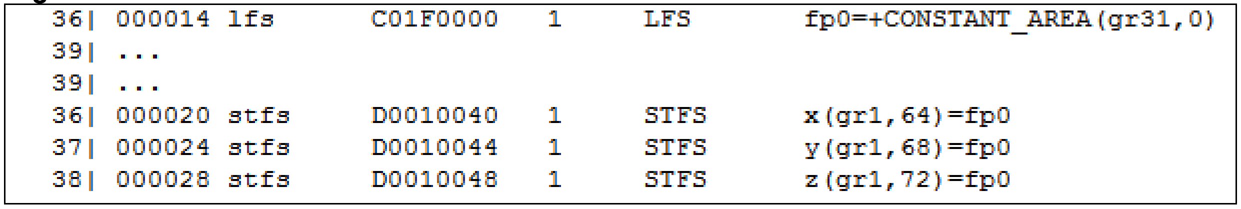
   The -O compiler option enables compiler optimization. The optimizer, when appropriate, will try to improve performance by reusing registers that already contain the value of x instead of loading x from memory every time.

   The -qlist compiler option will generate a listing file, block03.lst, containing an object section showing the pseudo-assembly instructions for the program.

   The VOLATILE attribute indicates that a variable might be referenced or modified by means other than the program. For example, the variable could be in shared memory, or accessible by multiple threads. To accommodate this, the compiler will always load the variable from memory when it is referenced and will immediately store it to memory when it's changed. This has a performance impact.

   Let's look at the object section of the listing file, specifically at the code for lines 37, 38, 46, 47, 52, and 54. (See Figure 11)

   Figure 11: Before the block: *x* is not volatile



```
   36| 000014 lfs      C01F0000    1      LFS        fp0=+CONSTANT_AREA(gr31,0)
   39| ...
   39| ...
   36| 000020 stfs     D0010040    1      STFS       x(gr1,64)=fp0
   37| 000024 stfs     D0010044    1      STFS       y(gr1,68)=fp0
   38| 000028 stfs     D0010048    1      STFS       z(gr1,72)=fp0
```

   The lfs instruction loads a real(4) value from memory and places it into a floating-point register. In this case, it's loading the value for 3.0 from the constant area and storing it into register fp1.

The stfs instruction stores a real(4) value from a floating-point register into memory. In this case, it's storing the value from register fp0 into variables *x*, *y*, and *z* in memory. Notice how fp0 was reused in lines 37 and 38 to save the cost of loading *x* (via lfs).

Figure 12: Inside the block: *x* is volatile

```
   45| 000048 stfs      D0410040   1      STFS       x(gr1,64)=fp2
   46| 00004C lfs       C0010040   1      LFS        fp0=x<auto1:21:9>(gr1,64)
   46| 000050 stfs      D0010044   1      STFS       y(gr1,68)=fp0
   47| 000054 lfs       C0210040   1      LFS        fp1=x<auto1:21:9>(gr1,64)
   47| 000058 stfs      D0210048   1      STFS       z(gr1,72)=fp1
```

Since x is given the VOLATILE attribute inside the block, *x* has to be loaded from memory every time its value is needed. In this case, the new value of *x* is in fp2 (line 45). However, to store *x* into *y* on line 46, we issue a new lfs instruction reload *x* from memory into register fp0. We then store the value from fp0 into *y*. To store *x* into *z*, we don't reuse fp2 or fp0. Instead, we reload *x* (again) from memory.

Figure 13: After the block: x is not volatile anymore

```
   52| 00006C lfs       C0010040   1      LFS        fp0=x(gr1,64)
   52| 000070 stfs      D0010044   1      STFS       y(gr1,68)=fp0
   54| 000074 stfs      D0010048   1      STFS       z(gr1,72)=fp0
```

Since *x* is not volatile outside the block, we can reuse registers containing its value. In this case, the lfs instruction is used to load *x* into register fp0. The stfs instructions are used to store the value of register fp0 into both *y* and *z*. So we were able to save an lfs.

What you have learned

In this exercise you learned how to:

- Use the IBM XL Fortran for AIX compiler to compile programs.
- Use the Fortran 2008 BLOCK construct to declare variables with a more limited scope.
- Declare BLOCK variables involving specification expressions.
- Use the Fortran 2008 BLOCK construct to reduce the performance impact of the VOLATILE attribute to just the code where the variable is volatile.

Conclusion

This concludes the tutorial on using the Fortran 2008 BLOCK construct with the IBM XL Fortran compiler.

Trademarks

Resources

Learn:

Overview of the IBM XL C/C++ and XL Fortran Compiler Family
http://www.ibm.com/support/docview.wss?uid=swg27005175

Getting Started with XL Fortran for AIX, V14.1
http://www.ibm.com/support/docview.wss?uid=swg27024216

Compiler Reference - XL Fortran for AIX, V14.1
http://www.ibm.com/support/docview.wss?uid=swg27024215

Language Reference - XL Fortran for AIX, V14.1
http://www.ibm.com/support/docview.wss?uid=swg27024218

Optimization:

Optimization and Programming Guide - XL Fortran for AIX, V14.1
http://www.ibm.com/support/docview.wss?uid=swg27024219

Code Optimization with the IBM XL Compilers
http://www.ibm.com/support/docview.wss?uid=swg27005174