**Sun Studio 12: Debugging a Program With dbx**

**Previous**: Chapter 15 Debugging C++ With dbx **Next**: Chapter 17 Debugging a Java Application With dbx

# Chapter 16 Debugging Fortran Using `dbx`

This chapter introduces `dbx` features you might use with Fortran. Sample requests to `dbx` are also included to provide you with assistance when debugging Fortran code using `dbx`.

This chapter includes the following topics:

- Debugging Fortran
- Debugging Segmentation Faults
- Locating Exceptions
- Tracing Calls
- Working With Arrays
- Showing Intrinsic Functions
- Showing Complex Expressions
- Showing Logical Operators
- Viewing Fortran 95 Derived Types
- Pointer to Fortran 95 Derived Type

## Debugging Fortran

The following tips and general concepts are provided to help you while debugging Fortran programs. For information on debugging Fortran OpenMP code with `dbx`, see Interacting With Events.

### Current Procedure and File

During a debug session, `dbx` defines a procedure and a source file as current. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, `stop at 5` sets different breakpoints, depending on which file is current.

### Uppercase Letters

If your program has uppercase letters in any identifiers, `dbx` recognizes them. You need not provide case-sensitive or case-insensitive commands, as in some earlier versions.

Fortran 95 and `dbx` must be in the same case-sensitive or case-insensitive mode:

- Compile and debug in case-insensitive mode without the `-U` option. The default value of the `dbx input_case_sensitive` environment variable is then `false` .

If the source has a variable named `LAST` , then in `dbx` , both the `print LAST` or `print last` commands work. Fortran 95 and `dbx` consider `LAST` and `last` to be the same, as requested.

- Compile and debug in case-sensitive mode using `-U` . The default value of the `dbx input_case_sensitive` environment variable is then `true` .

  If the source has a variable named `LAST` and one named `last` , then in `dbx` , `print last` works, but `print LAST` does not work. Fortran 95 and `dbx` distinguish between `LAST` and `last` , as requested.

**Note –**

File or directory names are always case-sensitive in `dbx` , even if you have set the `dbx input_case_sensitive` environment variable to `false` .

## Sample `dbx` Session

The following examples use a sample program called `my_program` .

Main program for debugging, a1.f:

```
    PARAMETER ( n=2 )
    REAL twobytwo(2,2) / 4 *-1 /
    CALL mkidentity( twobytwo, n )
    PRINT *, determinant( twobytwo )
    END
```

Subroutine for debugging, a2.f:

```
    SUBROUTINE mkidentity ( array, m )
    REAL array(m,m)
    DO 90 i = 1, m
        DO 20 j = 1, m
            IF ( i .EQ. j ) THEN
            array(i,j) = 1.
            ELSE
            array(i,j) = 0.
            END IF
20        CONTINUE
90    CONTINUE
    RETURN
    END
```

Function for debugging, `a3.f` :

```
    REAL FUNCTION determinant ( a )
    REAL a(2,2)
    determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
    RETURN
    END
```

### Running the Sample `dbx` Session

1. Compile and link with the `- g` option.

   You can do this in one or two steps.

   Compile and link in one step, with `-g` :

```
demo% f95 -o my_program -g a1.f a2.f a3.f
```

Or, compile and link in separate steps:

```
demo% f95  -c -g a1.f a2.f a3.f
demo% f95  -o my_program a1.o a2.o a3.o
```

2. Start `dbx` on the executable named `my_program` .

```
demo%  dbx my_program
Reading symbolic information…
```

3. Set a simple breakpoint by typing `stop  in` *subnam*, where *subnam* names a subroutine, function, or block data subprogram.

   To stop at the first executable statement in a main program.

```
(dbx)  stop in MAIN
(2) stop in MAIN
```

   Although `MAIN` must be all uppercase, *subnam* can be uppercase or lowercase.

4. Type the `run` command, which runs the program in the executable files named when you started `dbx` .

```
(dbx)  run
Running: my_program
stopped in MAIN at line 3 in file "a1.f"
    3         call mkidentity( twobytwo, n )
```

   When the breakpoint is reached, `dbx` displays a message showing where it stopped—in this case, at line 3 of the `a1.f` file.

5. To print a value, type the `print` command.

   Print value of `n` :

```
(dbx)  print n
n = 2
```

   Print the matrix `twobytwo` ; the format might vary:

```
(dbx)  print twobytwo
twobytwo =
   (1,1)       -1.0
   (2,1)       -1.0
   (1,2)       -1.0
   (2,2)       -1.0
```

   Print the matrix `array` :

```
(dbx)  print array
dbx: "array" is not defined in the current scope
(dbx)
```

   The print fails because `array` is not defined here—only in `mkidentity` .

6. To advance execution to the next line, type the `next` command.

   Advance execution to the next line:

```
(dbx)  next
```

```
stopped in MAIN at line 4 in file "a1.f"
    4             print *, determinant( twobytwo )
(dbx)  print twobytwo
twobytwo =
    (1,1)       1.0
    (2,1)       0.0
    (1,2)       0.0
    (2,2)       1.0
(dbx)  quit
demo%
```

The `next` command executes the current source line and stops at the next line. It counts subprogram calls as single statements.

Compare the `next` command with the `step` command. The `step` command executes the next source line or the next step into a subprogram. If the next executable source statement is a subroutine or function call, then:

- The `step` command sets a breakpoint at the first source statement of the subprogram.
- The `next` command sets the breakpoint at the first source statement after the call, but still in the calling program.

7. To quit `dbx`, type the `quit` command.

```
(dbx) quit
demo%
```

## Debugging Segmentation Faults

If a program gets a segmentation fault ( `SIGSEGV` ), it references a memory address outside of the memory available to it.

The most frequent causes for a segmentation fault are:

- An array index is outside the declared range.
- The name of an array index is misspelled.
- The calling routine has a `REAL` argument, which the called routine has as `INTEGER` .
- An array index is miscalculated.
- The calling routine has fewer arguments than required.
- A pointer is used before it has been defined.

### Using `dbx` to Locate Problems

Use `dbx` to find the source code line where a segmentation fault has occurred.

Use a program to generate a segmentation fault:

```
demo%  cat WhereSEGV.f
    INTEGER a(5)
    j = 2000000
    DO 9 i = 1,5
        a(j) = (i * 10)
9    CONTINUE
    PRINT *, a
    END
demo%
```

Use `dbx` to find the line number of a `dbx` segmentation fault:

```
demo%  f95 -g -silent WhereSEGV.f
demo%  a.out
Segmentation fault
```

```
demo%  dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx)  run
Running: a.out
signal SEGV (no mapping at the fault address)
    in MAIN at line 4 in file "WhereSEGV.f"
    4                   a(j) = (i * 10)
(dbx)
```

## Locating Exceptions

If a program gets an exception, there are many possible causes. One approach to locating the problem is to find the line number in the source program where the exception occurred, and then look for clues there.

Compiling with `-ftrap=common` forces trapping on all common exceptions.

To find where an exception occurred:

```
demo%  cat wh.f
                call joe(r, s)
                print *, r/s
                end
                subroutine joe(r,s)
                r = 12.
                s = 0.
                return
                end
demo%  f95 -g -o wh -ftrap=common wh.f
demo%  dbx wh
Reading symbolic information for wh
(dbx)  catch FPE
(dbx)  run
Running: wh
(process id 17970)
signal FPE (floating point divide by zero) in MAIN at line 2 in file “wh.f”
   2                     print *, r/s
(dbx)
```

## Tracing Calls

Sometimes a program stops with a core dump, and you need to know the sequence of calls that led it there. This sequence is called a **stack trace**.

The `whe re` command shows where in the program flow execution stopped and how execution reached this point—a **stack trace** of the called routines.

`ShowTrace.f` is a program contrived to get a core dump a few levels deep in the call sequence—to show a stack trace.

```
Note the reverse order:
demo% f77  -silent -g ShowTrace.f
demo%  a.out
MAIN called calc, calc called calcb.
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174%  dbx a.out
Execution stopped, line 23
Reading symbolic information for a.out
...
(dbx)  run
calcB called from calc, line 9
Running: a.out
(process id 1089)
calc called from MAIN, line 3
signal SEGV (no mapping at the fault address) in calcb at line 23 in file "ShowTrace.f"
```

```
   23                  v(j) = (i * 10)
(dbx)  where -V
=>[1] calcb(v = ARRAY , m = 2), line 23 in "ShowTrace.f"
  [2] calc(a = ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f"
  [3] MAIN(), line 3 in "ShowTrace.f"
(dbx)
Show the sequence of calls, starting at where the execution stopped:
```

## Working With Arrays

`dbx` recognizes arrays and can print them.

```
demo%  dbx a.out
Reading symbolic information…
(dbx)  list 1,25
    1          DIMENSION IARR(4,4)
    2          DO 90 I = 1,4
    3                  DO 20 J = 1,4
    4                          IARR(I,J) = (I*10) + J
    5   20             CONTINUE
    6   90     CONTINUE
    7          END
(dbx)   stop at 7
(1) stop at "Arraysdbx.f":7
(dbx)  run
Running: a.out
stopped in MAIN at line 7 in file "Arraysdbx.f"
    7          END
(dbx)  print IARR
iarr =
    (1,1) 11
    (2,1) 21
    (3,1) 31
    (4,1) 41
    (1,2) 12
    (2,2) 22
    (3,2) 32
    (4,2) 42
    (1,3) 13
    (2,3) 23
    (3,3) 33
    (4,3) 43
    (1,4) 14
    (2,4) 24
    (3,4) 34
    (4,4) 44
(dbx)  print IARR(2,3)
    iarr(2, 3) = 23  - Order of user-specified subscripts ok
(dbx)  quit
```

For information on array slicing in Fortran, see Array Slicing Syntax for Fortran.

## Fortran 95 Allocatable Arrays

The following example shows how to work with allocated arrays in `dbx` .

```
demo%  f95 -g Alloc.f95
  demo%  dbx a.out
  (dbx)  list 1,99
      1   PROGRAM TestAllocate
      2   INTEGER n, status
      3   INTEGER, ALLOCATABLE :: buffer(:)
      4           PRINT *, 'Size?'
      5           READ *, n
      6           ALLOCATE( buffer(n), STAT=status )
      7           IF ( status /= 0 ) STOP 'cannot allocate buffer'
      8           buffer(n) = n
      9           PRINT *, buffer(n)
     10           DEALLOCATE( buffer, STAT=status)
     11   END
(dbx)  stop at 6
 (2) stop at "alloc.f95":6
 (dbx)  stop at 9
 (3) stop at "alloc.f95":9
 (dbx)  run
 Running: a.out
 (process id 10749)
  Size?
  1000
Unknown size at line 6
 stopped in main at line 6 in file "alloc.f95"
     6           ALLOCATE( buffer(n), STAT=status )
 (dbx)  whatis buffer
 integer*4 , allocatable::buffer(:)
 (dbx)  next
 continuing
 stopped in main at line 7 in file "alloc.f95"
     7           IF ( status /= 0 ) STOP 'cannot allocate buffer'
 (dbx)  whatis buffer
 integer*4 buffer(1:1000)
Known size at line 9
 (dbx)  cont
 stopped in main at line 9 in file "alloc.f95"
     9           PRINT *, buffer(n)
 (dbx)  print n
buffer(1000) holds 1000
 n = 1000
 (dbx)  print buffer(n)
 buffer(n) = 1000
```

## Showing Intrinsic Functions

dbx recognizes Fortran intrinsic functions (SPARC™ platforms and x86 platforms only).

To show an intrinsic function in dbx , type:

```
demo%  cat ShowIntrinsic.f
    INTEGER i
    i = -2
    END
(dbx)  stop in MAIN
(2) stop in MAIN
(dbx)  run
Running: shi
(process id 18019)
stopped in MAIN at line 2 in file "shi.f"
    2              i = -2
(dbx)  whatis abs
Generic intrinsic function: "abs"
(dbx)  print i
i = 0
(dbx)  step
stopped in MAIN at line 3 in file "shi.f"
    3              end
(dbx)  print i
i = -2
(dbx)  print abs(1)
abs(i) = 2
(dbx)
```

## Showing Complex Expressions

`dbx` also recognizes Fortran complex expressions.

To show a complex expression in `dbx`, type:

```
demo% cat ShowComplex.f
   COMPLEX z
    z = ( 2.0, 3.0 )
    END
demo% f95 -g ShowComplex.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10953)
stopped in MAIN at line 2 in file "ShowComplex.f"
    2       z = ( 2.0, 3.0 )
(dbx) whatis z
complex*8  z
(dbx) print z
z = (0.0,0.0)
(dbx) next
stopped in MAIN at line 3 in file "ShowComplex.f"
    3       END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0,1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
demo%
```

## Showing Interval Expressions

To show an interval expression in `dbx`, type:

```
demo% cat ShowInterval.f95
   INTERVAL v
   v = [ 37.1, 38.6 ]
   END
demo% f95 -g -xia ShowInterval.f95
demo% dbx a.out
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: a.out
(process id 5217)
stopped in MAIN at line 2 in file "ShowInterval.f95"
    2      v = [ 37.1, 38.6 ]
(dbx) whatis v
INTERVAL*16  v
(dbx) print v
v = [0.0,0.0]
(dbx) next
stopped in MAIN at line 3 in file "ShowInterval.f95"
    3      END
(dbx) print v
v = [37.1,38.6]
(dbx) print v+[0.99,1.01]
v+[0.99,1.01] = [38.09,39.61]
(dbx) quit
demo%
```

**Note –**

Interval expressions are supported only for programs compiled to run on SPARC based platforms, with `-xarch=` { `sse` | `sse2` } to run on Solaris x86 SSE/SSE2 Pentium 4-compatible platforms, or with `-xarch=amd64` to run on x64 platforms.

## Showing Logical Operators

dbx can locate Fortran logical operators and print them.

To show logical operators in dbx, type:

```
demo%  cat ShowLogical.f
        LOGICAL a, b, y, z
        a = .true.
        b = .false.
        y = .true.
        z = .false.
        END
demo%  f95 -g ShowLogical.f
demo%  dbx a.out
(dbx)  list 1,9
    1           LOGICAL a, b, y, z
    2           a = .true.
    3           b = .false.
    4           y = .true.
    5           z = .false.
    6           END
(dbx)  stop at 5
(2) stop at "ShowLogical.f":5
(dbx)  run
Running: a.out
(process id 15394)
stopped in MAIN at line 5 in file "ShowLogical.f"
    5           z = .false.
(dbx)  whatis y
logical*4 y
(dbx)  print a .or. y
a.OR.y = true
(dbx)  assign z = a .or. y
(dbx)  print z
z = true
(dbx)  quit
demo%
```

## Viewing Fortran 95 Derived Types

You can show structures—Fortran 95 derived types—with dbx.

```
demo%  f95 -g DebStruc.f95
demo%  dbx a.out
(dbx)  list 1,99
    1   PROGRAM Struct ! Debug a Structure
    2      TYPE product
    3         INTEGER        id
    4         CHARACTER*16   name
    5         CHARACTER*8    model
    6         REAL           cost
    7 REAL price
    8      END TYPE product
    9
   10      TYPE(product) :: prod1
   11
   12      prod1%id = 82
   13      prod1%name = "Coffee Cup"
   14      prod1%model = "XL"
   15      prod1%cost = 24.0
   16      prod1%price = 104.0
   17      WRITE ( *, * ) prod1%name
   18   END
(dbx)  stop at 17
(2) stop at "Struct.f95":17
```

```
(dbx)  run
Running: a.out
(process id 12326)
stopped in main at line 17 in file "Struct.f95"
   17      WRITE ( *, * ) prod1%name
(dbx)  whatis prod1
product prod1
(dbx)  whatis -t product
type product
    integer*4 id
    character*16 name
    character*8 model
    real*4 cost
    real*4 price
end type product
(dbx)  n
(dbx)  print prod1
    prod1 = (
    id    = 82
    name = 'Coffee Cup'
    model = 'XL'
    cost = 24.0
    price = 104.0
)
```

## Pointer to Fortran 95 Derived Type

You can show structures—Fortran 95 derived types—and pointers with `dbx`.

```
demo%  f95 -o debstr -g DebStruc.f95
 demo%  dbx debstr
 (dbx)  stop in main
 (2) stop in main
 (dbx)  list 1,99
     1   PROGRAM DebStruPtr! Debug structures & pointers
Declare a derived type.
     2      TYPE product
     3         INTEGER        id
     4         CHARACTER*16   name
     5         CHARACTER*8    model
     6         REAL           cost
     7         REAL           price
     8      END TYPE product
     9
Declare prod1  and prod2 targets.
    10      TYPE(product), TARGET :: prod1, prod2
Declare curr and prior pointers.
    11      TYPE(product), POINTER :: curr, prior
    12
Make curr point to prod2.
    13      curr => prod2
Make prior point to prod1.
    14      prior => prod1
Initialize prior.
    15      prior%id = 82
    16      prior%name = "Coffee Cup"
    17      prior%model = "XL"
    18      prior%cost = 24.0
    19      prior%price = 104.0
Set curr to prior.
    20      curr = prior
Print name from curr and prior.
    21      WRITE ( *, * ) curr%name, " ", prior%name
    22   END PROGRAM DebStruPtr
 (dbx)  stop at 21
 (1) stop at "DebStruc.f95":21
 (dbx)  run
 Running: debstr
(process id 10972)
```

```
stopped in main at line 21 in file "DebStruc.f95"
   21      WRITE ( *, * ) curr%name, " ", prior%name
(dbx)  print prod1
 prod1 = (
    id = 82
    name = "Coffee Cup"
    model = "XL"
    cost = 24.0
    price = 104.0
)
```

Above, `dbx` displays all fields of the derived type, including field names.

You can use structures—inquire about an item of an Fortran 95 derived type.

```
Ask about the variable
(dbx)  whatis prod1
 product prod1
Ask about the type (-t)
 (dbx)  whatis -t product
 type product
    integer*4 id
    character*16 name
    character*8 model
    real cost
    real price
 end type product
```

To print a pointer, type:

```
dbx displays the contents of a pointer, which is an address. This address can be different with every run.
(dbx)  print prior
 prior = (
     id    = 82
     name = 'Coffee Cup'
     model = 'XL'
     cost = 24.0
     price = 104.0
 )
```

**Previous**: Chapter 15 Debugging C++ With dbx **Next**: Chapter 17 Debugging a Java Application With dbx