How to Debug Fortran Code with GDB

Jonathan Laver (engineer-in-training) and Mark Williamson (senior software engineer at Undo) write:

We are used to using GDB for debugging C and C++ but it can also be used to debug other languages including Fortran, D, Go and Ada. Debug info formats like DWARF, along with some language-specific extensions, allow GDB to support most of the commonly-used compiled languages. Today we'll see a simple example of how we can debug Fortran programs, using features provided by GDB and gfortran.

Our program

Here we have some Fortran code. It's supposed to calculate the ratios between successive integers but it contains a bug:

```
program bugs
   implicit none
   real last
   real c(10)
   integer p

   ! Initialise c with successive integer values.
   do p=1,10
       c(p)=p
    enddo

    ! Calculate and print ratios of successive integers.
    last = 0.0
    do p=1,10
        call divide(last, c(p))
        last = c(p)
```

```
    enddo
end program bugs

subroutine divide(d,e)
    implicit none
    real d,e
    print *,e/d
end subroutine divid
```

If we compile and run this code, we see the output contains unexpected text – the Infinity printed on the first line:

```
  $&gt; gfortran bugs.f90
  $&gt; ./a.out
           Infinity
     2.00000000
     1.50000000
     1.33333337
     1.25000000
     1.20000005
     1.16666663
     1.14285719
     1.12500000
     1.11111116
```

Building for debug

To diagnose this fault, we're going to run the program under GDB. As with other compilers from the gcc family, we need to supply the -g flag to generate debug information.

The Infinity output suggests that we have an error in our floating point maths, most likely a divide by zero. The gfortran debug options can help us here. The -ffpe-trap compilation flag enables exception traps for the various floating point errors; we can catch these with GDB to find the

exact cause of our bug.

Lets build our executable with these new options:

```
$> gfortran -g -ffpe-trap=zero,invalid,overflow,underflow bugs.f90
```

Debugging in GDB

Just as for a C program, we start our Fortran executable under control of the debugger. To start debugging our program we invoke gdb PROGRAM. To stop execution on the first line of our program we use b MAIN__. The reason we need to use MAIN__ instead of main (as we would usually use with C) is that main actually runs some startup code to set up the enviroment.

Lets start our debug session:

```
$&gt; gdb ./a.out
[ ... GDB start-up messages ... ]
(gdb) break MAIN__
Breakpoint 1 at 0x4008c4: file bugs.f90, line 8.
(gdb) run
Starting program: /home/blog_posts/a.out

Breakpoint 1, bugs () at bugs.f90:8
8         do p=1,10
```

Now we can step through each line of Fortran source code by pressing n:

```
(gdb) n
9             c(p)=p
(gdb) n
8         do p=1,10
(gdb) n
```

```
9             c(p)=p
```

This is just as we'd expect from our C / C++ debugging experience – GDB is using the debug information from gfortran to step through lines of source code, so we can see how the state is changing.

We could just do that until the error occurs – but it would be nice to jump to the exact moment of the floating point error we're expecting. Now we've recompiled to trap on floating point exceptions we will receive a SIGFPE if a floating point error happens. By default, GDB will stop when it sees signals that indicate errors – in C code we often we see this when a SIGSEGV occurs due to a pointer-related bug.

If we simply continue with execution, GDB will stop the program when a floating-point error occurs:

```
(gdb) cont
Continuing.

Program received signal SIGFPE, Arithmetic exception.
0x0000000000400887 in divide (d=0, e=1) at bugs.f90:24
24      print *,e/d
```

The debugger has now stopped on a floating point arithmetic exception – this is likely to be the source of our maths error. We can use the debugger to find out where we are:

```
(gdb) bt
#0  0x0000000000400887 in divide (d=0, e=1) at bugs.f90:24
#1  0x0000000000400932 in bugs () at bugs.f90:15
#2  0x0000000000400996 in main (argc=1, argv=0x7fffffffe811) at bugs.f90:
#3  0x00000034ff821d65 in __libc_start_main ([...]) at libc-start.c:285
#4  0x0000000000400759 in _start ()
```

(arguments to __libc_start_main are omitted for brevity)

We're in our divide function, called from our bugs routine. In an interactive session, we might use the list command to check on the surrounding code. For the purposes of this example, lets just inspect the values of the variables for mathematical errors:

```
(gdb) p d
$1 = 0
(gdb) p e
$2 = 1
(gdb) p e/d
$3 = inf
```

Given the values of d and e, it looks like we are dividing by zero by mistake – GDB confirms that e/d gives the result inf, or infinity. The order of the variables in the division (see line 23) is wrong – we should be dividing d by e and not the other way around. If we fix this bug and re-run then we'll see the following output:

```
$&gt; ./a.out
 0.00000000
0.500000000
0.666666687
0.750000000
0.800000012
0.833333313
0.857142866
0.875000000
0.888888896
0.899999976
```

We've fixed our unwanted Infinity message! We're now seeing the ratios of successive integers trending progressively closer to 1.0, as we would

expect.

Summary

This was a quick intro into how to debug a less common (though still very popular) programming language. By compiling the programs with debug info, GDB is able to work correctly and, apart from a few differences in how to breakpoint at startup, the process is largely the same as we're used to. By switching on traps for floating point exceptions we were able to stop the program just as the bug occurred. Once there, we could inspect the source code, show the state of program variables and try out arithmetic expressions in order to understand our bug.

In the future I plan to look at how to use GDB with Rust, Ada, D and Go and to explore the additional error checking capabilities provided by each language.