

Debugging mit ddd (Data Display Debugger)

1 Testprogramm installieren und ausführen

Laden Sie sich das Fortran-Programm `sample.f90` und das Makefile herunter und speichern Sie sie in einem Verzeichnis. Das Programm sortiert ganze Zahlen der Größe nach.

Geben Sie in einer Shell in diesem Verzeichnis den Befehl "make" ein, um aus dem Fortran-Programm eine ausführbares Programm `sample.x` zu erzeugen.

```
$ make
```

Führen Sie das Programm mit den Argumenten 4, 3 und 7 aus.

```
$ ./sample.x 4 3 7
```

Die Ausgabe "3 4 7" ist korrekt.

Führen Sie das Programm nun mit den Argumenten 4 und 1 aus.

```
./sample.x 4 1
```

Die Ausgabe "0 1" ist falsch. Korrekt wäre '1 4'.

Das Programm produziert also für gewisse Eingabedaten falsche Ergebnisse.

2 Installation von ddd

```
$ su  
[Passwort eingeben]  
$ apt-get install ddd  
$ exit
```

2.1 Das Programm ausführen

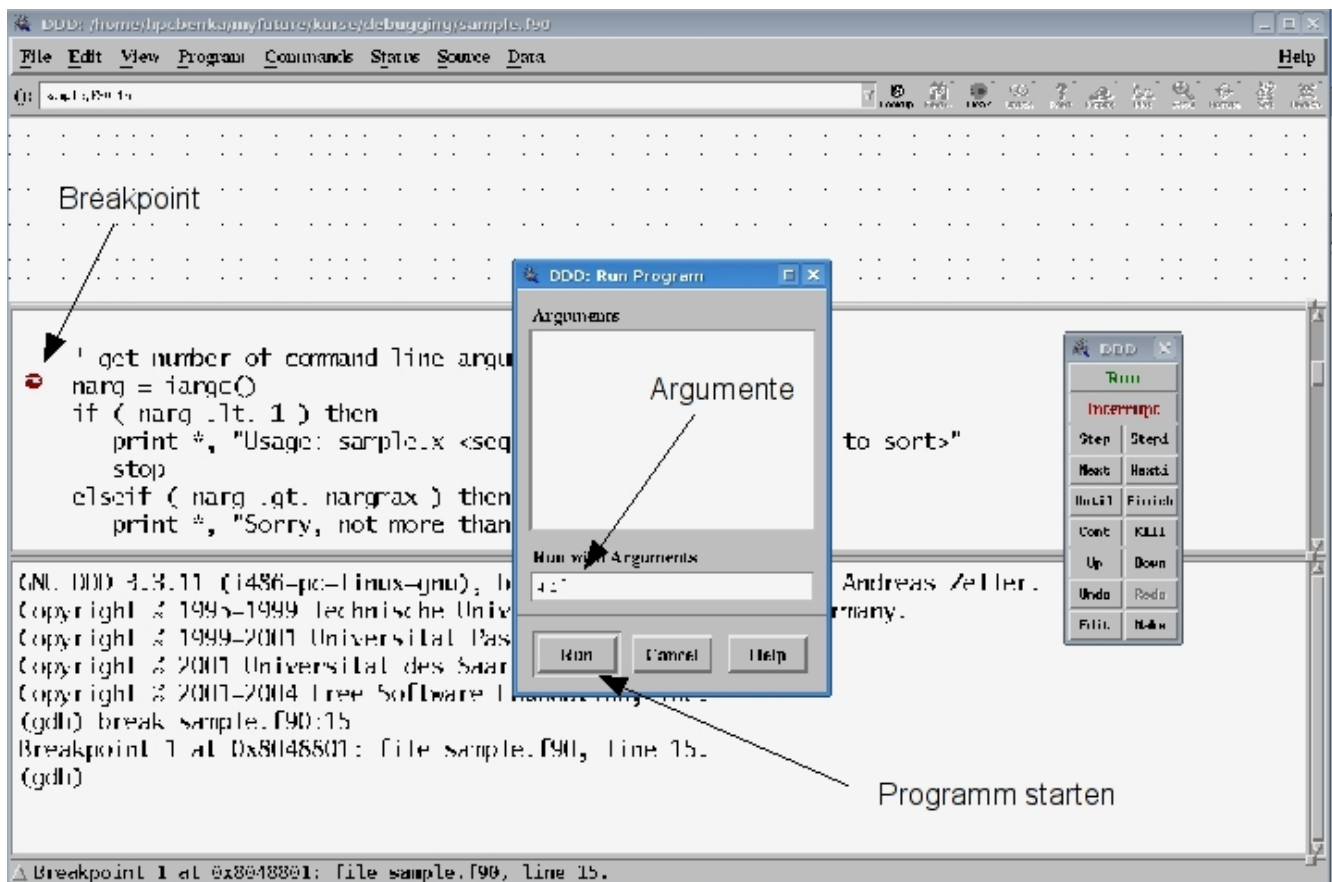
Als erstes müssen wir einen Breakpoint setzen, sonst würde das Programm einfach bis zum Ende durchlaufen, sobald wir es starten.

Klicken Sie dazu auf den freien Platz links neben der Zeile "`narg = iargc()`" mit der rechten Maustaste und wählen aus dem Kontextmenü 'Set breakpoint' aus. Ein Stop-Schild wird neben der Zeile angezeigt. Im Kommando-Fenster sehen Sie, die Ausgabe

```
(gdb) break sample.f90:18  
Breakpoint 1 at 0x8049cef: file sample.f90, line 15
```

Sie hätten also genauso auf der Kommandozeile 'break sample.f90:15' oder einfacher 'break 15' eingeben können, um einen Breakpoint in Zeile 15 zu setzen.

Als nächstes führen wir das Programm aus, um sein Verhalten zu untersuchen. Dies geht über das Menü Program -> Run. Im Fenster "Run Program" geben Sie bitte bei "Run with Arguments" als Argumente "4 1" an und klicken dann auf "Run".

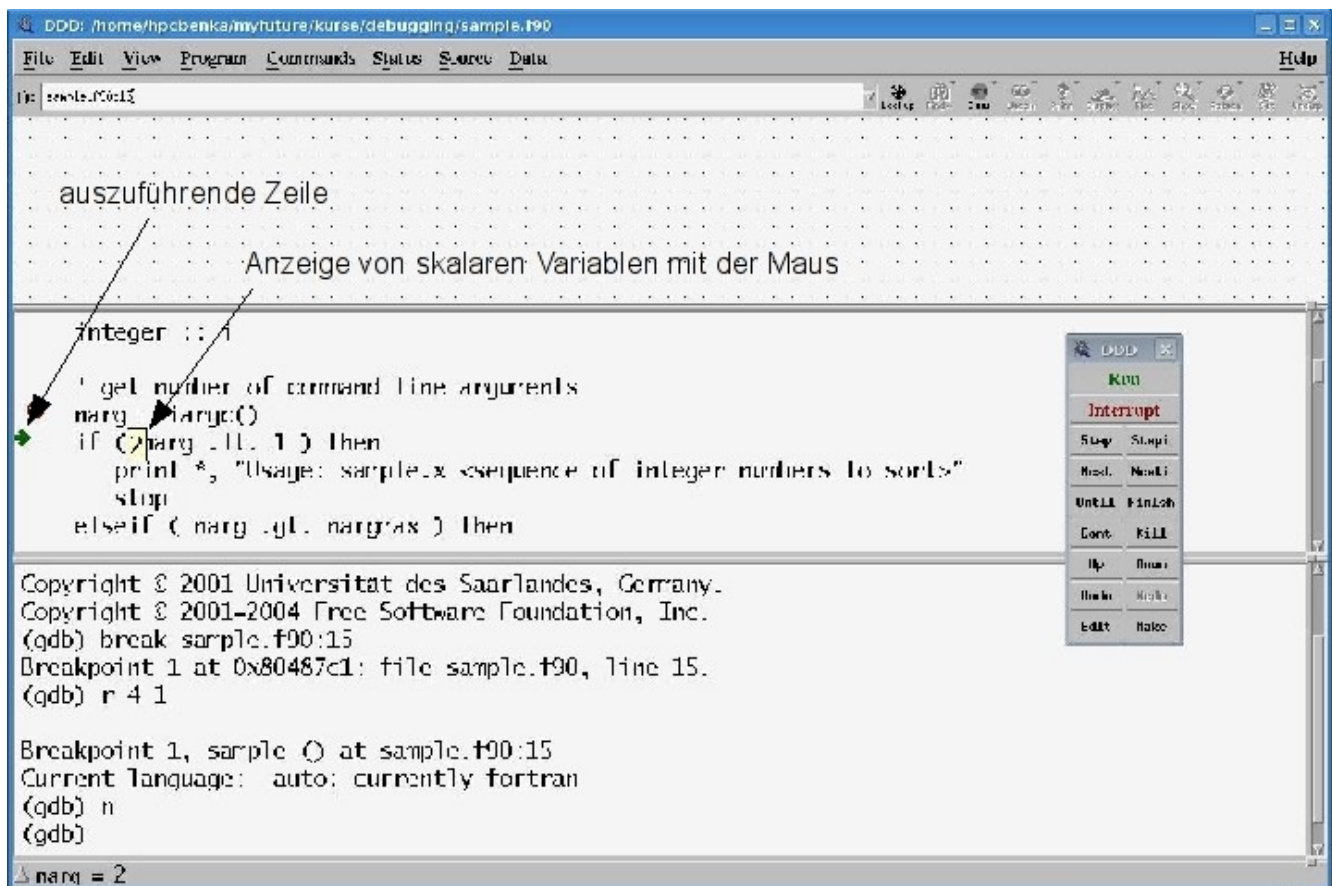


GDB führt jetzt `sample.x` aus. Die Ausführung stoppt nach einem kurzen Moment und neben dem Stop-Zeichen erscheint ein kleiner grüner Pfeil. Dieser zeigt an, welche Zeile als nächstes ausgeführt wird. GDB meldet

```
Breakpoint 1, sample () at sample.f90:15
(gdb) _
```

Jetzt können wir die Werte von Variablen anzeigen lassen. Wenn wir eine skalare Variable wie z.B. die Anzahl der Argumente `narg` untersuchen wollen, können wir mit dem Mauszeiger auf die Variable zeigen und nach kurzer Zeit wird der Wert der Variable in einem kleinen Fenster angezeigt. Nachdem `narg` noch kein Wert zugewiesen wurde, steht dort Müll drin.

Um diese Programmzeile auszuführen, klicken Sie bitte in der Toolbar auf "Next" oder geben "next" oder einfach "n" in der Kommandozeile ein. Der grüne Zeiger springt eine Zeile weiter. Wenn Sie jetzt noch einmal mit dem Mauszeiger auf `narg` zeigen, sehen Sie, dass der Wert sich geändert hat und `narg` jetzt mit 2 initialisiert wurde.



Führen Sie das Programm weiter bis zur Zeile 34 “read(arg, fmt='(i5)') a(i)” aus.

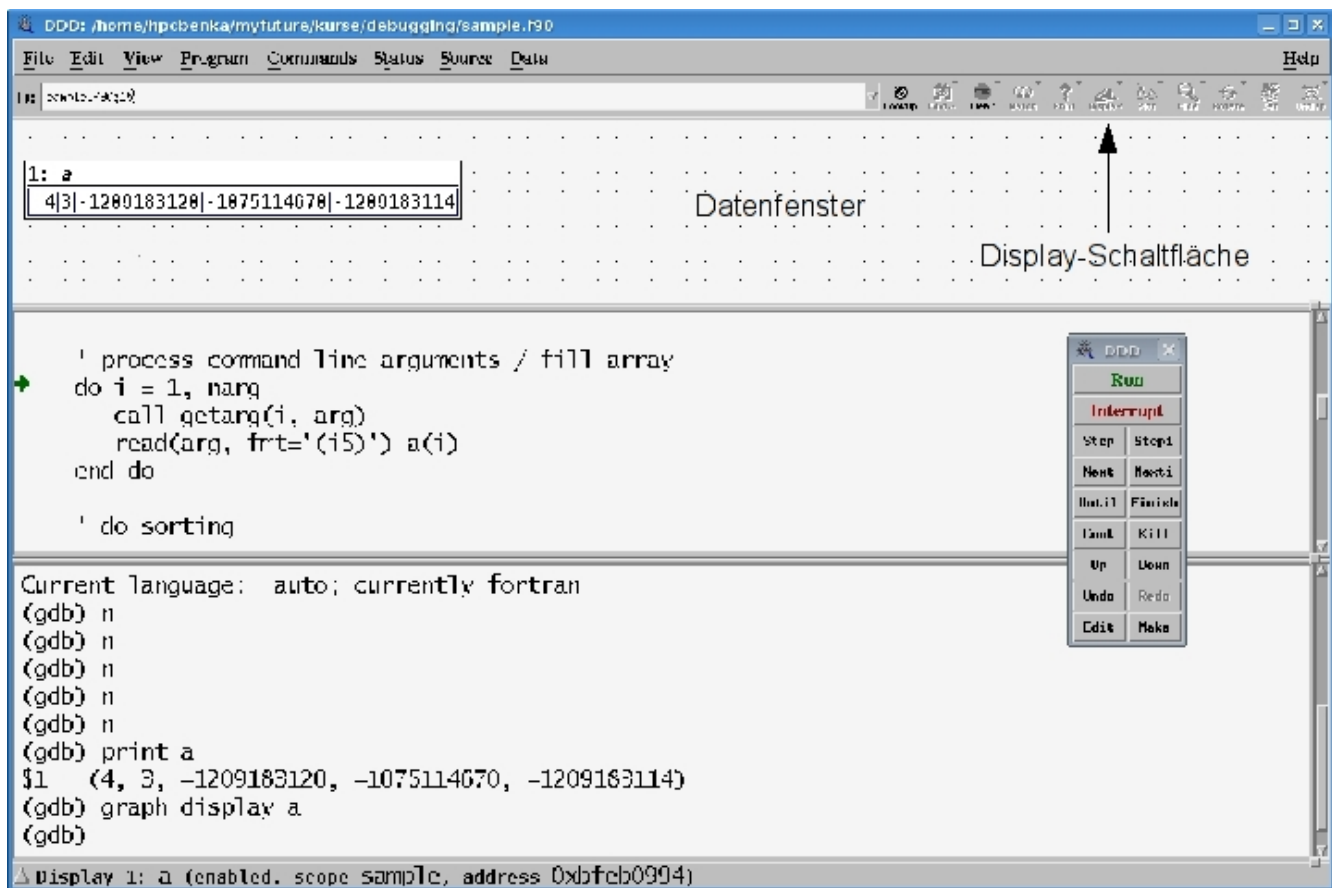
Tip: wenn Sie im Kommandofenster “Return” drücken, wird der letzte Befehl nochmals ausgeführt. Es reicht also, wenn Sie einmal “next” eingeben und dann mehrmals die Taste “Return” betätigen um von Zeile zu Zeile zu springen.

Wenn Sie ein weiteres Mal auf “Next” drücken, wird das erste Element des Feldes a(:) mit 4 belegt. Lassen Sie sich das Feld a anzeigen: klicken Sie mit der rechten Maustaste auf a und wählen Sie im Kontextmenü “Print a” aus. Das Ergebnis ist

```
(gdb) print a
$1 = (4, -1078608264, -1208657586, 3, -1209518992)
(gdb) _
```

a wurde als statisches Feld der Größe 5 deklariert, wir sehen also 5 Einträge und das erste Feld ist mit 4 belegt worden. Die anderen 4 Einträge wurden noch nicht initialisiert, deshalb steht dort Müll.

Statt mit Print das Feld in jeder Iteration auszugeben, können wir a auch anzeigen lassen. Dazu klicken Sie bitte mit der rechten Maustaste auf a und wählen im Kontextmenü Display aus. Der Inhalt von a wird jetzt in im Datenfenster angezeigt. Klicken Sie auf das Feld im Datenfenster mit der rechten Maustaste und wählen im Kontextmenü Rotate aus um das Feld horizontal anzuzeigen.



Beim nächsten Schleifendurchlauf wird das Feld a im Datenfenster automatisch aktualisiert. Werte, die sich geändert haben werden farbig hervorgehoben.

Um den Loop weiter auszuführen, benutzen wir `Until`. GDB führt das Programm weiter aus bis eine Zeile erreicht ist, die größer als die bisherige ist. Klicken Sie auf `Until` bis Sie bei dem Aufruf von `shell_sort` sind.

```
=> shell_sort(a, nargs+1);
```

Die Werte von a and diesem Punkt sind 4 1, das Einlesen von a hat also funktioniert. Klicken Sie wieder auf `Next` um das Programm über den Funktionsaufruf hinweg auszuführen. DDD ist jetzt an der Stelle

```
=> print *, a
```

und im Datenfenster sehen wir, dass nach dem Aufruf von `and shell_sort` der Inhalt von a seltsam aussieht. `shell_sort` ist also für die falschen Ergebnisse verantwortlich.

Um genau herauszufinden, was passiert ist, müssen wir das Programm nochmals ausführen. Wir fügen also bei dem Aufruf zu `shell_sort` einen neuen Breakpoint ein und löschen den alten, indem wir mit der rechten Maustaste auf das Stop-Zeichen klicken und "Delete Breakpoint" auswählen. Danach klicken wir in der Toolbar auf "Run". Die Argumente werden vom letzten Programmaufruf übernommen und müssen nicht mehr neu eingegeben werden.

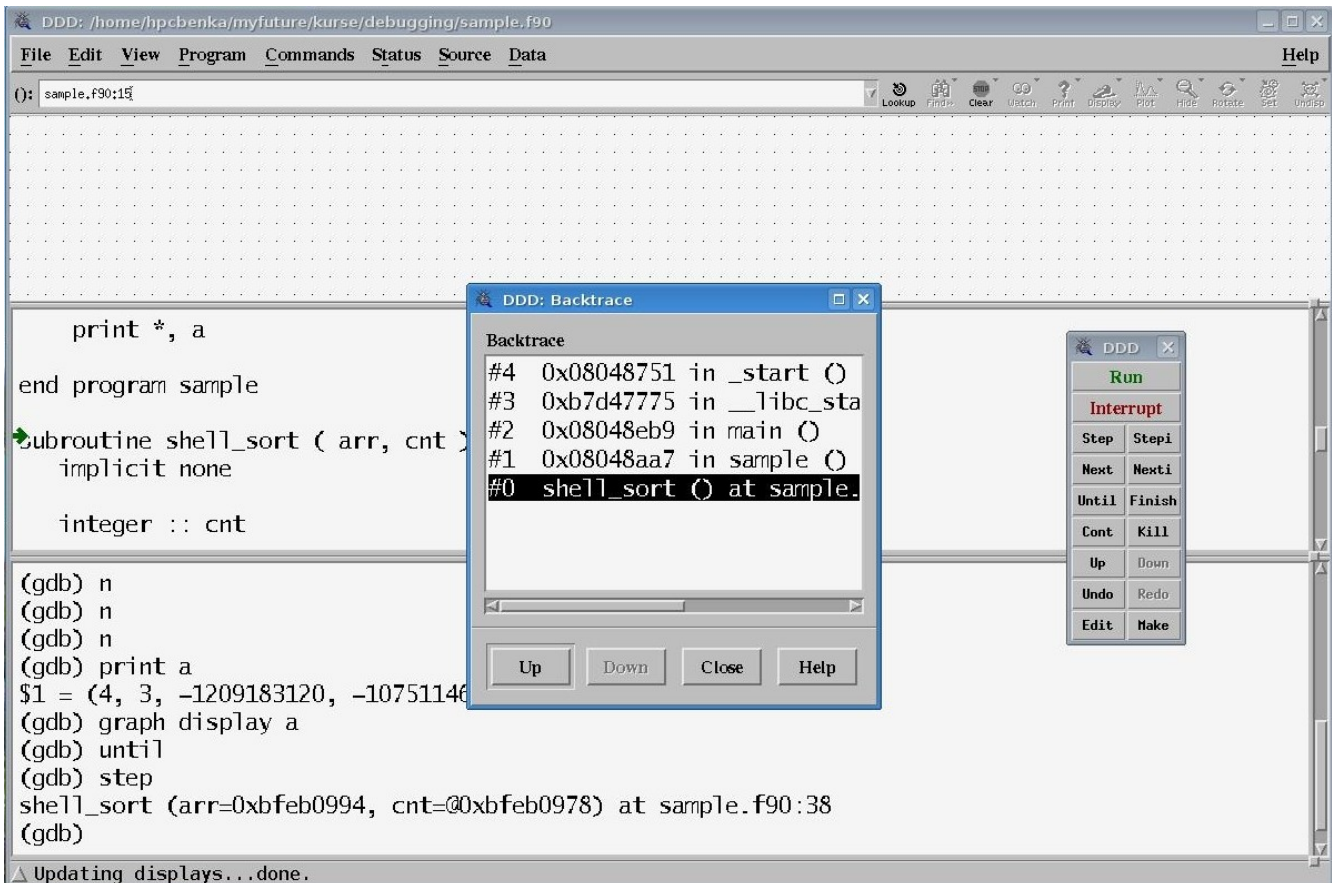
Wir landen in der Zeile, in der `shell_sort` aufgerufen wird:

```
=> call shell_sort(a, nargs)
```

Dieses Mal wollen wir uns ansehen, was in `shell_sort` passiert. Klicken Sie auf `Step`, um in das `shell_sort` hineinzugehen. DDD teilt uns im Kommandofenster mit, dass wir `shell_sort` aufgerufen haben

```
(gdb) step
shell_sort (arr=0xbfe2a10c, cnt=@0xbfe2a0c) at sample.f90:53
(gdb) _
```

Das Feld `a` im Datenfenster ist verschwunden, da wir uns in einer Subroutine befinden, in der es dieses Feld nicht gibt. Diese Abfolge von Aufrufen zu Subroutinen mit den dazugehörigen Änderungen in den Variablen nennt man `stack`. Diesen Stack können wir uns anzeigen lassen, wenn wir auf `Status => Backtrace` klicken. Wenn wir eine Zeile auswählen (oder `Up` und `Down` benutzen) bewegen wir uns im Stack hin und her. Klicken wir zum Beispiel auf `main()` wird `a` wieder angezeigt.



Überprüfen wir also, ob die Argumente, mit denen wir `shell_sort` aufgerufen haben, korrekt sind. Nachdem wir wieder im Stack frame "`shell_sort`" sind, geben wir die Werte von `arr(1)`, `arr(2)` und `cnt` aus.

```
(gdb) print arr(1)
$11 = 4
(gdb) print arr(2)
$11 = 1
(gdb) print cnt
$12 = (REF T0 -> ( integer=(kind=4) )) @0xbfe2a0bc: 3
(gdb) _
```

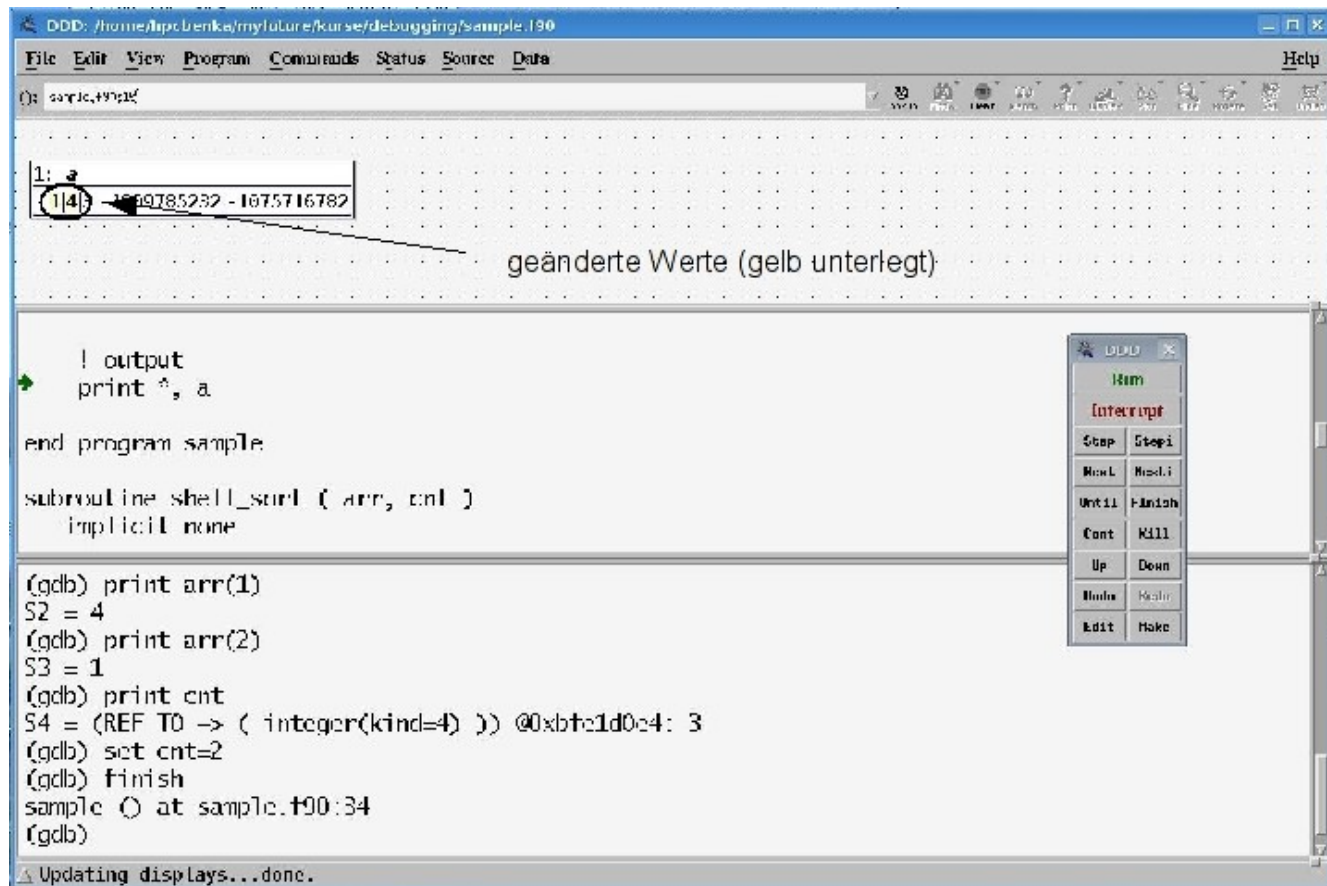
Die unverständliche Teil der Ausgabe von `cnt` ist der schlechten Fortran-Unterstützung von `gdb` geschuldet. Was uns aber interessiert, ist der Wert ganz rechts: `3!` Wir haben aber nur zwei Argumente, also wird ein `arr(3)` mitsortiert, obwohl dort Müll drinsteht.

Um zu sehen, ob das wirklich der Grund allen Übels ist, können wir jetzt cnt den korrekten Wert, nämlich 2, zuweisen. Dazu benutzen wir die Schaltfläche 'SET' oder geben in der Kommandozeile

```
(gdb) set cnt=2 ein
$13 = (REF TO -> ( integer=(kind=4) )) @0xbfe2a0bc: 2
(gdb) _
```

Mit Finish führen wir den Rest von shell_sort aus und kommen zurück ins Hauptprogramm:

```
(gdb) finish
sample () at sample.c:34
(gdb) _
```



Erfolg! Im Datenfeld sehen wir, dass in `a` jetzt die richtigen Werte 1 4 stehen. Wir können das Programm also zu Ende ausführen lassen. Dies geschieht mit `Cont`

```
(gdb) cont
1 4

Program exited normally.
(gdb) _
```

Die Nachricht `Program exited normally.` von GDB zeigt an, dass das Programm `sample` korrekt beendet wurde.

Wir können nun also den Source-Code berichtigen, indem wir auf `Edit` klicken um `sample.f90` zu berichtigen. Wir ändern die Zeile

```
call shell_sort(a, nargs+1)
```

in die richtige Variante

```
call shell_sort(a, nargs)
```

um. Bestätigen Sie die Änderung indem sie “:wq” eingeben. Das ist ein Vim-Kommando, zum Speichern (w - write) und Verlassen (q - quit) der Datei. Drücken Sie “Make” um das Programm neu zu kompilieren.

```
$ gfortran -g -O0 -c sample.f90
$ gfortran -g -O0 -o sample.x sample.o
$ _
```

und überprüfen jetzt, indem wir das Programm nochmals ausführen, ob sample jetzt funktioniert

```
(gdb) run
(gdb) Breakpoint 1, sample () at sample.f90:41
(gdb) c
      1          4
Program exited normally.
(gdb) _
```

Beenden Sie DDD mit File => Exit oder Strg+Q.

3 Der einfachere Weg

Jeder Compiler hat eine Menge an Debugflags, um Warnungen und Laufzeittests anzuschalten. Informieren Sie sich, indem Sie das Handbuch oder die Manpages lesen.

Editieren Sie jetzt das Makefile und indem Sie das #-Zeichen in Zeile 17 nach den DBGFLG= entfernen. Compilieren Sie das Programm neu:

```
$ make clean
$ make
```

Wenn man jetzt das Program in der Shell mit den Argumenten '4 1' ausführt, erhält man:

```
At line 56 of file sample.f90
Fortran runtime error: Array reference out of bounds for array 'arr', lower bound
of dimension 1 exceeded (0 < 1)
```

Backtrace for this error:

```
+ function shell_sort (0x8048C96)
  at line 56 of file sample.f90
+ function sample (0x8048AA7)
  at line 34 of file sample.f90
+ /lib/i686/cmov/libc.so.6(__libc_start_main+0xe5) [0xb7cf8775]
```

Den ersten Teil der Ausgabe bekommen wir durch das Flag “-fbounds-check”, dass die Feldgrenzen überprüft. Der zweite Teil ist von “-backtrace”. Er beschreibt den Stack, bis zu dem Punkt, an dem der Fehler aufgetreten ist. Wir hätten also schon gewusst, wonach wir suchen müssen, wenn wir die richtigen Compiler-Flags verwendet hätten.

[Und ich habe herausgefunden, dass mein Code nicht dem Fortran 2003-Standard entspricht, was auch keine gute Sache ist.]