A Sample DDD Session

This document is adapted from a chapter in DDD Tutorial. The example C program in the original document has been changed to a C++ program. This chapter illustrates a handful of features are enough to get started using DDD.

The sample program `sample.cpp` exhibits the following bug. Normally, `sample` should sort and print its arguments numerically, as in the following example:

```
$ ./sample 8 7 5 4 1 3
1 3 4 5 7 8
```

However, with certain arguments, this goes wrong:

```
$ ./sample 8000 7000 5000 1000 4000
0 1000 4000 5000 7000
```

Although the output is sorted and contains the right number of arguments, some arguments are missing and replaced by bogus numbers; here, `8000` is missing and replaced by `0` (Actual numbers and behavior on your system may vary).

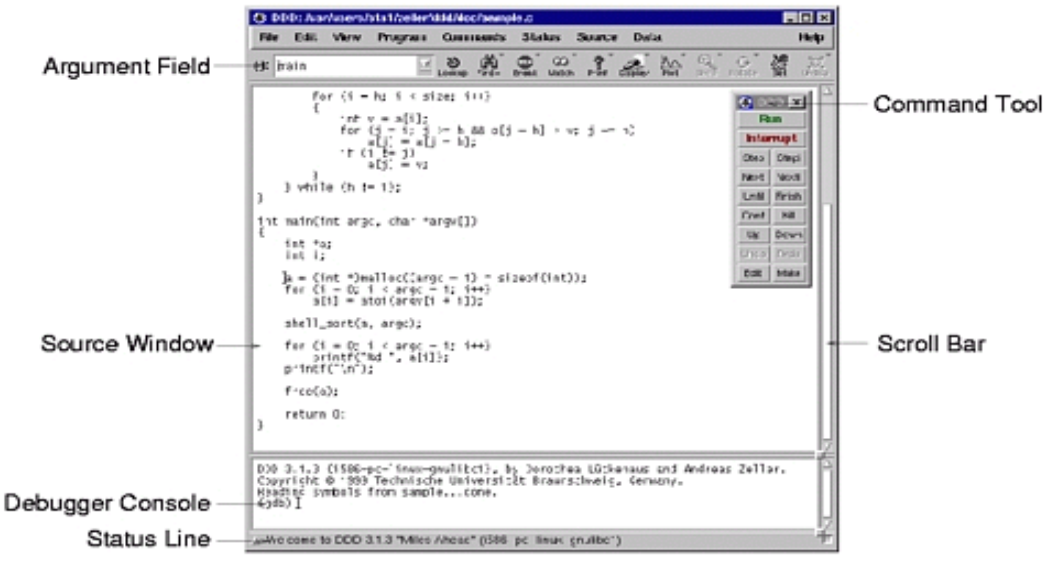
Let us use DDD to see what is going on. First, you must compile `sample.cpp` for debugging, giving the `-g` flag while compiling:

```
$ g++ -g -o sample sample.cpp
```

Now, you can invoke DDD on the `sample` executable:

```
$ ddd sample
```

After a few seconds, DDD comes up. The *Source Window* contains the source of your debugged program; use the *Scroll Bar* to scroll through the file.



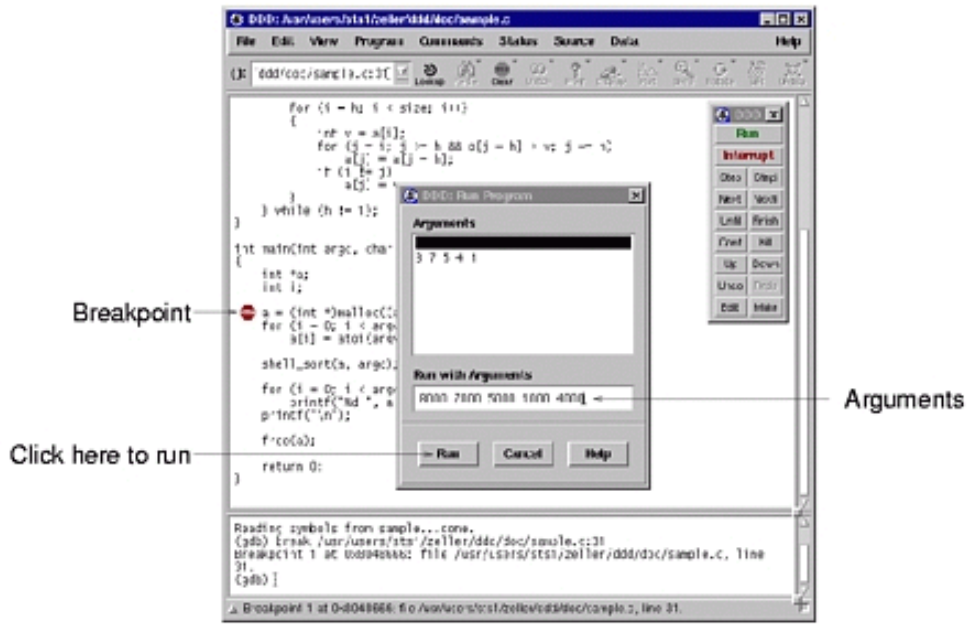
Initial DDD Window

The *Debugger Console* (at the bottom) contains DDD version information as well as a GDB prompt.

```
GNU DDD Version 3.2, by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1999 Technische Universität Braunschweig, Germany.
Copyright © 1999 Universität Passau, Germany.
Reading symbols from sample...done.
(gdb)
```

The first thing to do now is to place a *Breakpoint* (see Breakpoints), making `sample` stop at a location you are interested in. Click on the blank space left to the initialization of `a`. The *Argument field* `():` now contains the location (`sample.cpp:31`). Now, click on `Break` to create a breakpoint at the location in `()`. You see a little red stop sign appear in line 31.

The next thing to do is to actually *execute* the program, such that you can examine its behavior (see Running). Select `Program => Run` to execute the program; the `Run Program` dialog appears.



Running the Program

In `Run with Arguments`, you can now enter arguments for the `sample` program. Enter the arguments resulting in erroneous behavior here--that is, `8000 7000 5000 1000 4000`. Click on `Run` to start execution with the arguments you just entered.

GDB now starts `sample`. Execution stops after a few moments as the breakpoint is reached. This

is reported in the debugger console.

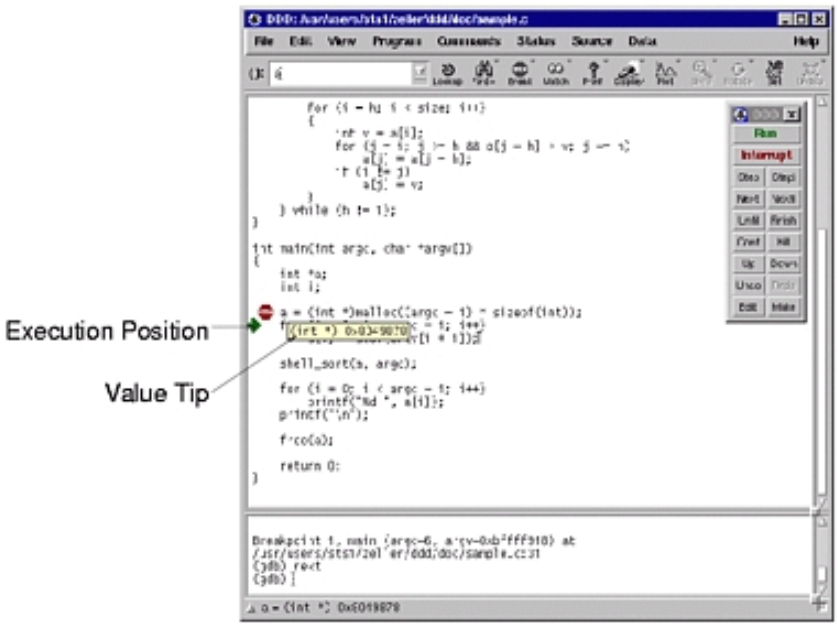
```
(gdb) break sample.cpp:31
Breakpoint 1 at 0x8048666: file sample.cpp, line 31.
(gdb) run 8000 7000 5000 1000 4000
Starting program: sample 8000 7000 5000 1000 4000

Breakpoint 1, main (argc=6, argv=0xbffff918) at sample.cpp:31
(gdb)
```

The current execution line is indicated by a green arrow.

```
=> a = new int[(argc - 1) * sizeof(int)];
```

You can now examine the variable values. To examine a simple variable, you can simply move the mouse pointer on its name and leave it there. After a second, a small window with the variable value pops up (see Value Tips). Try this with `argc` to see its value (`6`). The local variable `a` is not yet initialized; you'll probably see `0x0` or some other invalid pointer value.

To execute the current line, click on the `Next` button on the command tool. The arrow advances to the following line. Now, point again on `a` to see that the value has changed and that `a` has actually been initialized.



Viewing Values in DDD

To examine the individual values of the `a` array, enter `a[0]` in the argument field (you can clear it beforehand by clicking on `()`:) and then click on the `Print` button. This prints the current value of `()` in the debugger console (see Printing Values). In our case, you'll get

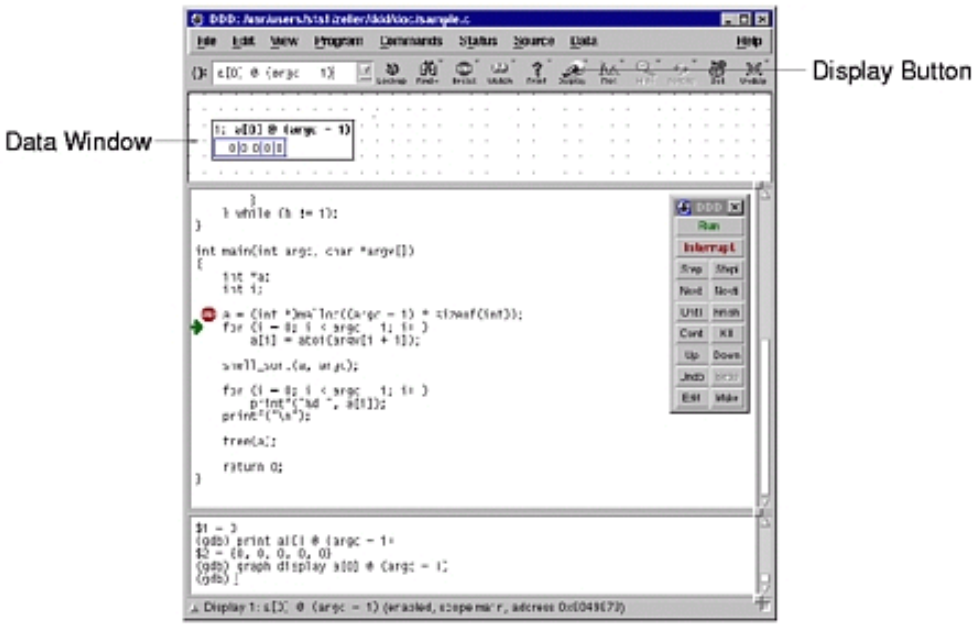
```
(gdb) print a[0]
$1 = 0
(gdb)
```

or some other value (note that `a` has only been allocated, but the contents have not yet been initialized).

To see all members of `a` at once, you must use a special GDB operator. Since `a` has been allocated dynamically, GDB does not know its size; you must specify it explicitly using the `@` operator (see Array Slices). Enter `a[0]@(argc - 1)` in the argument field and click on the `Print` button. You get the first `argc - 1` elements of `a`, or

```
(gdb) print a[0]@(argc - 1)
$2 = {0, 0, 0, 0, 0}
(gdb)
```

Rather than using `Print` at each stop to see the current value of `a`, you can also *display* `a`, such that its is automatically displayed. With `a[0]@(argc - 1)` still being shown in the argument field, click on `Display`. The contents of `a` are now shown in a new window, the *Data Window*. Click on `Rotate` to rotate the array horizontally.



Data Window

Now comes the assignment of `a`'s members:

```
=>  for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);
```

You can now click on `Next` and `Next` again to see how the individual members of `a` are being assigned. Changed members are highlighted.

To resume execution of the loop, use the `Until` button. This makes GDB execute the program until a line greater than the current is reached. Click on `Until` until you end at the call of `shell_sort` in

```
=>  shell_sort(a, argc);
```

At this point, `a`'s contents should be `8000 7000 5000 1000 4000`. Click again on `Next` to step over the call to `shell_sort`. DDD ends in

```
=>  for (i = 0; i < argc - 1; i++)
        cout << a[i] << ' ';
```

and you see that after `shell_sort` has finished, the contents of `a` are `0, 1000, 4000, 5000, 7000`--that is, `shell_sort` has somehow garbled the contents of `a`.

To find out what has happened, execute the program once again. This time, you do not skip through the initialization, but jump directly into the `shell_sort` call. Delete the old breakpoint by selecting it and clicking on `Clear`. Then, create a new breakpoint in line 33 before the call to `shell_sort`. To execute the program once again, select `Program => Run Again`.

Once more, DDD ends up before the call to `shell_sort`:

```
=>  shell_sort(a, argc);
```

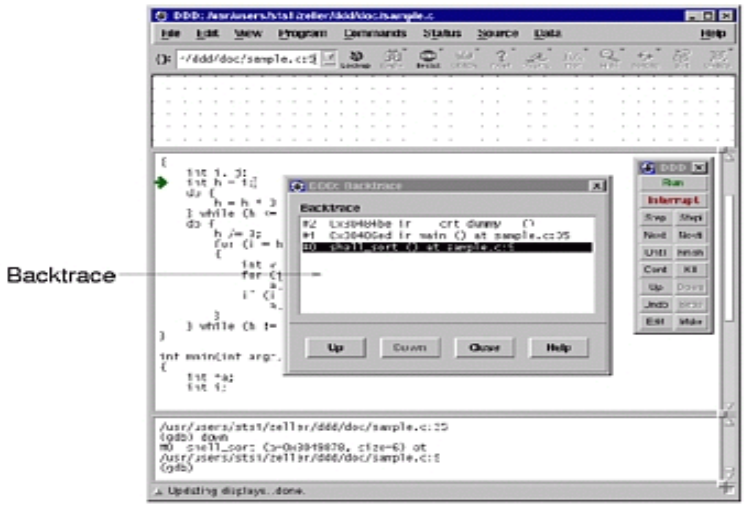
This time, you want to examine closer what `shell_sort` is doing. Click on `Step` to step into the call to `shell_sort`. This leaves your program in the first executable line, or

```
=> int h = 1;
```

while the debugger console tells us the function just entered:

```
(gdb) step
shell_sort (a=0x8049878, size=6) at sample.cpp:9
(gdb)
```

This output that shows the function where `sample` is now suspended (and its arguments) is called a *stack frame display*. It shows a summary of the stack. You can use `Status => Backtrace` to see where you are in the stack as a whole; selecting a line (or clicking on `Up` and `Down`) will let you move through the stack. Note how the `a` display disappears when its frame is left.



The DDD Backtrace

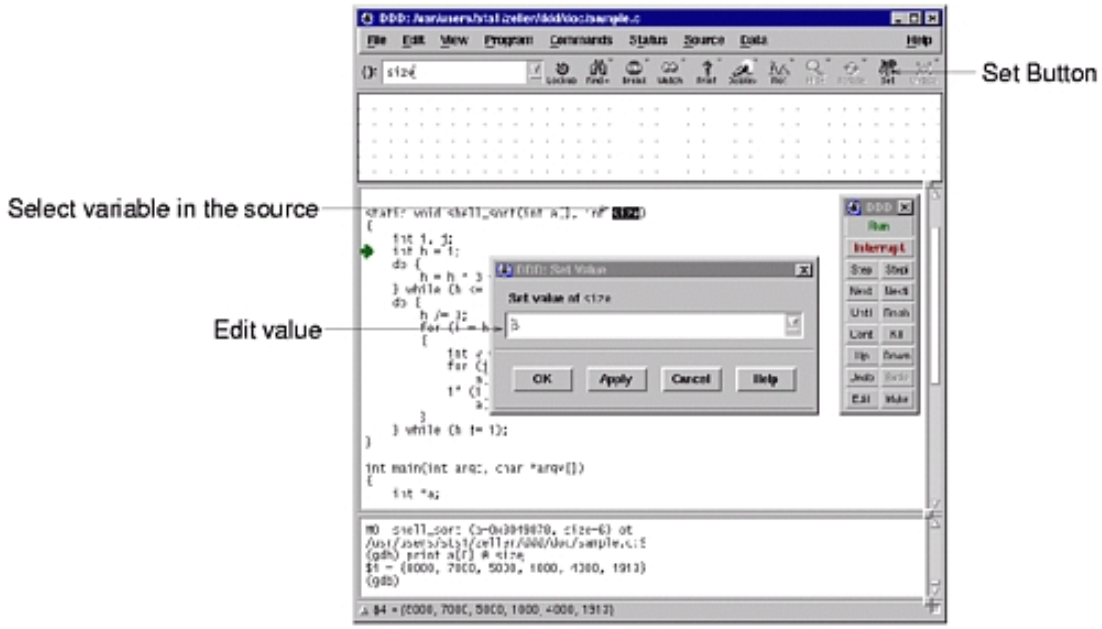
The DDD Backtrace

Let us now check whether `shell_sort`'s arguments are correct. After returning to the lowest frame, enter `a[0]@size` in the argument field and click on `Print`:

```
(gdb) print a[0] @ size
$4 = {8000, 7000, 5000, 1000, 4000, 0}
(gdb)
```

Surprise! Where does this additional value `0` come from? The answer is simple: The array size as passed in `size` to `shell_sort` is *too large by one*--`0` is a bogus value which happens to reside in memory after `a`. And this last value is being sorted in as well.

To see whether this is actually the problem cause, you can now assign the correct value to `size` (see Assignment). Select `size` in the source code and click on `Set`. A dialog pops up where you can edit the variable value.



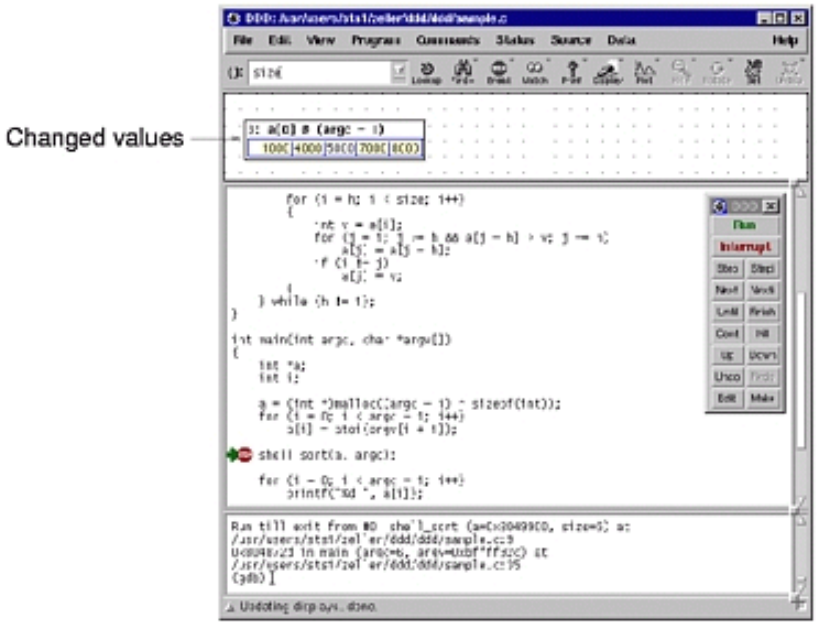
Setting a Value

Change the value of `size` to `5` and click on `OK`. Then, click on `Finish` to resume execution of the `shell_sort` function:

```
(gdb) set variable size = 5
(gdb) finish
Run till exit from #0  shell_sort (a=0x8049878, size=5) at sample.cpp:9
0x80486ed in main (argc=6, argv=0xbffff918) at sample.cpp:33
```

```
(gdb)
```

Success! The `a` display now contains the correct values `1000, 4000, 5000, 7000, 8000.`



Changed Values after Setting

You can verify that these values are actually printed to standard output by further executing the program. Click on `Cont` to continue execution.

```
(gdb) cont
1000 4000 5000 7000 8000

Program exited normally.
(gdb)
```

The message `Program exited normally.` is from GDB; it indicates that the `sample` program has finished executing.

Having found the problem cause, you can now fix the source code. Click on `Edit` to edit `sample.cpp`, and change the line

```
shell_sort(a, argc);
```

to the correct invocation

```
shell_sort(a, argc - 1);
```

You can now recompile `sample`

```
$ g++ -g -o sample sample.cpp
```

and verify (via `Program => Run Again`) that `sample` works fine now.

```
(gdb) run
'sample' has changed; re-reading symbols.
Reading in symbols...done.
Starting program: sample 8000 7000 5000 1000 4000
1000 4000 5000 7000 8000

Program exited normally.
(gdb)
```

All is done; the program works fine now. You can end this DDD session with `Program => Exit` or `Ctrl+Q`.
