Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Visual Debugging with DDD

Source Code Accompanies This Article. Download It Now.

- vdebug.txt

If a debugger is a tool that lets you "see" what's going on in a program, then DDD is the tool that lets you see the most.

March 01, 2001
URL:http://www.drdobbs.com/tools/visual-debugging-with-ddd/184404519

Seeing is believing when it comes to tracking errors

Andreas is an assistant professor at the University of Passau, Germany, where he works on Delta Debugging (a method to isolate failure causes automatically) and maintains and extends DDD. He can be contacted at http://www.fmi.uni-passau.de/st/staff/zeller/.

Do you remember the movie *Tron*? A young computer genius becomes part of a computer game, walking through a world where every component lives and breathes, from the programs (represented by alter egos of their creators) to the individual bits (glowing green spheres that can say only "yes" or "no"), fighting against the evil Master Control Program (see http://www.tronmovie.com/).

When the movie came out in 1982, I had just started programming, and dreamt of a tool that would actually let me immerse myself into my computer's internal secret world, where I could see the data whirling around and capture all its interactions. So I got excited when a fellow at the university introduced me to debugging tools, which promised to "let me see what's going on in my program." Needless to say, I was disappointed. What he showed me was one of those command-line debuggers where you can enter some variable name and see its value. Useful, for sure, and versatile too, but I still dreamt of immersion and direct manipulation.

Since the days of *Tron* and command-line debuggers, the typical debugging tool has evolved. Of course, it comes with a GUI, shows the actual program text being executed (with breakpoints and backtraces), and you can click on anything with your mouse. Unfortunately, the data presentation has not evolved at all. For instance, Figure 1 shows a linked list as displayed by your favorite debugger. Do you notice anything special about it?

Actually, it is a circular list. How do you know? By reading and comparing the pointer values, which happen to be the same for *list* and *list->next->next->next*. (Now try this with a 100-element list.) The structure is much easier to capture in classical textbook notation, as in Figure 2.

Here comes the good news. Figure 2 is neither taken from a textbook nor drawn by a human. It actually is a snapshot of a running program, generated by a freely available debugger named "DDD" (available at http://www.gnu.org/software/ddd/). What DDD does is visualize objects and references in a textbook style — you select what you want to see and how you want to see it.

A Quick Tour of DDD

Figure 3 is the main DDD screen. At the center of all things is the source code. Following the usual standards for debugger GUIs, the current execution position is indicated by a green arrow; breakpoints are shown as stop signs.

You can navigate around the code using the Lookup toolbar button or the Open Source dialog from the File menu. Double clicking on a function name leads you to its definition. Using the Undo and Redo buttons, you can navigate to previous and later positions — similar to your web browser.

You can set and edit breakpoints by double clicking in the white space to the left; to step through your program or to continue execution, use the floating command tool on the right. Command-line aficionados will find a debugger console at the bottom. If you need anything else, try the Help menu for detailed instructions.

Visualization

Moving the mouse pointer on an active variable shows its value in a little pop-up screen. Snapshots of more complex values can be "printed" in the debugger console. To view a variable permanently, though, use the Display button. This creates a small permanent window (a display) which shows the variable name and value. These displays are updated every time the program changes its state.

To access a variable value, you must bring the program in a state where the variable is actually alive; that is, within the scope of the current execution position. Typically, you set a breakpoint within the function of interest, run the program, and display the function's variables.

To actually visualize data structures (that is, data as well as relationships), DDD lets you create new displays out of existing displays. For instance, if you have displayed a pointer variable *list*, you can dereference it and view the value it points to simply by double-clicking on the pointer value. This creates a new display **list*, with an arrow pointing from *list* to **list*. You can repeat this, say, with *list->self, list->next,* and the like, until you eventually see the entire list (Figure 4).

Each new display is automatically laid out in a fashion to support simple visualization of lists and trees. For instance, if an element already has a predecessor, its successor will be laid out in line with these two. You can always move elements around manually simply by dragging and dropping the displays. Also, DDD lets you scroll around, layout the structure, change values manually, or see them change while the program runs. An Undo/Redo capability even lets you redisplay previous and later states of the program, so that you can see how your data structure evolves.

Graph Structures

The drawback of DDD's simple visualization scheme is that since only one arrow can point to each display, the visualization is limited to tree structures. In the aforementioned example for instance, each *X->self* actually points to *X,* not another object. You can resolve this problem by activating alias recognition in the Data menu. With alias recognition enabled, DDD merges all displays with identical memory addresses into one; in this case, the displays *X->self* and *X*. This results in the view of Figure 2. Not only does every *self* pointer reference the object itself, it also turns out that this is a circular list. Compare Figure 2 and Figure 1, and you will find that a picture says more than a thousand words.

How does alias recognition work? For each display, DDD tracks its address in memory. (Try *Data->Displays* to see the address.) After each change to the displayed data, DDD verifies whether two or more displays have the same memory address. If this is the case, all displays of this group except one are suppressed and the arrows pointing towards them now point to the one remaining display. This one remaining display is actually the oldest of the group, that is, the first one to be reached and displayed by users.

Besides pointers, DDD can visualize arbitrary references between data. If, for instance, you create a new display *a[0].data* from display *a*, then DDD will annotate the arrow with *[0].data*. It will also memorize the operation such that you can apply it on *b* to obtain *b[0].data* and likewise. This is useful for languages and data structures without pointers. Alias recognition still applies here.

The luxury of textbook data representation comes at a price — screen space. While it is still feasible to examine a 20-element list, there is no convenient way to view a 1000-element list in its entirety. The thousand pictures may be worth more than a million words, but even this is too much to comprehend at a single glance. Defensive programming, with several assertions, will help you bring the problem down to a size where it can be displayed and examined.

Screen space also gets scarce when displaying multiple variables at once. DDD provides a *Cluster* function which combines several displays into one list — instead of having one display for each variable, you have one display showing all variables. A similar display can be created with *Data->Local Variables* to view all local variables in one display. A problem with GDB is that variables at different backtrace levels cannot be shown simultaneously; you have to walk up and down the backtrace to view the variables local to the selected frame.

Currently, DDD is being expanded to support custom views called "themes" — a kind of filter that changes some particular aspect of the data rendering. For instance, you can have specific variables or values be displayed in a different color, at a different size, or not at all. This lets you focus on specific aspects of a data structure. According to the developers, this theme's feature will be extended such that even large linked lists will be displayed as a single entity. If you display each list element using only a few pixels, then you'll be able to examine very large data structures. The disadvantage is that every new theme will require a little bit of user-side programming, something the DDD developers have avoided so far.

Plots

If you are dealing with numerical data, DDD provides an alternate visualization method: plots. By means of the Plot button, DDD can display any array of numerical data in a separate plot window (Figure 5). This view is also updated each time the program stops.

DDD can plot one- and two-dimensional numerical arrays in two- or three-dimensional plots, respectively. Scalars (that is, simple numerical values) can also be plotted. This is useful when combining several clustered variables into one plot, such

as an array and its iterator. DDD internally relies on the Gnuplot program to produce the plots and thus inherits some of its capabilities, such as various output options and formats.

If the array to be displayed is allocated dynamically, DDD must be told about the actual size. If you want to see, say, the first 50 elements of array *a,* you enter *a[0] @ 50* as the value to be displayed. This "array slicing" syntax is inherited from GDB. As an alternative, you can also enter *a[0..49].* (This is somewhat slower because every single value from *a[0]* to *a[49]* is individually queried from GDB.) Finally, it is possible to plot the history of a variable, that is, its value during the last program stops. To make this work, be sure to display the variable right from the program start, such that DDD can record its value.

The DDD plot facility does not replace a full-fledged data mining package like Sun's Prism debugger (http://docs.sun.com/; search for "Prism User's Guide"), which lets you view two-dimensional data as thresholds, surfaces, or vectors. Also, DDD unfortunately does not let you plug in your own visualizers. On the other hand, DDD's plots are straightforward to use and only one mouse click away.

Other Goodies

Besides the eye-catching visualization capabilities, DDD has some hidden treasures. One I find useful in practice is session management. You can save the entire state of the program (and DDD) to disk and resume investigation at a later time. All debugger settings are restored, as well as the memory and execution state of the program. This lets you abandon your debugging work in the evening — and resume it in the morning, with a fresh mind. The only drawback is that you cannot resume execution of a restored program (the data is restored, but the external resources such as open files are not); it must be restarted. But this is a breeze because of course, the arguments are restored as well.

Another goody is the ability to move breakpoints and the execution position just by dragging and dropping. It frequently happens that I am off by one line when setting a breakpoint, so I can easily correct my mistake by moving the breakpoint to the correct position. (The Undo button would undo this mistake, too.)

A more arcane feature is the ability to move the execution arrow around. If you missed something during a function execution, you can move the execution arrow back to the function call and run through the function a second time. (Don't try this for *free()* or *close(),* or other functions with nonrepeatable side effects.) It is even possible to move the execution position from one function to another, but DDD requires your confirmation for this. Anyway, why would you want that?

Behind the Scenes

Technically speaking, DDD is not a stand-alone debugger, but only a front end to a command-line debugger — typically, the GNU debugger (GDB). Actually, every user interaction eventually gets translated into a debugger command, and the debugger's reply is shown in the GUI. Some of this interaction takes place in the debugger console. For instance, Example 1(a) is the interaction that happens if you click on a button to set a breakpoint, say, in line 99 of source file foobar.c. The *break* command is automatically generated by DDD and sent to GDB. From GDB's reply, DDD extracts the breakpoint number, its address and location, and displays a stop sign at this location.

The good thing about this explicit interaction in the debugger console is that it maintains a command history and a command editor. You can easily search, repeat, and alter previous commands. In the long term, you will also learn most of the GDB commands (something that isn't easy for a beginner). And as an expert, you may even prefer to enter your commands right there — or define your own commands.

But besides the interaction in the debugger console, much more communication takes place behind the scenes. If you invoke DDD using the *verbose* option, all interaction between GDB and DDD will be printed on standard output. If you dereference a displayed *list* to obtain **list,* Example 1(b) is the interaction happening behind the scenes.

You see that besides the actual contents of **list* (which will eventually be shown on the screen), DDD also inquires the current frame and the address of **list*. The frame is required for saving and restoring sessions. When restoring a session and DDD fails to create **list* (typically, because **list* is not alive yet), DDD will postpone the creation of display **list* until execution has reached the previously recorded frame. Later on, the address of **list* is recorded for detecting aliases; that is, displayed with the same address which will then be merged.

Another example of what's going on behind the scenes is breakpoint management. Let's say you actually tried to move the execution arrow from one function to another. This is what's happening: First, DDD sets a temporary breakpoint (a breakpoint that will be deleted when reached) at the expected location; an *info breakpoints* command follows to verify whether the breakpoint has actually been set. This breakpoint is required because the GDB *jump* command (which actually moves the execution position) would otherwise immediately resume execution; see Example 2(a). The GDB confirmation question gets transformed into a confirmation dialog with the same wording, see Example 2(b), which I prefer to answer with a click on the No button. This results in DDD sending a *no* reply to GDB's query, followed by a *delete* command to delete the temporary breakpoint.

All of this interaction takes place without you ever typing a single command. In fact, if you use DDD with different command-line debuggers, you will find that the commands differ quite a lot, but DDD's user interface remains the same.

On the other hand, you can always enter commands if you prefer, and even with your own user-defined commands, DDD will always reflect the current state in its windows.

Weaknesses

Now for the weaknesses. Besides GDB, DDD supports DBX, JDB, XDB, WDB, the Perl debugger, and a variant of the Python debugger. Okay, this is a strength rather than a weakness, but if your debugger is not in that list, DDD will not work for you. Nor DDD for Microsoft and Borland compilers, either, because their debugging information is not understood by GDB.

The front-end nature of DDD makes it quite universal and versatile (one front end for all debuggers), but also imposes a couple of limits. For instance, DDD's knowledge about the debugged program is only at a lexical level. If some C++ variable is named *a* in the program text, but *X::a* in the symbol table, DDD cannot know this, and it's up to you to find this out. Likewise, DDD cannot display other sets of variables than those provided by the command-line debugger. That is, all local variables or all parameters can be displayed if the command-line debugger provides a command for doing so, but there is no menu from which to choose all live variables, for instance.

Another problem is that every bit of information must go through the command-line debugger. This makes the combination of DDD and the command-line debugger slower than a single debugger with an integrated GUI. On the other hand, if your command-line debugger crashes, DDD will live on and offer to let you restart it without losing breakpoints or other settings.

Also, every deficiency of the command-line debugger eventually becomes a problem in DDD. Although DDD works with JDB, Sun's Java debugger, JDB has not been very reliable in the past — at least not on my Linux box. Your favorite Windows Java IDE is probably more stable. If you use GDB as command-line debugger, though, DDD is the GUI of choice.

Another weakness of DDD is the lack of user-side extensibility. If you expect something like Emacs, where every minor detail can be reprogrammed, you'll be disappointed. You cannot adapt DDD to another command-line debugger easily, nor is there an interface for adding alternate data rendering tools. (You can, however, integrate DDD wherever GDB fits in, such as in Emacs.) The GVD project (http://libre.act-europe.fr/gvd/) aims to overcome these deficiencies by providing a more modular architecture.

Finally, DDD is quite a memory hog. This is no surprise considering that your program, the command-line debugger, DDD, and the X server all compete for time and memory. If you have a Linux box with less than 32 MB of RAM, or a processor of less than 200 MHz, consider one of the lightweight alternatives such as letting GDB run within Emacs, or even running the command-line debugger standalone. Don't worry, DDD will teach you all the commands you need.

Getting DDD

All things considered, where do you get DDD? First, check the requirements. DDD wants a POSIX environment and an X server. Both requirements are trivial to fulfill on Linux and UNIX boxes, but to make DDD run under Windows is a major task. People do this, though, and they have been successful with the help of Red Hat's Cygwin package. (Search the archives in http://sources.redhat.com/cygwin/ for "DDD.")

The third thing that DDD requires is a Motif toolkit, or its clone, LessTif. Motif is normally part of your UNIX box, and LessTif can easily be installed on your Linux box (or your Windows box, if you have Cygwin). If you have all these prerequisites, you can either download the C++ source code or some ready-to-run executable via the project home page (http://www.gnu.org/software/ddd/). For free, of course, since DDD is a GNU program. Normally, DDD should build and run right out of the box; if it doesn't, ask your local guru (or the DDD mailing list) for assistance.

Back to the Future

If a debugger is a tool that allows you to see what's going on in a program, then DDD is probably the tool that allows you to see the most. Obviously, this is still dull compared to a movie like *Tron*, or even to the marvels as produced by scientific visualization. But we are getting closer. If you want to take a look into the future, step by at Wim de Pauw's Jinsight project at IBM and gaze in wonder at the Java postmortem trace visualizations (http://www.research.ibm.com/jinsight/). If these beauties can be inspected dynamically during a program run, and if they can be combined with DDD-like data visualization, then we've come much closer to *Tron*-like immersion.

Of course, we'd need 3D visualization, and, of course, direct 3D manipulation. Eventually, I'll jump into my data suit, put on my 3D goggles, immerse myself into the running program, follow that data flow, grab it, and redirect it to the proper sink. But wait. Why can't I bend this pointer? Help! Overflow! I'm drowning!

DDJ

(a)

```
(gdb) break foobar.c:99
```

```
Breakpoint 1 at 0x8048a7d: file foobar.c, line 99.
(gdb) _

(b)
(gdb) frame
#0 list_test (start=46) at foobar.c:399
\032\032foobar.c:99:8266:beg:0x8049143
(gdb) display *list
2: *list = {
value = 85,
self = 0x804eb50,
next = 0x804eb60
}
(gdb) output &(*list)
(List *) 0x804eb50
(gdb) _

(gdb) delete 3
(gdb) info breakpoints
 ...
(gdb) _
```

Example 1: (a) Interaction at the debugger console; (b) interaction behind the scenes.

```
(a)
(gdb) tbreak foobar.c:422
Breakpoint 3 at 0x80492b0: file foobar.c, line 422.
(gdb) info breakpoints
 ...
(gdb) jump foobar.c:422
Line 422 is not in `list_test(int)'. Jump anyway? (y or n)

(b)
Line 422 is not in `list_test(int)'. Jump anyway? (y or n) no
Not confirmed.
(gdb) delete 3
(gdb) info breakpoints
 ...
(gdb) _
```

Example 2: (a) Do we really want to move the execution position? (b) No, we don't want to move it.

```
list = (List *) 0x804eb50
list->next = (List *) 0x804eb60
list->next->next = (List *) 0x804eb70
list->next->next->next = (List *) 0x804eb50
list->next->next->next->next = (List *) 0x804eb60
 ...
```

Figure 1: A list in textual notation.

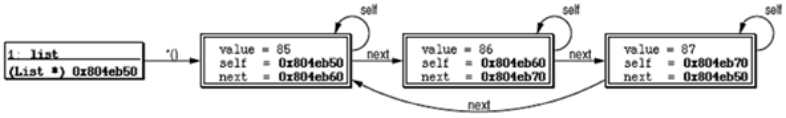

Figure 2: A textbook view of the same list.

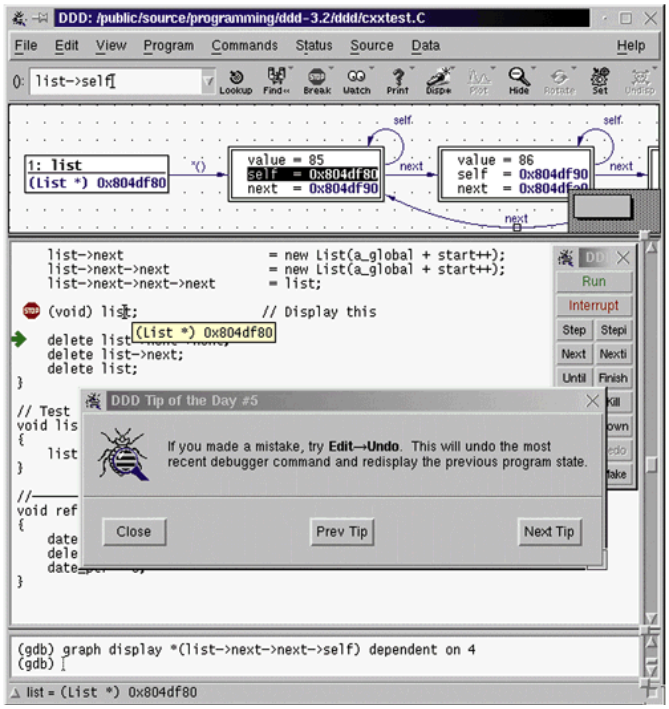

Figure 3: A DDD screen shot.

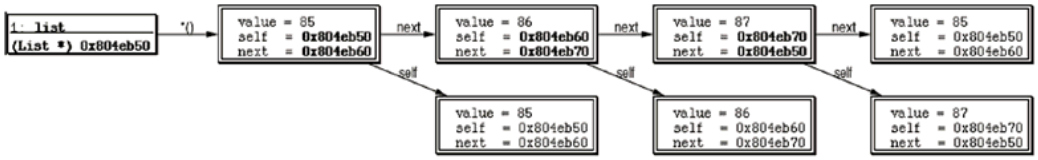

Figure 4: A list in DDD.

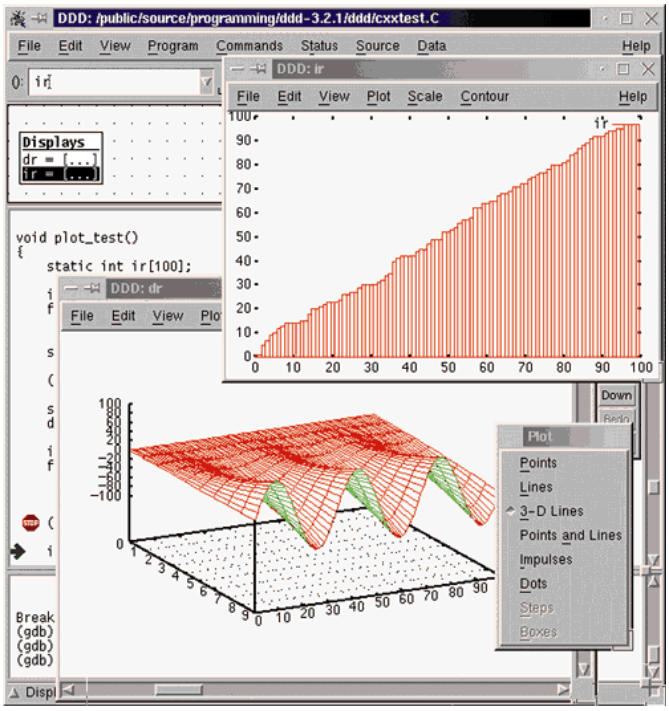

Figure 5: Plots within DDD.