Debugging with NuWa

Brett Viren

Physics Department



Daya Bay Offline Workshop @ BNL, April 2008

Outline

1. Compiling for Debugging
2. Running GDB
3. Interrogating and Exploring a Program in GDB
4. Controlling Program Execution
5. Stepping Through the Execution
6. Dealing with Dynamically Loaded Libraries

Compiling code for easier debugging.

GDB can use extra information in the object code to provide more information. This is added with GCC's -g flag.

```
shell$ g++ -ggdb [-O] file.cxx ...
```

Note:

- Using -ggdb will provide the best information for GDB. Using -g alone will use the OS's native format (which can be equivalent).
- One can optionally still keep optimization on (-O2) but the generated code may be very confusing when investigated in GDB. Variables can be optimized away, branches can execute out-of-order, etc.

Compiling Externals for Debugging

Work needed here:

- The external packages are not yet configured to build debug versions.
- They should build into a platform-specific directory that are unique from the non-debug versions as specified by the CMT macro `LCG_system`.
- Support a mix of some packages w/ debug and some w/out?

Volunteers?

Compiling projects for Debugging

You can compile the projects for debugging by adding "debug" to "CMTEXTRATAGS".

```
bash$ export CMTEXTRATAGS=dayabay,debug
```

```
tcsh$ setenv CMTEXTRATAGS dayabay,debug
```

Work needed: this will overwrite existing libraries. It would be good to support installation of both debug and optimized packages side-by-side.

Start the Process with GDB

Two ways to pass in command line arguments, from inside GDB:

```
shell$ gdb program
```

```
(gdb) run arg1 arg2
Starting program:  /path/to/program ...
```

or from the shell:

```
shell$ gdb --args program arg1 arg2 ...
```

```
(gdb) run
Starting program:  /path/to/program ...
```

But we use python as the main program.

No problem:

```
shell$ gdb --args python script.py arg1 arg2 ...
```

```
(gdb) run
Starting program:  /path/to/python ...
```

Postmortem with a "core" file

A core file holds an image of the program's memory at the time of the crash. To be produced your environment must be properly set:

```
bash$ ulimit -c unlimited
```

```
tcsh$ limit coredumpsize unlimited
```

GDB can attach to the core file along with the program like:

```
shell$ gdb program core
```

GDB will position itself at the point where execution crashed.

Connecting to a process that is already running

This can be useful if you suspect the program is in an infinite loop, or catch the program after all libraries are loaded (but see below for other ways).

```
shell$ gdb program PID
```

Where `PID` is the running process's ID. GDB will halt the process and will position itself where the process was just executing.

Caveat About Running **root** with GDB

The **root** executable is a small program that just handles a few things like printing a help message or showing the graphical ROOT "splash" screen. It calls **exec** to start the **root.exe** program which then does the heavy lifting. This **exec** can confuse GDB so when debugging "ROOT", do:

```
shell$ gdb root.exe ...
```

Caveat About Running GDB under **tcsh**

GDB will run the program under your shell but **only** if the `SHELL` environment variable is correctly set. The **tcsh** shell does not always set this variable. It can be set like:

```
tcsh$ setenv SHELL /usr/bin/tcsh
```

If this variable is left unset, **/bin/sh** is used.

Or better yet, switch to bash....

Simple Test Program

```
1 void func2(int *p)
2 {
3     int i = *p;
4 }
5 void func1(int* p)
6 {
7     func2(p);
8 }
9 void func0(int*&p)
10 {
11     p = new int;
12     func1(p);
13     delete p;
14     p = 0;
15 }
16 int main (int argc, char *argv[])
17 {
18     int* p=0;
19     func0(p);
20     func1(p);
21     return 0;
22 }
```

Where am I and how did I get here?

When GDB runs a program and stops either due to a crash or an interupt it is positioned somewhere in the function call stack. You can see where with the **backtrace** (aka **bt**, **where**):

```
(gdb) backtrace
#0  0x0804838d in func2(int*) (p=0x0) at crash.cc:3
#1  0x080483a5 in func1(int*) (p=0x0) at crash.cc:7
#2  0x080483ca in main (argc=1, argv=0xbffffae4) at crash.cc:20
```

This shows **main** called **func1** which called **func2**. The function argument types and values are printed as are their file and line numbers. If this info is missing it means the code was not compiled with debugging turned on.

What is here?

Show the source code corresponding to the current execution point:

```
(gdb) list
20          func1(p);
21          return 0;
22      }
}
```

Check values of variables:

```
(gdb) print p
$1 = (int *) 0x0
```

The type and value are printed. The $1 is now a variable that can be used later.

Going elsewhere.

You can go up and down the call stack with the `up` and `down` commands. This helps get to the real root cause:

```
(gdb) up
#1  0x080483a5 in func1(int*) (p=0x0) at crash.cc:7
7           func2(p);
```

```
(gdb) up
#2  0x080483ca in main (argc=1, argv=0xbffffae4) at crash.cc:12
20          func1(p);
```

We could also have jumped to that frame via `frame 2` command. If the frame is not specified the current one will be printed.

Setting Break Points

Break points tell GDB to stop execution at some point in the program.

⇒ Break when named function is run:

```
(gdb) break func1
Breakpoint 1 at 0x804839a:  file crash.cc, line 7.
```

```
(gdb) run
Breakpoint 1, func1(int*) (p=0x0) at crash.cc:7
7           func2(p);
```

⇒ Break at file:line

```
(gdb) break crash.cc:11
Breakpoint 2 at 0x8048384:  file crash.cc, line 11.
```

```
(gdb) run
Breakpoint 2, func0(int*&) (p=@0xbffffa24) at crash.cc:11
11          p = new int;
```

Caveats with Break Points

There are some things to keep in mind about break points.

1. The code must be linked (see below)
2. When breaking in a class method, the full name, including arguments, must be used. It helps to put the partial name in single quotes and hit tab:

```
(gdb) break 'MyClass::getD<TAB>'
```

```
(gdb) break 'MyClass::getData(int id)'
```

Setting Watch Points

Setting a watch point will cause GDB to halt execution when some expression changes.

```
(gdb) break main
Breakpoint 5 at 0x804850a:  file crash.cc, line 18.
```

```
(gdb) run
Breakpoint 5, main (argc=1, argv=0xbffffa84) at crash.cc:18
18          int* p=0;
```

```
(gdb) watch p
Hardware watchpoint 6:  p
```

```
(gdb) continue
Continuing.
Hardware watchpoint 6: p
Old value = (int *) 0xbffffa84
New value = (int *) 0x8049830
func0(int*&) (p=@0xbffffa24) at crash.cc:12
12          func1(p);
```

Watch Points Caveats

Hardware Watchpoints are handled by the CPU directly and thus incur almost no slowdown. They are limited in number and in the amount of memory they can watch.

Software Watchpoints are handled by GDB and don't have the limitation of those in hardware. However they will slow down the execution of the program by orders of magnitude. This makes them essentially unusable.

Sometimes setting a watch point on a class member is difficult.

```
(gdb) print fConst
$40 = (double*) 0xdcf6260
```

```
(gdb) print &fConst
$20 = (double**) 0xb960578
```

```
(gdb) awatch *0xdcf6260
```

```
(gdb) awatch *0xb960578
```

Controlling Execution

Once execution has been broken there are a number of ways to control it.

`step` or `s` step to the next executable statement. This lets one step into a function call.

`next` or `n` step to the next executable statement in current frame. This avoids stepping into a function call.

`continue` or `c` continue to the next break or watch point or until the program execution is terminated.

`delete` or `d` delete all break and watch points. Given an optional number, delete just that point.

Hitting `ENTER` repeats the last command.

Handling Dynamically Loaded Libraries

Before code in a library can be accessed by GDB, it must be linked into the running process. However, it is common for libraries to be linked during running. To handle this you must break in `main()`:

1. Set a **break** in `main`
2. **run**
3. **break** by "file:line"
4. **run** to final break

For Further Reading I

- GDB manual. On the web at:
  `http://www.gnu.org/software/gdb/documentation/`
- Two GUI GDB front ends:
  - GVD, `http://libre.act-europe.fr/gvd/`
  - DDD, `http://www.gnu.org/software/ddd/`
- For fun: TCSH considered harmfull,
  `http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/`