USCC
gdb Debugging Tutorial

By: Sanjay Madhav
University of Southern California

Introduction

If you are running USCC on Mac OS X, you will be able to debug directly in Xcode. However, Windows and Linux environments must debug through gdb (or similar) environment. This guide will walk you through how to debug USCC via gdb. Most of these steps will be familiar to you if you've previously debugged in gdb before.

When to Use gdb

As a good rule of thumb, if you encounter any segmentation faults, assertions from LLVM, or anything that terminates USCC unexpectedly, you should immediately turn to gdb. At the very least, this will allow you to isolate precisely where the crash occurred. This method is far more effective than attempting to use cout/printf statements to debug.

Furthermore, although the interface may not be trivial to use, gdb can also be used just like any other visual debugger to set breakpoints and step through the program.

Running USCC in gdb

Suppose you run the following command line:

```
$ ../bin/uscc.exe -a parse06e.usc
```

The resulting output you get is a dreaded segmentation fault:

```
Segmentation fault (core dumped)
```

This means you should run the program inside gdb to isolate the segmentation fault. First, you need to start gdb with the path of the executable as a parameter:

```
$ gdb ../bin/uscc.exe
```

You should see output along the lines of:

```
GNU gdb (GDB) 7.8
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

(Several more lines of information...)

Reading symbols from ../bin/uscc.exe...done.
(gdb)
```

The `(gdb)` prompt is the signal that gdb is ready to accept commands. In order to execute with command line parameters, you should use the `run` command followed by the desired parameters:

```
(gdb) run -a parse06e.usc
```

You should then see some information about the program starting, followed by specific information regarding the segmentation fault:

```
Program received signal SIGSEGV, Segmentation fault.
uscc::parse::Parser::parseAddrOfArrayFactor (this=this@entry=0x28aa4c)
    at ParseExpr.cpp:805
805                     std::cout << *a << std::endl;
(gdb)
```

In this case, gdb states that the segmentation fault occurred on line 805 of `ParseExpr.cpp`.

Examining Variables

Since the line above only has one pointer, it should be fairly clear what is causing the segmentation fault. However, in cases where it's unclear you can use `x` ("examine") command to inspect a particular variable or expression. For example:

```
(gdb) x a
0x0:    Cannot access memory at address 0x0
(gdb) x *a
Cannot access memory at address 0x0
(gdb)
```

This tells us that the variable `a` is set to `0x0` or `nullptr`. So we can infer that the segmentation fault is occurring because `a` is null.

Looking at the Call Stack

Although the cause of the crash in this example is very clear, in more complex cases, it may be helpful to see the call stack. To see the call stack in gdb, use the `bt` ("backtrace") command:

```
(gdb) bt
#0  uscc::parse::Parser::parseAddrOfArrayFactor (this=this@entry=0x28aa4c)
    at ParseExpr.cpp:805
#1  0x0040d59e in uscc::parse::Parser::parseFactor (this=0x28aa4c)
    at ParseExpr.cpp:427
#2  0x0040d775 in uscc::parse::Parser::parseValue (this=this@entry=0x28aa4c)
    at ParseExpr.cpp:400
#3  0x0040de5b in uscc::parse::Parser::parseTerm (this=this@entry=0x28aa4c)
    at ParseExpr.cpp:317
(Any further frames...)
(gdb)
```

The numbers in front of each function on the stack is called a *frame*. You can use this number to switch to a specific frame, and even examine variables in that frame. For example:

```
(gdb) frame 3
#3  0x0040de5b in uscc::parse::Parser::parseTerm (this=this@entry=0x28aa4c)
    at ParseExpr.cpp:317
317             shared_ptr<ASTExpr> value = parseValue();
(gdb) x &value
0x28a548:       0x00000000
(gdb)
```

Breakpoints and Stepping

Now that we know the crash is in `parseAddrOfArrayFactor`, we can also set a breakpoint on this function:

```
(gdb) break uscc::parse::Parser::parseAddrOfArrayFactor
Breakpoint 1 at 0x40fcd0: file ParseExpr.cpp, line 800.
```

Note that you can typically use TAB to auto-complete symbols inside gdb. Keep note of the breakpoint number, because you can use that later to remove the breakpoint.

Once you've set the breakpoint, you can run the program again:

```
(gdb) run -a parse06e.usc
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /cygdrive/c/Users/Madhav/Documents/itp439/uscc/bin/uscc.exe -a
parse06e.usc
[New Thread 6224.0x1974]
[New Thread 6224.0x1a24]

Breakpoint 1, uscc::parse::Parser::parseAddrOfArrayFactor (
    this=this@entry=0x28aa4c) at ParseExpr.cpp:800
800     {
(gdb)
```

Notice that gdb has now stopped at line 800 of `ParseExpr.cpp`, corresponding to where we set the breakpoint.

You can now step line by line using the `s` (or step) command:

```
(gdb) s
801             shared_ptr<ASTExpr> retVal;
(gdb) s
802             if (peekAndConsume(Token::Addr))
(gdb) s
uscc::parse::Parser::peekAndConsume (this=this@entry=0x28aa4c,
    desired=desired@entry=uscc::scan::Token::Addr) at Parse.cpp:167
167     {
(gdb)
```

The `s` command performs a "step into," so it will jump into functions. If you instead want to "step over" a line, you can use the `n` command. Also, if you are inside a function that you do not care to debug, you can use `finish` to step out of it:

```
(gdb) finish
Run till exit from #0  uscc::parse::Parser::peekAndConsume (
    this=this@entry=0x28aa4c, desired=desired@entry=uscc::scan::Token::Addr)
    at Parse.cpp:167
0x0040fcfc in uscc::parse::Parser::parseAddrOfArrayFactor (
    this=this@entry=0x28aa4c) at ParseExpr.cpp:802
802             if (peekAndConsume(Token::Addr))
Value returned is $1 = false
(gdb)
```

Note that it will tell you what the return value was. In this case, we can probably infer that we will hit the breakpoint at the start of `parseAddrOfArrayFactor` several times, since this function may be speculatively executed frequently. So let's remove the breakpoint (which you may recall, was breakpoint 1) using the `delete` command:

```
(gdb) delete 1
(gdb)
```

We can now instead add a breakpoint on line 803 of `ParseExpr.cpp`, which should only be hit in the event that the `Addr` token is encountered:

```
(gdb) break ParseExpr.cpp:803
Breakpoint 2 at 0x40fd10: file ParseExpr.cpp, line 803.
(gdb)
```

Now you can use the `c` ("continue") command to continue to the next breakpoint:

```
(gdb) c
Continuing.

Breakpoint 2, uscc::parse::Parser::parseAddrOfArrayFactor (
    this=this@entry=0x28aa4c) at ParseExpr.cpp:805
805                     std::cout << *a << std::endl;
(gdb)
```

We've now returned to the exact point where the program will crash, so if you "continue" here, the program will seg fault again:

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
uscc::parse::Parser::parseAddrOfArrayFactor (this=this@entry=0x28aa4c)
    at ParseExpr.cpp:805
805                     std::cout << *a << std::endl;
(gdb)
```

Finally, to exit gdb you can use the `quit` command:

```
(gdb) quit
```

Conclusion

Although gdb is definitely not as easy to use as a visual debugger, it has all of the same functionality. There is also quite a bit of documentation and resources available online, so you can consult these resources if want to use features not discussed in this tutorial (such as conditional breakpoints).