dbp

Basic Assembler Debugging with GDB

or - you thought you only needed to know C/C++?

by Patrick Horgan

(Back to debugging.)
Click to show Table of Contents

Who is this for?

This article is aimed straight at people who will be doing debugging with gdb on a linux box. There will be information useful in general to anyone that debugs in C/C++ and needs to drop down to assembler, but the tools and information are definately linux-centric. The assembler language used will be x86 with AT&T syntax. I assume you know C and or C++, that you can understand hexadecimal numbers, that you can run things from the command line, that you have a normal development environment using GNU tools installed, and many other things. In other words this is an intermediate level, not beginner level tutorial. Additionally this is not intended to teach you the things you would need to know to write assembler, but rather the things you would need to know to understand assembler you see in the debugger.

Tell me it's not so! Why do I need to debug assembler?

There are several good reasons that you need to debug at the assembler level.

- A special purpose register changes in a way that affects your program and you don't know why. Register contents and the ops that change them are only visible from assembler.
- Your C/C++ code looks entirely correct, but the output is unexpected. You need to drop to assembler to see the code generated for your C/C++ to find out what's going on.
- Your code looks efficient, but it's not.
- You're debugging through a call into someone else's code, maybe in a library, you don't have the source code, but you still need to know what's going on.
- Or generally, you want to *really* know what's going on. In C or C++ things happen behind the scenes and you need to know what they are. Particularly in C++, details of how your compiler implements

inheritance, construction, and destruction are *only* visible in assembler, and you will not be able to do your job as well unless you are familiar with what's going on at the assembler level.

But don't panic, it's not so hard and debugging is debugging!

Your debugging skills will all translate to assembler programming, instead of s for step and n for next, it's si for step instruction and ni for next instruction.

You will have to learn a little about assembler

It's what the machine speaks after all.

But look out for the syntax wars

To make it more difficult, there's two types of syntax used to represent the exact same machine code. Intel syntax is used by everyone that came up through the PC/Microsoft world, and AT&T syntax is used by everyone that came up through the Unix/Linux world. Sigh. GDB (and inline assembler in C/C++ with GCC) use the AT&T syntax by default, since it comes out of the unix/linux tradition, so that's what I'll talk about in this article. Many of the tutorials about assembler that you'll find on the internet will use the intel syntax, because most of them are about Windows boxes. Articles about assembler on linux boxes will use AT&T syntax. And of course, it's a matter of probability that you are most likely using an Intel processor on your box, and the documentation for their processors use the Intel syntax of course. You should learn both. The Wikipedia article, x86 assembly language has a good summary of the differences in the two syntaxes. I detect a bit of a bias in the authors toward the intel syntax, for example listing many programs that support that syntax but not listing ones that support the AT&T syntax, but don't let that bother you.

You will be a better programmer if you avoid unreasoned bias

The computer world is full of unreasoned biases. People are passionate about everything from editors to operating systems. You might try to make an effort to avoid nourishing your own unreasoned biases. Most people will go with whatever they learned first. It doesn't make that choice better or worse, and you trying to beat them over the head with the reasons you think your choice is better will only point out that you are annoying. If you step above the fray and realize that most of these choices are perfectly valid, it will make you less pedantic, more open to new things, and a better programmer and human being. You'll also gain the perspective to see when there is really a difference between one choice and another, and to decide if that difference matters to you. 'Nuff said.

Go here to quickly learn assembler

If you don't know assembler at all, an extraordinary resource is Programming From The Ground Up by Jonathan Bartlett which teaches assembler programming on linux. I could not possibly recommend it highly enough. It begins with the assumption that you know little about programming and takes you to a fairly high level of expertise.

Another resource which assumes lots of knowlege but gives a whirlwind overview of the modern registers and their use is a nice white paper from Intel, written by David Kreitzer and Max Domeika; Ensuring Development Success by Understanding and Analyzing Assembly Language For IA-32 and Intel® 64 Instruction Set Architecture. Interestingly enough, it uses AT&T syntax for the assembler, since the article is for people using Intel's professional assembler for Linux.

So I'm not going to teach you assembler

Instead I'll jump right into using gdb with first a simple assembler program, then with a series of C/C++ programs. As we go, I'll teach you a bit about using other tools like nm and objdump, I'll teach you a little about how programs are started in linux, and I'll teach you a bit about what C/C++ stack frames are and how they look from assembler.

Our first program to debug is in assembler!

Here's our source

.section .text .globl _start _start: movl $1, %eax # tell kernel use system call #1 for exit movl $0xff, %ebx # exit with status 255 int $0x80 # interrupt 80 is do system call in %eax

Every executable file on a linux system must have a symbol named _start. That's the place that the system will hand control to in the program. We use .globl _start as a signal to the assembler and the loader that this will be a globally visible symbol, and then we place _start: in the program. Something that ends with a colon, (:), is called a symbol, and this one will be exported by the loader because we said it was global. It will refer to the address of whatever comes right after the declaration of the symbol. In this case, the next thing after _start: is movl $1, %eax, an instruction to tell the processor to move the value 1 into the %eax register. The address of that instruction will be associated with the global symbol _start:.

All this program will do is call the linux system call #1 which says to exit with the status value in register %ebx. That's why we have the line movl $0xff, %ebx. It moves the literal value 0xff (255) into %ebx. Finally we call interrupt number 128, (in hexadecimal 0x80), which is handled by the operating system handler for that interrupt. That handler does system calls for you. It's the interface between programs and the operating system.

Lets build it and run it.

Save a copy of the program as exit.s, and we'll assemble and link it.

$ as --gstabs+ exit.s -o exit.o $ ld exit.o -o exit

The assembler argument --gstabs+ tells the assembler that we want it to save debugging information that will let gdb print the line of assembler source code that corresponds to each assembler instruction. Run it and check that the return code is really returned to us like this.

$ ./exit $ echo $? 255 $

$? is the shell symbol that means the completion/error code returned by the last program. In this case, we expect it to be 255, since that's the value we put in %ebx, and if you try it, you'll see that indeed that's what happens.

Now let's run it in the debugger

$ gdb exit Reading symbols from /home/patrick/src/asm/progGrndUp/exit...done. (gdb) b _start Breakpoint 1 at 0x8048054: file exit.s, line 4. (gdb) run Starting program: /home/patrick/src/asm/progGrndUp/exit Breakpoint 1, _start () at exit.s:4 4 movl $1, %eax # tell kernel use system call #1 for exit (gdb) s 5 movl $0xff, %ebx # exit with status 255 (gdb) s 6 int $0x80 # interrupt 80 is do system call in %eax (gdb) s Program exited with code 0377. (gdb) q $

gdb exit tell the system to run gdb and to tell it that the program we want it to debug is exit. It starts up and tells us that it's done reading symbols from our program and give us the gdb command prompt, (gdb). Being kind obliging folks, we give gdb a command, b _start, which tells gdb that we want it to put a breakpoint at the address with the symbol _start associated with it. Next we tell gdb to run and after telling us that it's starting the program, the next news is that execution has been halted, at our request because it hit the breakpoint at _start.

Then gdb shows us the source code associated with _start and waits for us to tell it what to do. We do a series of s (single step) commands until we get to the end of the program. Once the interrupt call is made, system call #1 is run and we exit. gdb reports the exit code is 0377 which is octal for decimal 255 ($3 \times 8^2 + 7 \times 8^1 + 7 \times 8^0 = 3 \times 64 + 7 \times 8 + 7 \times 1 = 192 + 56 + 7 = 255$).

Wait a minute, why s and not si?

Some of you may be wondering why we can use step (s) instead of step instruction (si). If you're debugging C/C++, the source file is the C/C++ and a single step steps from one source line to the next. If you want to step through the assembler, you have to use si. In assembler it's the same. One step through the source is an assembler step. You can use si if you want, but you don't have to.

So how does gdb know about the source code?

Remember the argument to the assemblers, --gstabs+? It caused some information to be saved inside the executable. We can, quite easily, see what it is.

$ objdump --stabs exit exit: file format elf32-i386 Contents of .stab section: Symnum n_type n_othr n_desc n_value n_strx String -1 HdrSym 0 5 0000002a 1 0 SO 0 0 08048054 8 /home/patrick/src/asm/progGrndUp/ 1 SO 0 0 08048054 1 exit.s 2 SLINE 0 4 08048054 0 3 SLINE 0 5 08048059 0 4 SLINE 0 6 0804805e 0

This says that the file was built from the exit.s found in the named directory, and gives an association for each line of code between the line of code from the source, and the memory address of the executable. If we're at address 0x804805e, we know that's line six from the file, the line with the int $0x80.

We can see the code in the file too!

objdump can do a lot of other things for you, from the command line type man objdump for more information, but here we'll use it to get a disassembly of the file exit.

$ objdump -d exit exit: file format elf32-i386 Disassembly of section .text: 08048054 <_start>: 8048054: b8 01 00 00 00 mov $0x1,%eax 8048059: bb ff 00 00 00 mov $0xff,%ebx 804805e: cd 80 int $0x80 $

The -d argument to objdump means disassemble, and we can see that the disassembly matches the original source file.

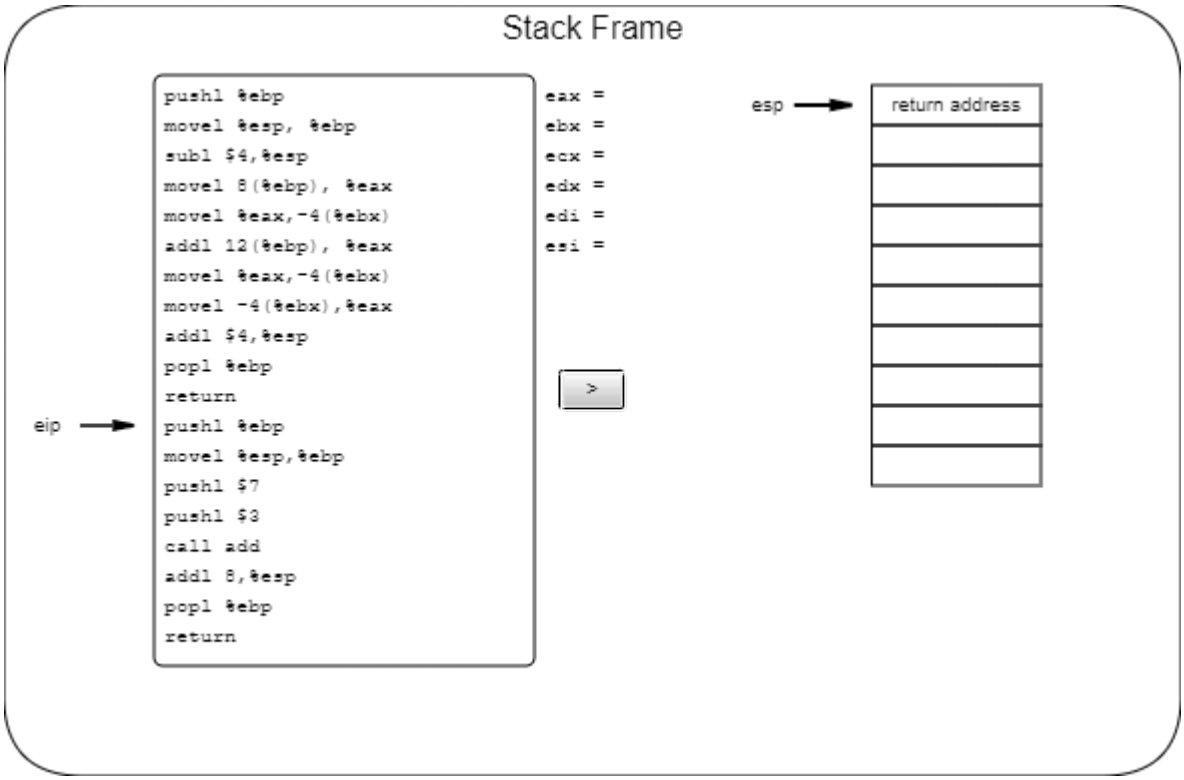
Now lets talk about the C calling conventions

Here's the source of a simple C file.

int add(int num1,int num2) { int a=num1; a=num1+num2; return a; } main() { return add(3,7); }

And here's an application to trace it in assembler

Below is an application (written in javascript and drawn on an html5 canvas), that will let you step through the the assembler that corresponds to the above C program one step at a time. When you're done, if you want to run it again, just refresh the page. I didn't make any attempt to model how we got into main, or where we go to after exiting main. There's a whole tutorial for that, Linux x86 Program Start Up.



C variables are on the *stack*

You know as a C/C++ programmer that on entry, your variables are on the stack. Now you're going to learn exactly what that means. Your stack is just memory that you have permission to write to. (There are details about memory and mapping and virtual vs. real that I will not talk about at all in this tutorial, because they are not pertinent).

We start in main

The line of assembler the instruction pointer (%eip) is pointing at is the first instruction of main. Each time you click, you can see what happens to the stack pointer, the registers, and the instruction pointer.

The preamble - On Entering A Function This You Will Do

According to the C calling convention, the first thing to do upon entry to a function is to save %ebp, the caller's base pointer, by pushing it onto the stack. Then the next thing is to copy the stack pointer into the base pointer so that you can find your arguments after the stack pointer changes. After that you would adjust your stack to make room for any locals, but main doesn't have any so you don't see that here.

Calling another function

So in a minute we're going to call add, so we need to push its arguments on to the stack. The convention says that when calling a function you push its arguments onto the stack in reverse order. Go ahead and click the button and watch them get pushed onto the stack. Each time something gets pushed on the stack, the stack pointer first moves down in memory, and then the item is moved to that memory location. The stack pointer always points at the last thing that got pushed. For simplicity, everything in this program that goes on the stack is 4 bytes long, so each slot you see is a 4 byte slot. It's also possible to pushw for 2 byte values, and pushb for one byte values. It's also important to remember that the stack grows downward in memory.

As soon as you click the button to execute the call, the instruction pointer moves to the new function, and you'll see that the return address was automatically pushed onto the stack.

Entering add() - Preamble again

In the new function we save main's %ebp. Now we can copy our stack pointer into %ebp. Now %ebp functions as *our* base pointer and we can access our arguments above in the stack with positive offsets, and we can access locals (there will be one for the local a) with negative offsets. Next we add 4 to our stack pointer to make room for a.

Preamble done - on to our business

a=num1;

We move the first argument off of the stack into %eax, and then save it into the local storage. That corresponds to the line of C, a=num1.

a=num1+num2;

The next line is a=num1+num2. That corresponds to the next couple of lines where num2 is pulled off the stack and added to %eax. Then it's saved back into our local on the stack. Then right away we copy it back from the stack to %eax! What's going on here? This is typical of unoptimized code generated by compilers. It looks silly because everything is generated by automated rules. When you turn optimization on it will find all of that stuff and clean it up.

Epilogue

Like the code prologue, there's a standard way to exit a function. We have to undo the things the prologue does, we adjust the stack, pop the caller's base pointer off the stack so that it points in the right place and with the return address now on the top of the stack, we can return. We have the return value in %eax where the convention says that we're supposed to put it, and so we adjust the stack to get rid of our variable, and pop main's %ebx. Notice that nothing takes the values out of stack memory, the pointer just adjusts to free the memory. Later if something else got pushed onto the stack it would overwrite the values, but otherwise they're just sitting there. Finally, the return pops the return address off of the stack into %eip the instruction pointer, and execution returns to main right after the call.

Back in main - now we see his epilogue.

Main pops the %ebx that belonged to main's caller off the stack and returns to them. Why didn't they have to set up the return value? Well, the return value goes in %eax, and the value we are returning is the same thing that add returned to us. It's already in %eax, so we can just return.

f(int i) { int a=3; i=4+i+a; return i; } int main() { return f(2); }

Now let's build the real program and look at it in gdb

Save it as test.c and build it like this,

gcc -ggdb -o test test.c

A few things you need to know

We're going to step through it in assembler, and there are a few things I want to point out before I start. First, almost always, a line of C will correspond to several lines of assembler. I tell gdb, set disassemble-next-line on which makes it print the next line of assembler that will be executed the same way you're used to it printing out the next line of C that will be executed. Since several lines of assembler correspond to one line of C, everytime a line of C is printed, you'll see several lines of assembler like this

0x080483a4 5 i=4+i+a; => 0x080483a1 <f+13>: 8b 45 08 mov 0x8(%ebp),%eax 0x080483a4 <f+16>: 83 c0 04 add $0x4,%eax 0x080483a7 <f+19>: 03 45 fc add -0x4(%ebp),%eax 0x080483aa <f+22>: 89 45 08 mov %eax,0x8(%ebp)
i=4+i+a;
mov 0x8(%ebp),%eax add $0x4,%eax add -0x4(%ebp),%eax mov %eax,0x8(%ebp)

<=--=>

This tells you that line 5 from the C source compiled to four lines of assembler.

0x080483a4 5 i=4+i+a; => 0x080483a1 <f+13>: ... 0x080483a4 <f+16>: ... ... (gdb) si 0x080483a4 5 i=4+i+a; 0x080483a1 <f+13>: ... => 0x080483a4 <f+16>: ... ...

I step through the assembler with si (step instruction), and you'll see that gdb will tell you which line of the assembler will be executed next by marking it on the left with =>. Each time you tell gdb to step to the next instruction, the => will move down one, but the line of C won't change until you si off the bottom of the assembler lines that correspond to the line of C.

When you start doing mixed assembler and C/C++ debugging there will be times you, by habit, type n (next) or s (step) and go to the next line of C when you meant to type ni (next instruction) or si (step instruction). It will be frustrating. At the worst you will have just done a lot of work to set up to see the smoking gun on a rare, hard to trace, bug and then you'll have to start over. Be careful. I'll do several step immediate before the C line will change. I'll only type si for the first one, I'll just press enter to take advantage of the way gdb will repeat the last instruction everytime you press enter.

Load it up in gdb

$ gdb test Reading symbols from /home/patrick/src/asm/test...done. (gdb) b main Breakpoint 1 at 0x80483b8: file test.c, line 12. (gdb) set disassemble-next-line on

To start I load the program into the debugger, tell gdb I want to see the assembler that's coming up whenever we stop, and set a breakpoint on main. There's a surprise about the breakpoint.

Run until the breakpoint

(gdb) r Starting program: /home/patrick/src/asm/test Breakpoint 1, main () at test.c:12 12 return f(2); => 0x080483b8 <main+6>: c7 04 24 02 00 00 00 movl $0x2,(%esp) 0x080483bf <main+13>: e8 d0 ff ff ff call 0x8048394 <f> (gdb) disassemble Dump of assembler code for function main: 0x080483b2 <+0>: push %ebp 0x080483b3 <+1>: mov %esp,%ebp 0x080483b5 <+3>: sub $0x4,%esp => 0x080483b8 <+6>: movl $0x2, (%esp) 0x080483bf <+13>: call 0x8048394 <f> 0x080483c4 <+18>: leave 0x080483c5 <+19>: ret End of assembler dump.

Ok, I ran it, and hit the breakpoint. gdb told us both that we'd broken at line 12, which had the return f(2);, and that that line corresponded to two lines of assembler, the first to move a literal 2 onto the stack, and the other to call f. I wanted to see the bigger picture, so I typed disassemble. By default, the current function is disassembled. In the dump of the assembler code for the function main, you see that they still mark the line at 0x080483b8 as being the next line that will execute, but now you can see that we've already executed the three preamble lines. If

we'd wanted to step through those, we would have to disassemble main first, and then set a breakpoint on the address of the first push, like b *0x80483b2.

Let's look at f too

(gdb) disassemble f Dump of assembler code for function f: 0x08048394 <+0>: push %ebp 0x08048395 <+1>: mov %esp,%ebp 0x08048397 <+3>: sub $0x10,%esp 0x0804839a <+6>: movl $0x3,-0x4(%ebp) 0x080483a1 <+13>: mov 0x8(%ebp),%eax 0x080483a4 <+16>: add $0x4,%eax 0x080483a7 <+19>: add -0x4(%ebp),%eax 0x080483aa <+22>: mov %eax,0x8(%ebp) 0x080483ad <+25>: mov 0x8(%ebp),%eax 0x080483b0 <+28>: leave 0x080483b1 <+29>: ret End of assembler dump.

Now I want to see what the function f will look like before we go into it, so I type disassemble f to ask gdb to do it.

(gdb) si 0x080483bf 12 return f(2); 0x080483b8 <main+6>: c7 04 24 02 00 00 00 movl $0x2,(%esp) => 0x080483bf <main+13>: e8 d0 ff ff ff call 0x8048394 <f>

Notice that we're still on the same C instruction but now we're on the next assembler instruction. It shows the call to f is the next this, so when I press enter again, we'll be in f().

Now we're in f

(gdb) f (i=2) at test.c:3 3 { => 0x08048394 <f+0>: 55 push %ebp 0x08048395 <f+1>: 89 e5 mov %esp,%ebp 0x08048397 <f+3>: 83 ec 10 sub $0x10,%esp

The opening brace of a function corresponds to the preamble!

(gdb) 0x08048395 3 { 0x08048394 <f+0>: 55 push %ebp => 0x08048395 <f+1>: 89 e5 mov %esp,%ebp 0x08048397 <f+3>: 83 ec 10 sub $0x10,%esp (gdb) 0x08048397 3 { 0x08048394 <f+0>: 55 push %ebp 0x08048395 <f+1>: 89 e5 mov %esp,%ebp => 0x08048397 <f+3>: 83 ec 10 sub $0x10,%esp

Past the preamble

It took three si to get through the preamble the next step will bring us to the first line of C in the function.

(gdb) 4 int a=3; => 0x0804839a <f+6>: c7 45 fc 03 00 00 00 movl $0x3,-0x4(%ebp) (gdb)

Obviously, a lives on the stack at -4 from our base pointer. That would make it the first local variable, as expected.

=> 0x080483a1 <f+13>: 8b 45 08 mov 0x8(%ebp),%eax 0x080483a4 <f+16>: 83 c0 04 add $0x4,%eax 0x080483a7 <f+19>: 03 45 fc add -0x4(%ebp),%eax 0x080483aa <f+22>: 89 45 08 mov %eax,0x8(%ebp) (gdb)

We move the argument i off of the stack and into %eax, where we'll do the calculation.

0x080483a4 5 i=4+i+a; 0x080483a1 <f+13>: 8b 45 08 mov 0x8(%ebp),%eax => 0x080483a4 <f+16>: 83 c0 04 add $0x4,%eax 0x080483a7 <f+19>: 03 45 fc add -0x4(%ebp),%eax 0x080483aa <f+22>: 89 45 08 mov %eax,0x8(%ebp) (gdb) 0x080483a7 5 i=4+i+a; 0x080483a1 <f+13>: 8b 45 08 mov 0x8(%ebp),%eax 0x080483a4 <f+16>: 83 c0 04 add $0x4,%eax => 0x080483a7 <f+19>: 03 45 fc add -0x4(%ebp),%eax 0x080483aa <f+22>: 89 45 08 mov %eax,0x8(%ebp) (gdb) 0x080483aa 5 i=4+i+a; 0x080483a1 <f+13>: 8b 45 08 mov 0x8(%ebp),%eax 0x080483a4 <f+16>: 83 c0 04 add $0x4,%eax 0x080483a7 <f+19>: 03 45 fc add -0x4(%ebp),%eax => 0x080483aa <f+22>: 89 45 08 mov %eax,0x8(%ebp) (gdb)

We added 4 to it, added our saved local variable a to it, and then saved it back into the input argument slot 8 above the base pointer.

6 return i; => 0x080483ad <f+25>: 8b 45 08 mov 0x8(%ebp),%eax (gdb)

return just means to put it into %eax. It was just there, so it's a silly instruction moving something somewhere it already is, but that's what optimizers are for.

7 } => 0x080483b0 <f+28>: c9 leave 0x080483b1 <f+29>: c3 ret (gdb)

The closing brace corresponds to the epilogue. If you haven't seen the leave instruction before, it does the same thing as

movl %ebp, %esp pop %ebp

We expect the return to load the address of the end of main in so let's try it.

Heading back to main

0x080483b1 in f (i=9) at test.c:7 7 } 0x080483b0 <f+28>: c9 leave => 0x080483b1 <f+29>: c3 ret (gdb) main () at test.c:13 13 } => 0x080483c4 <main+18>: c9 leave 0x080483c5 <main+19>: c3 ret

Yep, we're back in main.

(gdb) 0x080483c5 in main () at test.c:13 13 } 0x080483c4 <main+18>: c9 leave => 0x080483c5 <main+19>: c3 ret (gdb) __libc_start_main (main=0x80483b2 <main>, argc=1, ubp_av=0xbffff774, init=0x80483d0 <__libc_csu_init>, fini=0x8048430 <__libc_csu_fini>, rtld_fini=0x11ea50, stack_end=0xbffff76c) at libc-start.c:258 258 libc-start.c: No such file or directory. in libc-start.c

We backed out of main and into the function that called main, __libc_start_main. We don't have the source because Ubuntu didn't install the source to go with libc.

Now we're all done let the program finish

=> 0x00145e37 <__libc_start_main+231>: 89 04 24 mov %eax,(%esp) 0x00145e3a <__libc_start_main+234>: e8 61 8c 01 00 call 0x15eaa0 <exit> (gdb) c Continuing. Program exited with code 011. (gdb)

So I just type c (continue) and let the program exit, since we don't have any need to debug functions in libc.

int array1[]={ 0,1,2,3,4,5 }; int main() { int index, i=array1[1]; array1[4]=40; index=3; array1[index]=array1[index+1]; }

What does array access look like in assembler?

We'll write a simple C program to access elements of an array of ints so that you'll see what sort of assembler code corresponds to your C. We save the program as array1.c and then

gcc -ggdb -o array1 array1.c

We'll load it up in gdb and see what it looks like.

More to come tutorial in progress

(Back to debugging.)