

Linux x86 Program Start Up

or - How the heck do we get to main()?

by Patrick Horgan

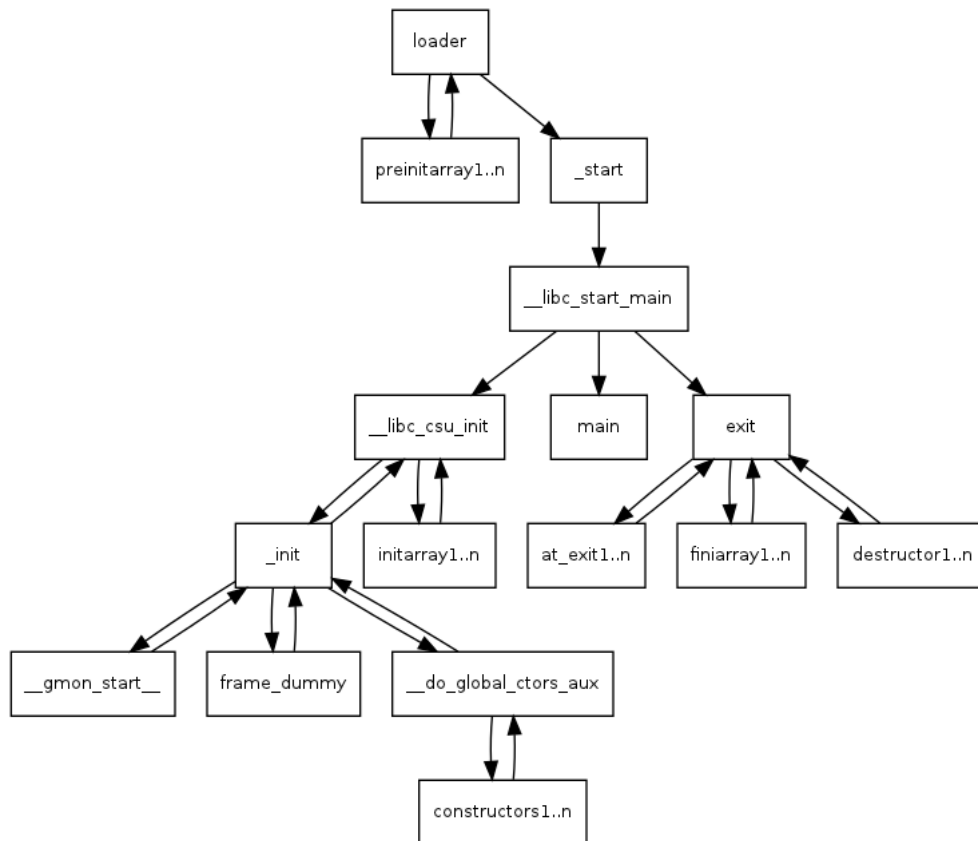
([Back to debugging.](#))

[Click to show Table of Contents](#)

Who's this for?

This is for people who want to understand how programs get loaded under linux. In particular it talks about dynamically loaded x86 ELF files. The information you learn will let you understand how to debug problems that occur in your program before main starts up. Everything I tell you is true, but some things will be glossed over since they don't take us toward our goal. Further, if you link statically, some of the details will be different. I won't cover that at all. By the time you're done with this though, you'll know enough to figure that out for yourself if you need to.

This is what we'll cover (pretty picture brought to you by dot - filter for drawing directed graphs)



When we're done, you'll understand this.

How did we get to main?

We're going to build the simplest C program possible, an empty main, and then we're going to look at the disassembly of it to see how we get to main. We'll see that the first thing that's run is a function linked to every program named `_start` which eventually leads to your program's main being run.

```
int
main()
{
}
```

Save a copy of this as `prog1.c` if you want, and follow along. The first thing I'll do is to build it like this.

```
gcc -ggdb -o prog1 prog1.c
```

Saving to Security
Linux x86 Program Start Up

Before we try to debug a later version of this (prog2), in gdb, we're going to look at a few things about how our program starts up. I'm going to show the output of `objdump -a prog2`, but I'm not going to show it in the order it would be dumped by `objdump`, but rather in the order it would be executed. (But you're perfectly welcome to dump it yourself. Something like `objdump -d prog1 >prog1.dump` will save a copy for you, and then you can use your favorite editor to look at it.) (But RPUVI - Real Programmers Use VI!) N.B. This quip originally said Real Men, because that was the humorous usage that was prevalent when I was a young programmer. Someone (thanks aroman) objected, and after thinking about it, I agreed. The current reader has no idea of the context in my head, and I am always trying to get people to see that we need a lot more women in STEM and that part of the problem is that there is a prevalent unconscious gender bias in STEM that makes it unwelcoming for women. (As mine was unconscious here.) Now we return you to the regularly scheduled tutorial.

But first, how do we get to `_start`?

When you run a program, the shell or gui calls `execve()` which executes the linux system call `execve()`. If you want more information about `execve()` then you can simply type `man execve` from your shell. It will come from section 2 of man where all the system calls are. To summarize, it will set up a stack for you, and push onto it `argc`, `argv`, and `envp`. The file descriptions 0, 1, and 2, (stdin, stdout, stderr), are left to whatever the shell set them to. The loader does much work for you setting up your relocations, and as we'll see much later, calling your preinitializers. When everything is ready, control is handed to your program by calling `_start()` Here from `objdump -d prog1` is the section with `_start`.

`_start` is, oddly enough, where we start

```
080482e0 <_start>:
80482e0: 31 ed          xor    %ebp,%ebp
80482e2: 5e            pop    %esi
80482e3: 89 e1          mov    %esp,%ecx
80482e5: 83 e4 f0      and    $0xfffffff0,%esp
80482e8: 50            push  %eax
80482e9: 54            push  %esp
80482ea: 52            push  %edx
80482eb: 68 00 84 04 08 push  $0x8048400
80482f0: 68 a0 83 04 08 push  $0x80483a0
80482f5: 51            push  %ecx
80482f6: 56            push  %esi
80482f7: 68 94 83 04 08 push  $0x8048394
80482fc: e8 c3 ff ff ff call   80482c4 <__libc_start_main@plt>
8048301: f4            hlt
```

xor of anything with itself sets it to zero. so the `xor %ebp,%ebp` sets `%ebp` to zero. This is suggested by the ABI (Application Binary Interface specification), to mark the outermost frame. Next we pop off the top of the stack. On entry we have `argc`, `argv` and `envp` on the stack, so the pop makes `argc` go into `%esi`. We're just going to save it and push it back on the stack in a minute. Since we popped off `argc`, `%esp` is now pointing at `argv`. The `mov` puts `argv` into `%ecx` without moving the stack pointer. Then we `and` the stack pointer with a mask that clears off the bottom four bits. Depending on where the stack pointer was it will move it lower, by 0 to 15 bytes. In any case it will make it aligned on an even multiple of 16 bytes. This alignment is done so that all of the stack variables are likely to be nicely aligned for memory and cache efficiency, in particular, this is required for SSE (Streaming SIMD Extensions), instructions that can work on vectors of single precision floating point simultaneously. In a particular run, the `%esp` was `0xbffff770` on entry to `_start`. After we popped `argc` off the stack, `%esp` was `0xbffff774`. It moved up to a higher address (putting things on the stack moves down in memory, taking things off moves up in memory). After the `and` the stack pointer is back at `0xbffff770`.

Now set up for calling `__libc_start_main`

So now we start pushing arguments for `__libc_start_main` onto the stack. The first one, `%eax` is garbage pushed onto the stack just because 7 things are going to be pushed on the stack and they needed an 8th one to keep the 16-byte alignment. It's never used for anything. `__libc_start_main` is linked in from glibc. In the source tree for glibc, it lives in `csu/libc-start.c`. `__libc_start_main` is specified like

```
int __libc_start_main( int (*main)(int, char **, char **),
                    int argc, char ** ubp_av,
```

```
void (*init) (void),
void (*fini) (void),
void (*rtld_fini) (void),
void (* stack_end));
```

Saving to Security

Linux x86 Program Start Up

So we expect `__start` to push those arguments on the stack in reverse order before the call to `__libc_start_main`.

value	__libc_start_main arg	content
%eax	Don't know.	Don't care.
%esp	void (*stack_end)	Our aligned stack pointer.
%edx	void (*rtld_fini)(void)	Destructor of dynamic linker from loader passed in %edx. Registered by <code>__libc_start_main</code> with <code>__cxat_exit()</code> to call the FINI for dynamic libraries that got loaded before us.
0x8048400	void (*fini)(void)	<code>__libc_csu_fini</code> - Destructor of this program. Registered by <code>__libc_start_main</code> with <code>__cxat_exit()</code> .
0x80483a0	void (*init)(void)	<code>__libc_csu_init</code> , Constructor of this program. Called by <code>__libc_start_main</code> before main.
%ecx	char **ubp_av	argv off of the stack.
%esi	argc	argc off of the stack.
0x8048394	int(*main)(int, char**,char**)	main of our program called by <code>__libc_start_main</code> . Return value of main is passed to <code>exit()</code> which terminates our program.

Stack contents just before call of `__libc_start_main`

`__libc_csu_fini` is linked into our code from glibc, and lives in the source tree in `csu/elf-init.c`. It's our program's C level destructor, and I'll look at it later in the white paper.

Hey! Where's the environment variables?

Did you notice that we didn't get `envp`, the pointer to our environment variables off the stack? It's not one of the arguments to `__libc_start_main`, either. But we know that `main` is called

```
void __libc_init_first(int argc, char *arg0, ...)
{
    char **argv = &arg0, **envp = &argv[argc + 1];
    __environ = envp;
    __libc_init (argc, argv, envp);
}
```

`int main(int argc, char** argv, char** envp)` so what's up?

Well, `__libc_start_main` calls `__libc_init_first`, who immediately uses secret inside information to find the environment variables just after the terminating null of the argument vector and then sets a global variable `__environ` which `__libc_start_main` uses thereafter whenever it needs it including when it calls `main`. After `envp` is established, then `__libc_start_main` uses the same trick and *surprise!* Just past the terminating null at the end of the `envp` array, there's *another* vector, the ELF auxiliary vector the loader uses to pass some information to the process. An easy way to see what's in there is to set the environment variable `LD_SHOW_AUXV=1` before running the program. Here's the result for our `prog1`.

```
$ LD_SHOW_AUXV=1 ./prog1
AT_SYSINFO: 0xe62414
AT_SYSINFO_EHDR: 0xe62000
AT_HWCAP: fpu vme de pse tsc msr pae mce cx8 apic
mtrr pge mca cmov pat pse36 clflush dts
acpi mmx fxsr sse sse2 ss ht tm pbe
AT_PAGESZ: 4096
AT_CLKTCK: 100
AT_PHDR: 0x8048034
AT_PHENT: 32
AT_PHNUM: 8
AT_BASE: 0x686000
AT_FLAGS: 0x0
AT_ENTRY: 0x80482e0
AT_UID: 1002
AT_EUID: 1002
AT_GID: 1000
AT_EGID: 1000
AT_SECURE: 0
AT_RANDOM: 0xbff09acb
```

Isn't that interesting. All sorts of information. The `AT_ENTRY` is the address of `_start`, there's our `userid`, our `effective userid`, and our `groupid`. We know we're a 686, times() frequency is 100, clock-ticks/s? I'll have to investigate this. The `AT_PHDR` is the location of the ELF program header that has information about the location of all the segments of the program in memory and about relocation entries, and anything else a loader needs to know. `AT_PHENT` is just the number of bytes in a header entry. We won't chase down this path just now, since we don't need *that* much information

```
AT_EXECFN:    ./prog1
AT_PLATFORM: i686
```

about the loading of a file to be an effective

Saving to Security Linux x86 Program Start Up

__libc_start_main in general

That's about as much as I'm going to get into the nitty-gritty details of how `__libc_start_main`, but in general, it

- Takes care of some security problems with `setuid` `setgid` programs
- Starts up threading
- Registers the `fini` (our program), and `rtld_fini` (run-time loader) arguments to get run by `at_exit` to run the program's and the loader's cleanup routines
- Calls the `init` argument
- Calls the `main` with the `argc` and `argv` arguments passed to it and with the global `__environ` argument as detailed above.
- Calls `exit` with the return value of `main`

Calling the init argument

The `init` argument, to `__libc_start_main`, is set to `__libc_csu_init` which is also linked into our code. It's compiled from a C program which lives in the glibc source tree in `csu/elf-init.c` and linked into our program. The C code is similar to (but with a lot more `#ifdefs`),

This is our program's constructor

It's pretty important to our program because it's our executable's constructor. "Wait!", you say, "This isn't C++!". Yes that's true, but the concept of constructors and destructors doesn't belong to C++, and preceded C++. Our executable, and every other executable gets a C level constructor `__libc_csu_init` and a C level destructor, `__libc_csu_fini`. Inside the constructor, as you'll see, the executable will look for global C level constructors and call any that it finds. It's possible for a C program to also have these, and I'll demonstrate it before this paper is through. If it makes you more comfortable though, you can call them initializers and finalizers. Here's the assembler generated for `__libc_csu_init`.

```
void
__libc_csu_init (int argc, char **argv, char **envp)
{
    _init ();

    const size_t size = __init_array_end - __init_array_start;
    for (size_t i = 0; i < size; i++)
        (*__init_array_start [i]) (argc, argv, envp);
}
```

What the heck is a thunk?

Not much to talk about here, but I

```
080483a0 <__libc_csu_init>:
80483a0:    55                push   %ebp
80483a1:    89 e5             mov    %esp,%ebp
80483a3:    57                push   %edi
80483a4:    56                push   %esi
80483a5:    53                push   %ebx
80483a6:    e8 5a 00 00 00   call  8048405 <__i686.get_pc_thunk.bx>
80483ab:    81 c3 49 1c 00 00 add    $0x1c49,%ebx
80483b1:    83 ec 1c         sub    $0x1c,%esp
80483b4:    e8 bb fe ff ff   call  8048274 <_init>
80483b9:    8d bb 20 ff ff ff lea   -0xe0(%ebx),%edi
80483bf:    8d 83 20 ff ff ff lea   -0xe0(%ebx),%eax
80483c5:    29 c7            sub    %eax,%edi
80483c7:    c1 ff 02        sar    $0x2,%edi
80483ca:    85 ff            test   %edi,%edi
80483cc:    74 24            je     80483f2 <__libc_csu_init+0x52>
80483ce:    31 f6            xor    %esi,%esi
80483d0:    8b 45 10         mov    0x10(%ebp),%eax
80483d3:    89 44 24 08     mov    %eax,0x8(%esp)
80483d7:    8b 45 0c         mov    0xc(%ebp),%eax
80483da:    89 44 24 04     mov    %eax,0x4(%esp)
80483de:    8b 45 08         mov    0x8(%ebp),%eax
80483e1:    89 04 24        mov    %eax,(%esp)
80483e4:    ff 94 b3 20 ff ff ff call  *-0xe0(%ebx,%esi,4)
80483eb:    83 c6 01        add    $0x1,%esi
```

```

80483ee:    39 fe          cmp    %edi,%esi
80483f0:    72 de          jb     80483d0 <__libc
80483f2:    83 c4 1c      add    $0x1c,%esp
80483f5:    5b            pop    %ebx
80483f6:    5e            pop    %esi
80483f7:    5f            pop    %edi
80483f8:    5d            pop    %ebp
80483f9:    c3           ret

```

Saving to Security

Linux x86 Program Start Up

thought you'd want to see it. The `get_pc_thunk` thing is a little interesting. It's used for position independent code. They're setting up for position independent code to be able to work. In order for it to work, the base pointer needs to have the address of the `GLOBAL_OFFSET_TABLE`. The code had something like:

So,
look

```

__get_pc_thunk_bx:
movl (%esp),%ebx
return

```

```

push %ebx
call __get_pc_thunk_bx
add $_GLOBAL_OFFSET_TABLE_,%ebx

```

closely at what happens. The call to `__get_pc_thunk_bx`, like all other calls, pushes onto the stack the address of the next instruction, so that when we return, the execution continues at the next consecutive instruction. In this case, what we really want is that address. So in `__get_pc_thunk_bx`, we copy the return address from the stack into `%ebx`. When we return, the next instruction adds to it `_GLOBAL_OFFSET_TABLE_` which resolves to the difference between the current address and the global offset table used by position independent code. That table keeps a set of pointers to data that we want to access, and we just have to know offsets into the table. The loader fixes up the address in the table for us. There is a similar table for accessing procedures. It could be really tedious to program this way in assembler, but you can just write C or C++ and pass the `-pic` argument to the compiler and it will do it automagically. Seeing this code in the assembler tells you that the source code was compiled with the `-pic` flag.

But what is that loop?

The loop from `__libc_csu_init` will be discussed in a minute after we discuss the `init()` call that really calls `_init`. For now, just remember that it calls any C level initializers for our program.

init gets the call

Ok, the loader handed control to `_start`, who called `__libc_start_main` who called `__libc_csu_init` who now calls `_init`.

```

08048274 <_init>:
8048274:    55            push   %ebp
8048275:    89 e5         mov    %esp,%ebp
8048277:    53            push   %ebx
8048278:    83 ec 04     sub    $0x4,%esp
804827b:    e8 00 00 00 00 call  8048280 <_init+0xc>
8048280:    5b            pop    %ebx
8048281:    81 c3 74 1d 00 00 add    $0x1d74,%ebx      (.got.plt)
8048287:    8b 93 fc ff ff mov    -0x4(%ebx),%edx
804828d:    85 d2         test   %edx,%edx
804828f:    74 05         je     8048296 <_init+0x22>
8048291:    e8 1e 00 00 00 call  80482b4 <__gmon_start__@plt>
8048296:    e8 d5 00 00 00 call  8048370 <frame_dummy>
804829b:    e8 70 01 00 00 call  8048410 <__do_global_ctors_aux>
80482a0:    58            pop    %eax
80482a1:    5b            pop    %ebx
80482a2:    c9           leave
80482a3:    c3           ret

```

It starts with the regular C calling convention

If you want to know more about the C calling convention, just look at [Basic Assembler Debugging with GDB](#). The short story is that we save our caller's base pointer on the stack and point our base pointer at the top of the stack and then save space for a 4 byte local of some sort. An interesting thing is the first call. Its purpose is quite similar to that call to `get_pc_thunk` that we saw earlier. If you look closely, the call is to the next sequential address! That gets you to the next address as if you'd just continued, but with the side effect that the address is now on the stack. It gets popped into `%ebx` and then used to set up for access to the global access table.

Show me your best profile

Then we grab the address of `gmon_start`. If it's zero then we don't call it, instead we jump past it. Otherwise, we call it to set up profiling. It runs a routine to start profiling, and calls `at_exit` to schedule to write `gmon.out` at the end of execution.



This guy's no dummy! He's been framed!

In either case, next we call `frame_dummy`. The intention is to call `__register_frame_info`, but `frame_dummy` is called to set up the arguments to it. The purpose of this is to set up for unwinding stack frames for exception handling. It's interesting, but not a part of this discussion, so I'll leave it for another tutorial perhaps. (Don't be too disappointed, in our case, it doesn't get run anyway.)

Finally we're getting constructive!

Finally we call `_do_global_ctors_aux`. If you have a problem with your program that occurs before `main` starts, this is probably where you'll need to look. Of course, constructors for global C++ objects are put in here but it's possible for other things to be in here as well.

Let's set up an example

Let's modify our `prog1` and make a `prog2`. The exciting part is the `__attribute__((constructor))` that tells `gcc` that the linker should stick a pointer to this in the table used by `_do_global_ctors_aux`. As you can see, our fake constructor gets run. (`__FUNCTION__` is filled in by the compiler with the name of the function. It's `gcc` magic.)

```
#include <stdio.h>

void __attribute__((constructor)) a_constructor() {
    printf("%s\n", __FUNCTION__);
}

int
main()
{
    printf("%s\n", __FUNCTION__);
}
```

```
$ ./prog2
a_constructor
main
$
```

prog2's `_init`, much the same as `prog1`

In a minute we'll drop into `gdb` and see it happen. We'll be going into `prog2's _init`.

```
08048290 <_init>:
8048290:    55                push   %ebp
8048291:    89 e5             mov    %esp,%ebp
8048293:    53                push   %ebx
8048294:    83 ec 04          sub   $0x4,%esp
8048297:    e8 00 00 00 00    call  804829c <_init+0xc>
804829c:    5b                pop    %ebx
804829d:    81 c3 58 1d 00 00 add   $0x1d58,%ebx
80482a3:    8b 93 fc ff ff ff mov   -0x4(%ebx),%edx
80482a9:    85 d2             test  %edx,%edx
80482ab:    74 05             je    80482b2 <_init+0x22>
80482ad:    e8 1e 00 00 00    call  80482d0 <__gmon_start__@plt>
80482b2:    e8 d9 00 00 00    call  8048390 <frame_dummy>
80482b7:    e8 94 01 00 00    call  8048450 <__do_global_ctors_aux>
80482bc:    58                pop    %eax
80482bd:    5b                pop    %ebx
80482be:    c9                leave
80482bf:    c3                ret
```

As you can see, the addresses are slightly different than in `prog1`. The extra bit of data seems to have shifted things 28 bytes. So, there's the name of the two functions, "`a_constructor`" (14 bytes with null terminator), and "`main`" (5 bytes with null terminator) and the two format strings, "`%s\n`" (2*4 bytes with the newline as 1 character and the null terminator), so $14 + 5 + 4 + 4 = 27$? Hmm off by one somewhere. It's just a guess anyway, I didn't

go and look. Anyway, we're going to break on the call to `__do_global_ctors_aux`, and then single step and watch what happens.



And here's the code that will get called

Just to help, here's the C source code for `__do_global_ctors_aux` out of the gcc source code where it lives in a file `gcc/crtstuff.c`.

```
__do_global_ctors_aux (void)
{
    func_ptr *p;
    for (p = __CTOR_END__ - 1; *p != (func_ptr) -1; p--)
        (*p) ();
}
```

As you can see, it initializes `p` from a global variable `__CTOR_END__` and subtracts 1 from it. Remember this is pointer arithmetic though and the pointer points at a function, so in this case, that -1 backs it up one function

pointer, or 4 bytes. We'll see that in the assembler as well. While the pointer doesn't have a value of -1 (cast to a pointer), we'll call the function we're pointing at, and then back the pointer up again. Obviously, the beginning of this table starts with -1, and then has some number (perhaps 0) function pointers.

Here's the same in assembler

Here's the assembler that corresponds to it from `objdump -d`. We'll go over it carefully so you understand it completely before we trace through it in the debugger.

```
08048450 <__do_global_ctors_aux>:
8048450:    55                push   %ebp
8048451:    89 e5             mov    %esp,%ebp
8048453:    53                push   %ebx
8048454:    83 ec 04         sub   $0x4,%esp
8048457:    a1 14 9f 04 08   mov   0x8049f14,%eax
804845c:    83 f8 ff         cmp   $0xffffffff,%eax
804845f:    74 13             je    8048474 <__do_global_ctors_aux+0x24>
8048461:    bb 14 9f 04 08   mov   $0x8049f14,%ebx
8048466:    66 90             xchg  %ax,%ax
8048468:    83 eb 04         sub   $0x4,%ebx
804846b:    ff d0             call  *%eax
804846d:    8b 03             mov   (%ebx),%eax
804846f:    83 f8 ff         cmp   $0xffffffff,%eax
8048472:    75 f4             jne  8048468 <__do_global_ctors_aux+0x18>
8048474:    83 c4 04         add   $0x4,%esp
8048477:    5b                pop   %ebx
8048478:    5d                pop   %ebp
8048479:    c3                ret
```

First the preamble

There's the normal preamble with the addition of saving `%ebx` as well because we're going to use it in the function, and we also save room for the pointer `p`. You'll notice that even though we save room on the stack for it, we never store it there. `p` will instead live in `%ebx`, and `*p` will live in `%eax`.

Now set up before the loop

It looks like an optimization has occurred, instead of loading `__CTOR_END__` and then subtracting 1 from it, and dereferencing it, instead, we go ahead and load `*(__CTOR_END__ - 1)`, which is the immediate value `0x8049f14`. We load the value in it (remember `$0x8049f14` would mean put that value, without the \$, just `0x8049f14` means the contents of that address), into `%eax`. Immediately, we compare this first value with -1 and if it's equal, we're done and jump to address `0x8048474`, where we clean up our stack, pop off the things we've saved on there and return.

Assuming that there's at least one thing in the function table, though, we also move the immediate value `$0x8049f14`, into `%ebx` which is our function pointer, and then do the `xchg %ax,%ax`. What the heck is that? Well, grasshopper, that is what they use for a nop (No Operation) in 16 or 32 bit x86. It does nothing but take a cycle and some space. In this case, it's used to make the loop (the top of the loop is the subtract on the next line) start on `8048468` instead of `8048466`. The advantage of that is that it aligns the start of the loop on a 4 byte boundary and gives a better chance that the whole loop will fit in a cache line instead of being broken across two. It speeds things up.

And now we hit the top of the loop

Next we subtract 4 from `%ebx` to be ready for the next time through the loop, call the address of in `%eax`, move the next function pointer into `%eax`, and compare it to `-1` to the subtract and loop again.



And finally the epilogue

Otherwise we fall through into our function epilogue and return to `_init`, which immediately falls through into its epilogue and returns to `__libc_csu_init`. Bet you forgot all about him. There's still a loop to deal with there but first--

I promised you we'd go into the debugger with prog2!

So here we go! Remember that `gdb` always shows you the line or instruction that you are *about* to execute.

```
$ !gdb
gdb prog2
Reading symbols from /home/patrick/src/asm/prog2...done.
(gdb) set disassemble-next-line on
(gdb) b *0x80482b7
Breakpoint 1 at 0x80482b7
```

We ran it in the debugger, turned `disassemble-next-line` on, so that it will always show us the disassembly for the line of code that is about to be executed, and set a breakpoint at the line in `_init` where we're about to call `__do_global_ctors_aux`.

```
(gdb) r
Starting program: /home/patrick/src/asm/prog2

Breakpoint 1, 0x080482b7 in _init ()
=> 0x080482b7 <_init+39>:      e8 94 01 00 00 call   0x8048450 <__do_global_ctors_aux>
(gdb) si
0x08048450 in __do_global_ctors_aux ()
=> 0x08048450 <__do_global_ctors_aux+0>:      55      push   %ebp
```

I typed `r` to run the program and hit the breakpoint. My next command to `gdb` was `si`, step instruction, to tell `gdb` to single step one instruction. We've now entered `__do_global_ctors_aux`. As we go along you'll see times when it seems that I entered no command to `gdb`. That's because, if you simply press return, `gdb` will repeat the last instruction. So if I press enter now, I'll do another `si`.

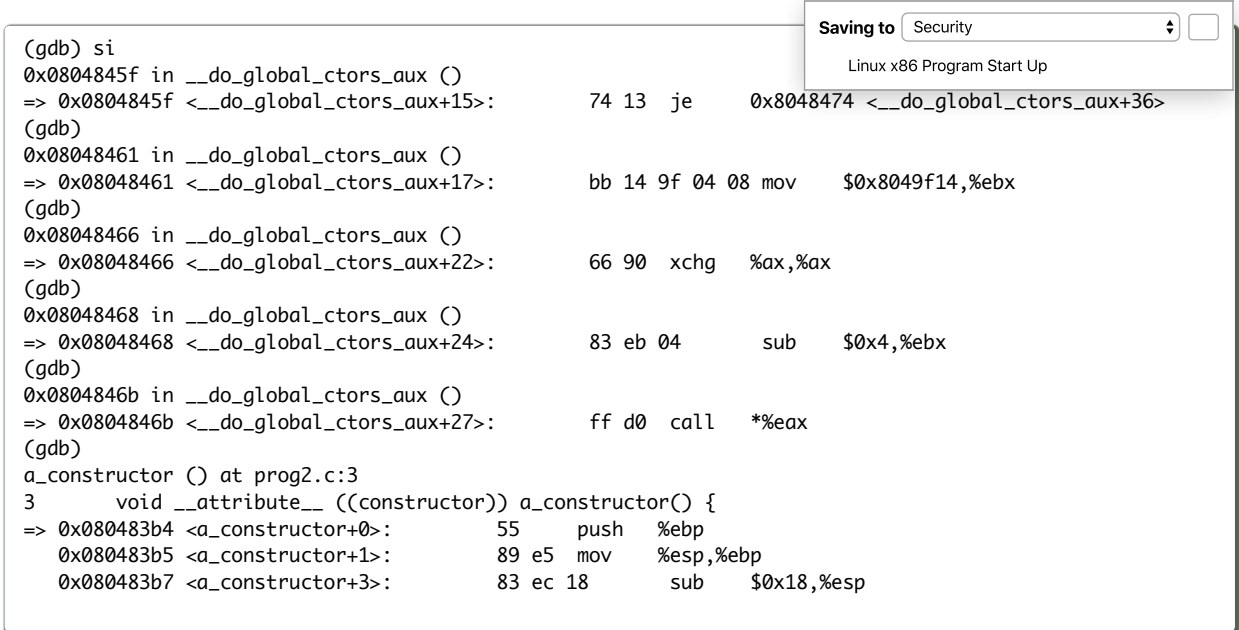
```
(gdb)
0x08048451 in __do_global_ctors_aux ()
=> 0x08048451 <__do_global_ctors_aux+1>:      89 e5   mov    %esp,%ebp
(gdb)
0x08048453 in __do_global_ctors_aux ()
=> 0x08048453 <__do_global_ctors_aux+3>:      53      push  %ebx
(gdb)
0x08048454 in __do_global_ctors_aux ()
=> 0x08048454 <__do_global_ctors_aux+4>:      83 ec 04   sub   $0x4,%esp
(gdb)
0x08048457 in __do_global_ctors_aux ()
```

Ok, now we've finished the preamble, and the real code is about to start.

```
(gdb)
=> 0x08048457 <__do_global_ctors_aux+7>:      a1 14 9f 04 08 mov    0x8049f14,%eax
(gdb)
0x0804845c in __do_global_ctors_aux ()
=> 0x0804845c <__do_global_ctors_aux+12>:     83 f8 ff   cmp   $0xffffffff,%eax
(gdb) p/x $eax
$1 = 0x80483b4
```

I was curious after loading the pointer so I told `gdb` `p/x $eax` which means print as hexadecimal the contents of the register `%eax`. It's not `-1`, so we can assume that we'll continue through the loop. Now, since my last command

was the print, I can't hit enter to get an si, I'll have to type it the next time.



```
(gdb) si
0x0804845f in __do_global_ctors_aux ()
=> 0x0804845f <__do_global_ctors_aux+15>:    74 13 je      0x8048474 <__do_global_ctors_aux+36>
(gdb)
0x08048461 in __do_global_ctors_aux ()
=> 0x08048461 <__do_global_ctors_aux+17>:    bb 14 9f 04 08 mov   $0x8049f14,%ebx
(gdb)
0x08048466 in __do_global_ctors_aux ()
=> 0x08048466 <__do_global_ctors_aux+22>:    66 90 xchg   %ax,%ax
(gdb)
0x08048468 in __do_global_ctors_aux ()
=> 0x08048468 <__do_global_ctors_aux+24>:    83 eb 04      sub   $0x4,%ebx
(gdb)
0x0804846b in __do_global_ctors_aux ()
=> 0x0804846b <__do_global_ctors_aux+27>:    ff d0 call  *%eax
(gdb)
a_constructor () at prog2.c:3
3      void __attribute__((constructor)) a_constructor() {
=> 0x080483b4 <a_constructor+0>:    55 push  %ebp
0x080483b5 <a_constructor+1>:    89 e5 mov   %esp,%ebp
0x080483b7 <a_constructor+3>:    83 ec 18 sub   $0x18,%esp
```

Now this is very interesting. We've single stepped into the call. Now we're in our function, **a_constructor**. Since gdb has the source code for it, it shows us the C source for the next line. Since I turned on **disassemble-next-line**, it will also give us the assembler that corresponds to that line. In this case, it's the preamble for the function that corresponds to the declaration of the function, so we get all three lines of the preamble. Isn't that interesting? Now I'm going to switch over to the command n (for next) because our printf is coming up. The first n will skip the preamble, the second the printf, and the third the epilogue. If you've ever wondered why you have to do an extra step at the beginning and end of a function when single stepping with gdb, now you know the answer.

```
(gdb) n
4      printf("%s\n", __FUNCTION__);
=> 0x080483ba <a_constructor+6>:    c7 04 24 a5 84 04 08 movl  $0x80484a5,(%esp)
0x080483c1 <a_constructor+13>:    e8 2a ff ff ff call  0x80482f0 <puts@plt>
```

We moved the address of the string "a_constructor" onto the stack as an argument for **printf**, but it calls **puts** since the compiler was smart enough to see that **puts** was all we needed.

```
(gdb) n
a_constructor
5      }
=> 0x080483c6 <a_constructor+18>:    c9 leave
0x080483c7 <a_constructor+19>:    c3 ret
```

Since we're tracing the program, it is, of course running, so we see **a_constructor** print out above. The closing brace (}) corresponds to the epilogue so that prints out now. Just a note, if you don't know about the instruction **leave** it does exactly the same as

```
movl %ebp, %esp
popl %ebp
```

One more step and we exit the function and return, I'll have to switch back to si.

```
(gdb) n
0x0804846d in __do_global_ctors_aux ()
=> 0x0804846d <__do_global_ctors_aux+29>:    8b 03 mov   (%ebx),%eax
(gdb) si
0x0804846f in __do_global_ctors_aux ()
=> 0x0804846f <__do_global_ctors_aux+31>:    83 f8 ff      cmp   $0xffffffff,%eax
(gdb)
0x08048472 in __do_global_ctors_aux ()
=> 0x08048472 <__do_global_ctors_aux+34>:    75 f4 jne   0x8048468 <__do_global_ctors_aux+24>
```

```
(gdb) p/x $eax
$2 = 0xffffffff
```

Saving to Security

Linux x86 Program Start Up

Got curious and checked again. This time, our function pointer is -1, so we'll exit the loop.

```
(gdb) si
0x08048474 in __do_global_ctors_aux ()
=> 0x08048474 <__do_global_ctors_aux+36>:      83 c4 04      add    $0x4,%esp
(gdb)
0x08048477 in __do_global_ctors_aux ()
=> 0x08048477 <__do_global_ctors_aux+39>:      5b          pop    %ebx
(gdb)
0x08048478 in __do_global_ctors_aux ()
=> 0x08048478 <__do_global_ctors_aux+40>:      5d          pop    %ebp
(gdb)
0x08048479 in __do_global_ctors_aux ()
=> 0x08048479 <__do_global_ctors_aux+41>:      c3          ret
(gdb)
0x080482bc in _init ()
=> 0x080482bc <_init+44>:          58          pop    %eax
```

Notice we're back in `_init` now.

```
(gdb)
0x080482bd in _init ()
=> 0x080482bd <_init+45>:          5b          pop    %ebx
(gdb)
0x080482be in _init ()
=> 0x080482be <_init+46>:          c9          leave
(gdb)
0x080482bf in _init ()
=> 0x080482bf <_init+47>:          c3          ret
(gdb)
0x080483f9 in __libc_csu_init ()
=> 0x080483f9 <__libc_csu_init+25>:      8d bb 1c ff ff ff    lea   -0xe4(%ebx),%edi
(gdb) q
A debugging session is active.

        Inferior 1 [process 17368] will be killed.

Quit anyway? (y or n) y
$
```

Notice we jumped back up into `__libc_csu_init`, and that's when I typed `q` to quite the debugger. That's all the debugging I promised you. Now that we're back in `__libc_csu_init` there's another loop to deal with, and I'm not going to step through it, but I am about to talk about it.

Back up to `__libc_csu_init`

Since we've spent a long tedious time dealing with a loop in assembler and the assembler for this one is even more tedious, I'll leave it to you to figure it out if you want. Just to remind you, here it is in C.

```
void
__libc_csu_init (int argc, char **argv, char **envp)
{
    _init ();

    const size_t size = __init_array_end - __init_array_start;
    for (size_t i = 0; i < size; i++)
        (*__init_array_start [i]) (argc, argv, envp);
}
```

Here's another function call loop

What is this `__init_array`? I thought you'd never ask. You can have code run at this stage as well. Since this is just after returning from running `_init` which ran our constructors, that means anything in this array will run after

constructors are done. You can tell the compiler you want a function to run at this phase. The function will receive the same arguments as main.

```
void init(int argc, char **argv, char **envp) {
    printf("%s\n", __FUNCTION__);
}

__attribute__((section(".init_array"))) typeof(init) *__init = init;
```

Saving to Security

Linux x86 Program Start Up

We won't do it, yet, because there's more things like that. Lets just return from `__libc_csu_init`. Do you remember where that will take us?

We'll be all the way back in `__libc_start_main__`

He calls our main now, and then passes the result to `exit()`.

`exit()` runs some *more* loops of functions

`exit()` runs the functions registered with `at_exit` run in the order they were added. Then he runs another loop of functions, this time, functions in the `fini` array. After that he runs another loop of functions, this time destructors. (In reality, he's in a nested loop dealing with an array of lists of functions, but trust me this is the order they come out in.) Here, I'll show you.

This program, `hooks.c` ties it all together

```
#include <stdio.h>

void preinit(int argc, char **argv, char **envp) {
    printf("%s\n", __FUNCTION__);
}

void init(int argc, char **argv, char **envp) {
    printf("%s\n", __FUNCTION__);
}

void fini() {
    printf("%s\n", __FUNCTION__);
}

__attribute__((section(".init_array"))) typeof(init) *__init = init;
__attribute__((section(".preinit_array"))) typeof(preinit) *__preinit = preinit;
__attribute__((section(".fini_array"))) typeof(fini) *__fini = fini;

void __attribute__((constructor)) constructor() {
    printf("%s\n", __FUNCTION__);
}

void __attribute__((destructor)) destructor() {
    printf("%s\n", __FUNCTION__);
}

void my_atexit() {
    printf("%s\n", __FUNCTION__);
}

void my_atexit2() {
    printf("%s\n", __FUNCTION__);
}

int main() {
    atexit(my_atexit);
    atexit(my_atexit2);
}
```

If you build and run this, (I call it `hooks.c`), the output is

```
$ ./hooks
preinit
constructor
init
```

```

my_atexit2
my_atexit
fini
destructor
$

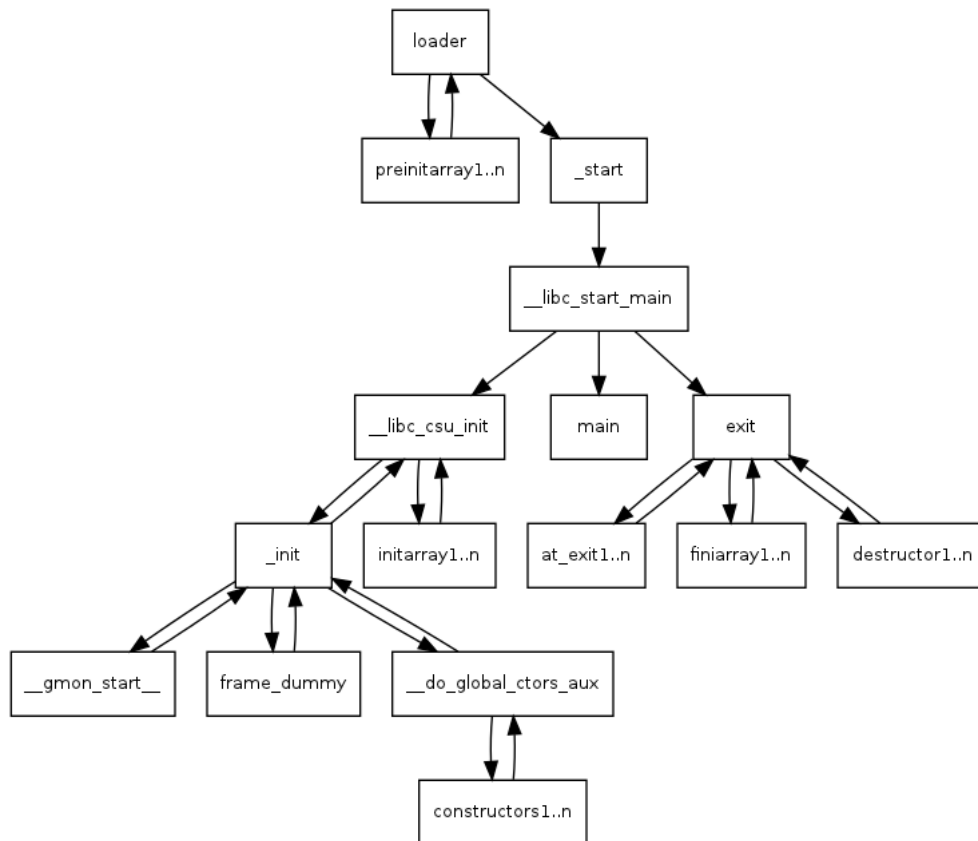
```

Saving to Security

Linux x86 Program Start Up

The End

I'll give you a last look at how far we've come. This time it should all be familiar territory to you.



[\(Back to debugging.\)](#)