2021-06-01 · 4898 words · 27 minutes

#dark-arts · #linkers · #toolchains

Everything You Never Wanted To Know About Linker Script

Low level software usually has lots of `.cc` or `.rs` files. Even lower-level software, like your cryptography library, probably has `.S` containing assembly, my least favorite language for code review.

The lowest level software out there, firmware, kernels, and drivers, have one third file type to feed into the toolchain: an `.ld` file, a "linker script". The linker script, provided to Clang as `-Wl,-T,foo.ld`[1], is like a template for the final executable. It tells the linker how to organize code from the input objects. This permits extremely precise control over the toolchain's output.

Very few people know how to write linker script; it's a bit of an obscure skill. Unfortunately, I'm one of them, so I get called to do it on occasion. Hopefully, this post is a good enough summary of the linker script language that you, too, can build your own binary!

Everything in this post can be found in excruciating detail in GNU `ld`'s documentation; `lld` accepts basically the same syntax. There's no spec, just what your linker happens to accept. I will, however, do my best to provide a more friendly introduction.

No prior knowledge of how toolchains work is necessary! Where possible, I've tried to provide historical context on the names of everything. Toolchains are, unfortunately, bound by half a century of tradition. Better to at least know why they're called that.

> ### Wait, an `.s` file?
>
> On Windows, assembly files use the sensible `.asm` extension. POSIX we use the `.s` extension, or `.S` when we'd like Clang to run the C preprocessor on them (virtually all hand-written assembly is of the second kind).
>
> I don't actually have a historical citation[2] for `.s`, other than that it came from the Unix tradition of obnoxiously terse names. If we are to believe that `.o` stands for "object", and `.a` stands for "archive", then `.s` must stand for "source", up until the B compiler replaced them with `.b` files! See http://man.cat-v.org/unix-1st/1/b.
>
> A final bit of trivia: `.C` files are obviously different from `.c` files... they're C++ files! (Seriously, try it.)

> Note: This post is specifically about POSIX. I know basically nothing about MSVC and `link.exe` other than that they exist. The most I've done is helped people debug trivial `__declspec` issues.

> I will also only be covering things specific to linking an executable; linking other outputs, like shared libraries, is beyond this post.

Seriously, What's a linker?

A linker is but a small part of a *toolchain*, the low-level programmer's toolbox: everything you need to go from source code to execution.

The crown jewel of any toolchain is the compiler. The LLVM toolchain, for example, includes Clang, a C/C++[3] compiler. The compiler takes source code, such as `.cc`, and lowers it down to a `.s` file, an *assembly file* which textually describes machine code for a specific architecture (you can also write them yourself).

Another toolchain program, the assembler, *assembles* each `.s` into a `.o` file, an *object file*[4]. An assembly file is merely a textual representation of an object file; assemblers are not particularly interesting programs.

A third program, the linker, *links* all of your object files into a final *executable* or *binary*, traditionally given the name `a.out`[5].

This three (or two, if you do compile/assemble in one step) phase process is sometimes called the *C compilation model*. All modern software build infrastructure is built around this model[6].

> ### Even More Stages!
>
> Clang, being based on LLVM, actually exposes one stage in between the `.cc` file and the `.s` file. You can ask it to

> skip doing codegen and emit a `.ll` file filled with LLVM IR, an intermediate between human-writable source code and assembly. The magic words to get this file are `clang -S -emit-llvm`. (The Rust equivalent is `rustc --emit=llvm-ir`.)
>
> The LLVM toolchain provides `llc`, the LLVM compiler, which performs the `.ll -> .s` step (optionally assembling it, too). `lli` is an interpreter for the IR. Studying IR is mostly useful for understanding optimization behavior; topic for another day.

The compiler, assembler, and linker are the central components of a toolchain. Other languages, like Rust, usually provide their own toolchain, or just a compiler, reusing the existing C/C++ toolchain. The assembler and linker are language agnostic.

The toolchain also provides various debugging tools, including an interactive debugger, and tools for manipulating object files, such as `nm`, `objdump`, `objcopy`, and `ar`.

These days, most of this stuff is bundled into a single program, the compiler frontend, which knows how to compiler, assemble, and link, in one invocation. You can ask Clang to spit out `.o` files with `clang -c`, and `.s` files with `clang -S`.

> **Trs Nms**
>
> The UNIX crowd at Bell Labs was very excited about short, terse names. This tradition survives in Go's somewhat questionable practice of single-letter variables.

> Most toolchain program names are cute contractions. `cc` is "C compiler"; compilers for almost all other languages follow this convention, like `rustc`, `javac`, `protoc`, and `scalac`; Clang is just `clang`, but is perfectly ok being called as `cc`.
>
> `as` is "assembler"; `ld` is "loader" (you'll learn why sooner). `ar` is "archiver", `nm` is "names". Other names tend to be a bit more sensible.

Final Link

Some fifty years ago at Bell Labs, someone really wanted to write a program with more than one `.s` file. To solve this, a program that could "link" symbol references across object files was written: the first linker.

You can take several `.o` files and use `ar` (an archaic `tar`, basically) to create a library, which always have names like `libfoo.a` (the `lib` is mandatory). A static library is just a collection of objects, which can be provided on an as-needed basis to the linker.

The "final link" incorporates several `.o` files and `.a` files to produce an executable. It does roughly the following:

1. Parse all the objects and static libraries and put their *symbols* into a database. Symbols are named addresses of functions and global variables.
2. Search for all unresolved symbol references in the `.o` files and match it up with a symbol from the database, recursively doing this for any code in a `.a` referenced

during this process. This forms a sort of dependency graph between sections. This step is called *symbol resolution*.

3. Throw out any code that isn't referenced by the input files by tracing the dependency graph from the entry-point symbol (e.g., `_start` on Linux). This step is called *garbage collection*.
4. Execute the linker script to figure out how to stitch the final binary together. This includes discovering the offsets at which everything will go.
5. Resolve *relocations*, "holes" in the binary that require knowing the final runtime address of the section. Relocations are instructions placed in the object file for the linker to execute.
6. Write out the completed binary.

This process is extremely memory-intensive; it is possible for colossal binaries, especially ones with tons of debug information, to "fail to link" because the linker exhausts the system's memory.

We only care about step 4; whole books can be written about the previous steps. Thankfully, Ian Lance Taylor, mad linker scientist and author of `gold`, has written several excellent words on this topic: https://lwn.net/Articles/276782/.

Object Files and Sections

Linkers, fundamentally, consume object files and produce object files; the output is executable, meaning that all relocations have been resolved and an entry-point address (where the OS/bootloader will jump to to start the binary).

It's useful to be able to peek into object files. The `objdump` utility is best for this. `objdump -x my_object.o` will show *all* headers, telling you what exactly is in it.

At a high level, an object file describes how a program should be loaded into memory. The object is divided into sections, which are named blocks of data. Sections may have file-like permissions, such as allocatable, loadable, readonly, and executable. `objdump -h` can be used to show the list of sections. Some selected lines of output from `objdump` on my machine (I'm on a 64-bit machine, but I've trimmed leading zeros to make it all fit):

```
$ objdump -h "$(which clang)"
/usr/bin/clang:     file format elf64-x86-64

Sections:
Idx Name    Size      VMA       LMA       File off  Algn
 11 .init   00000017  00691ab8  00691ab8  00291ab8  2**2
            CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .plt    00006bb0  00691ad0  00691ad0  00291ad0  2**4
            CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .text   0165e861  00698680  00698680  00298680  2**4
            CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .fini   00000009  01cf6ee4  01cf6ee4  018f6ee4  2**2
            CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .rodata 0018ec68  01cf6ef0  01cf6ef0  018f6ef0  2**4
            CONTENTS, ALLOC, LOAD, READONLY, DATA
 24 .data   000024e8  021cd5d0  021cd5d0  01dcc5d0  2**4
            CONTENTS, ALLOC, LOAD, DATA
 26 .bss    00009d21  021cfac0  021cfac0  01dceab8  2**4
            ALLOC
```

Terminal

Allocateable (`ALLOC`) sections must be *allocated* space by the operating system; if the section is loadable (`LOAD`), then the operating system must further fill that space with the contents of the section. This process is called *loading* and is performed by a *loader* program[7]. The loader is sometimes called the "dynamic linker", and is often the same program as the "program linker"; this is why the linker is called `ld`.

Loading can also be done beforehand using the `binary` output format. This is useful for tiny microcontrollers that are too primitive to perform any loading. `objcopy` is useful for this and many other tasks that involve transforming object files.

Some common (POSIX) sections include:

- `.text`, where your code lives[8]. It's usually a loadable, readonly, executable section.
- `.data` contains the initial values of global variables. It's loadable.
- `.rodata` contains constants. It's loadable and readonly.
- `.bss` is an empty allocatable section[9]. C specifies that uninitialized globals default to zero; this is a convenient way for avoiding storing a huge block of zeros in the executable!
- Debug sections that are not loaded or allocated; these are usually removed for release builds.

After the linker decides which sections from the `.o` and `.a` inputs to keep (based on which symbols it decided it needed), it looks to the linker script how to arrange them in the output.

Let's write our first linker script!

```
SECTIONS {
  /* Define an output section ".text". */
  .text : {
    /* Pull in all symbols in input sections named .text */
    *(.text)
    /* Do the same for sections starting with .text.,
       such as .text.foo */
    *(.text.*)
```

```
  }

  /* Do the same for ".bss", ".rodata", and ".data". */
  .bss : { *(.bss); *(.bss.*) }
  .data : { *(.data); *(.data.*) }
  .rodata : { *(.rodata); *(.rodata.*) }
}
```
Linker Script

This tells the linker to create a `.text` section in the output, which contains all sections named `.text` from all inputs, plus all sections with names like `.text.foo`. The content of the section is laid out in order: the contents of all `.text` sections will come before any `.text.*` sections; I don't think the linker makes any promises about the ordering between different objects[10].

As I mentioned before, parsers for linker script are fussy[11]: the space in `.text :` is significant.

Note that the two `.text` sections are different, and can have different names! The linker generally doesn't care what a section is named; just its attributes. We could name it `code` if we wanted to; even the leading period is mere convention. Some object file formats don't support arbitrary sections; all the sane ones (ELF, COFF, Mach-O) don't care, but they don't all spell it the same way; in Mach-O, you call it `__text`.

Before continuing, I recommend looking at the appendix so that you have a clear path towards being able to run and test your linker scripts!

> ### Input Section Syntax
>
> None of this syntax is used in practice but it's useful to contextualize the syntax for pulling in a section. The full form of the syntax is

> ```
> archive:object(section1 section2 ...)
> ```
> Plaintext
>
> Naturally, all of this is optional, so you can write `foo.o` or `libbar.a:(.text)` or `:baz.o(.text .data)`, where the last one means "not part of a library". There's even an `EXCLUDE_FILE` syntax for filtering by source object, and a `INPUT_SECTION_FLAGS` syntax for filtering by the presence of format-specific flags.
>
> Do not use any of this. Just write `*(.text)` and don't think about it too hard. The `*` is just a glob for all objects.

Each section has an *alignment*, which is just the maximum of the alignments of all input sections pulled into it. This is important for ensuring that code and globals are aligned the way the architecture expects them to be. The alignment of a section can be set explicitly with

```
SECTIONS {
  .super_aligned : ALIGN(16) {
    /* ... */
  }
}
```
Linker Script

You can also instruct the linker to toss out sections using the special `/DISCARD/` output section, which overrides any decisions made at garbage-collection time. I've only ever used this to discard debug information that GCC was really excited about keeping around.

On the other hand, you can use `KEEP(*(.text.*))` to ensure no `.text` sections are discarded by garbage-collection. Unfortunately, this doesn't let you pull in sections from static libraries that weren't referenced in the input objects.

LMA and VMA

Every section has three addresses associated with it. The simplest is the file offset: how far from the start of the file to find the section.

The *virtual memory address*, or VMA, is where the program expects to find the section at runtime. This is the address that is used by pointers and the program counter.

The *load memory address*, or LMA, is where the loader (be it a runtime loader or `objcpy`) must place the code. This is almost always the same as the VMA. Later on, in Using Symbols and LMAs, I'll explain a place where this is actually useful.

When declaring a new section, the VMA and LMA are both set to the value[12] of the *location counter*, which has the *extremely* descriptive name `.` [13]. This counter is automatically incremented as data is copied from the input

We can explicitly specify the VMA of a section by putting an expression before the colon, and the LMA by putting an expression in the `AT(lma)` specifier *after* the colon:

```
SECTIONS {
  .text 0x10008000: AT(0x40008000) {
    /* ... */
  }
}
```

Linker Script

This will modify the location counter; you could also write it as

```
SECTIONS {
  . = 0x10008000;
  .text : AT(0x40008000) {
    /* ... */
```

```
    }
  }
```

Linker Script

Within `SECTIONS`, the location counter can be set at any point, even while in the middle of declaring a section (though the linker will probably complain if you do something rude like move it backwards).

The location counter is incremented automatically as sections are added, so it's rarely necessary to fuss with it directly.

Memory Regions and Section Allocation

By default, the linker will simply allocate sections starting at address `0`. The `MEMORY` statement can be used to define *memory regions* for more finely controlling how VMAs and LMAs are allocated without writing them down explicitly.

A classic example of a `MEMORY` block separates the address space into ROM and RAM:

```
MEMORY {
  rom (rx)   : ORIGIN = 0x8000,     LENGTH = 16K
  ram (rw!x) : ORIGIN = 0x10000000, LENGTH = 256M
}
```

Linker Script

A region is a block of memory with a name and some attributes. The name is irrelevant beyond the scope of the linker script. The attributes in parens are used to specify what sections could conceivably go in that region. A section is compatible if it has any of the attributes before the `!`, and none which come after the `!`. (This filter mini-language isn't very expressive.)

The attributes are the ones we mentioned earlier: `rwxal` are readonly, read/write, executable, allocated, and loadable[14].

When allocating a section a VMA, the linker will try to pick the best memory region that matches the filter using a heuristic. I don't really trust the heuristic, but you can instead write `> region` to put something into a specific region. Thus,

```
SECTION {
  .data {
    /* ... */
  } > ram AT> rom
}
```

Linker Script

`AT>` is the "obvious" of `AT()` and `>`, and sets which region to allocate the LMA from.

The origin and length of a region can be obtained with the `ORIGIN(region)` and `LENGTH(region)` functions.

Other Stuff to Put In Sections

Output sections can hold more than just input sections. Arbitrary data can be placed into sections using the `BYTE`, `SHORT`, `LONG` and `QUAD` for placing literal 8, 16, 32, and 64-bit unsigned integers into the section:

```
SECTIONS {
  .screams_internally : { LONG(0xaaaaaaaa) }
}
```

Linker Script

Numeric literals in linker script may, conveniently, be given the suffixes `K` or `M` to specify a kilobyte or megabyte quantity. E.g., `4K` is sugar for `4096`.

Fill

You can fill the unused portions of a section by using the `FILL` command, which sets the "fill pattern" from that point onward.

For example, we can create four kilobytes of `0xaa` using `FILL` and the location counter:

```
SECTIONS {
  .scream_page : {
    FILL(0xaa)
    . += 4K;
  }
}
```
Linker Script

The "fill pattern" is used to fill any unspecified space, such as alignment padding or jumping around with the location counter. We can use multiple FILLs to vary the fill pattern, such as if we wanted half the page to be `0x0a` and half `0xa0`:

```
SECTIONS {
  .scream_page : {
    FILL(0x0a)
    . += 2K;
    FILL(0xa0)
    . += 2K;
  }
}
```
Linker Script

When using one fill pattern for the whole section, you can just write `= fill;` at the end of the section. For example,

```
SECTIONS {
  .scream_page : {
    . += 4K;
  } = 0xaa;
}
```
Linker Script

Linker Symbols

Although the linker needs to resolve all symbols using the input `.o` and `.a` files, you can also declare symbols directly in linker script; this is the absolute latest that symbols can be provided. For example:

```
SECTIONS {
  my_cool_symbol = 5;
}
```
Linker Script

This will define a new symbol with value `5`. If we then wrote `extern char my_cool_symbol;`, we can access the value placed by the linker. However, note that the value of a symbol is an *address*! If you did

```
extern char my_cool_symbol;

uintptr_t get() {
  return my_cool_symbol;
}
```
C

the processor would be very confused about why you just dereferenced a pointer with address `5`. The *correct* way to extract a linker symbol's value is to write

```
extern char my_cool_symbol;

uintptr_t get() {
  return (uintptr_t)&my_cool_symbol;
}
```
C

It seems a bit silly to take the address of the global and use that as some kind of magic value, but that's just how it works. The exact same mechanism works in Rust, too:

```
fn get() -> usize {
  extern "C" {
    #[link_name = "my_cool_symbol"]
    static SYM: u8;
  }

  addr_of!(SYM) as usize
}
```
Rust

The most common use of this mechanism is percolating information not known until link time. For example, a common idiom is

```
SECTIONS {
  .text : {
    __text_start = .;
    /* stuff */
    __text_end = .;
  }
}
```

Linker Script

This allows initialization code to find the section's address and length; in this case, the pointer values are actually meaningful!

> **Wunderbars**
>
> It's common practice to lead linker symbols with two underscores, because C declares a surprisingly large class of symbols reserved for the implementation, so normal user code won't call them. These include names like `__text_start`, which start with two underscores, and names starting with an underscore and an uppercase letter, like `_Atomic`.
>
> However, libc and STL headers will totally use the double underscore symbols to make them resistant to tampering by users (which they are entitled to), so beware!

Symbol assignments can even go inside of a section, to capture the location counter's value between input sections:

```
SECTIONS {
  .text : {
    *(.text)
    text_middle = .;
    *(.text.*)
```

```
    }
  }
```


Symbol names are not limited to C identifiers, and may contain dashes, periods, dollar signs, and other symbols. They may even be quoted, like `"this symbol has spaces"`, which C will never be able to access as an `extern`.

There is a mini-language of expressions that symbols can be assigned to. This includes:

- Numeric literals like `42`, `0xaa`, and `4K`.
- The location counter, `.`.
- Other symbols.
- The usual set of C operators, such as arithmetic and bit operations. Xor is curiously missing.
- A handful of builtin functions, described below.

There are some fairly complicated rules around how symbols may be given relative addresses to the start of a section, which are only relevant when dealing with position-independent code: https://sourceware.org/binutils/docs/ld/Expression-Section.html

Functions belong to one of two board categories: getters for properties of sections, memory regions, and other linker structures; and arithmetic. Useful functions include:

- `ADDR`, `LOADADDR`, `SIZEOF`, and `ALIGNOF`, which produce the VMA, LMA, size, and alignment of a previously defined section.
- `ORIGIN` and `LENGTH`, which produce the start address and length of a memory region.
- `MAX`, `MIN` are obvious; `LOG2CEIL` computes the base-2 log, rounded up.

- `ALIGN(expr, align)` rounds `expr` to the next multiple of `align`. `ALIGN(align)` is roughly equivalent to `ALIGN(., align)` with some subtleties around PIC. `. = ALIGN(align);` will align the location counter to `align`.

Some other builtins can be found at https://sourceware.org/binutils/docs/ld/Builtin-Functions.html.

A symbol definition can be wrapped in the `PROVIDEO()` function to make it "weak", analogous to the "weak symbol" feature found in Clang. This means that the linker will not use the definition if any input object defines it.

Using Symbols and LMAs

As mentioned before, it is extremely rare for the LMA and VMA to be different. The most common situation where this occurs is when you're running on a system, like a microcontroller, where memory is partitioned into two pieces: ROM and RAM. The ROM has the executable burned into it, and RAM starts out full of random garbage.

Most of the contents of the linked executable are read-only, so their VMA can be in ROM. However, the `.data` and `.bss` sections need to lie in RAM, because they're writable. For `.bss` this is easy, because it doesn't have loadable content. For `.data`, though, we need to separate the VMA and LMA: the VMA must go in RAM, and the LMA in ROM.

This distinction is important for the code that initializes the RAM: while for `.bss` all it has to do is zero it, for `.data`, it has to copy from ROM to RAM! The LMA lets us distinguish the copy source and the copy destination.

This has the important property that it tells the loader (usually `objcopy` in this case) to use the ROM addresses for actually loading the section to, but to link the code as if it were at a RAM address (which is needed for things like PC-relative loads to work correctly).

Here's how we'd do it in linker script:

```
MEMORY {
  rom : /* ... */
  ram : /* ... */
}

SECTIONS {
  /* .text and .rodata just go straight into the ROM. We don't need
     to mutate them ever. */
  .text : { *(.text) } > rom
  .rodata : { *(.rodata) } > rom

  /* .bss doesn't have any "loadable" content, so it goes straight
     into RAM. We could include `AT> rom`, but because the sections
     have no content, it doesn't matter. */
  .bss : { *(.bss) } > ram

  /* As described above, we need to get a RAM VMA but a ROM LMA;
     the > and AT> operators achieve this. */
  .data : { *(.data) } > ram AT> rom
}

/* The initialization code will need some symbols to know how to
   zero the .bss and copy the initial .data values. We can use the
   functions from the previous section for this! */

bss_start = ADDR(.bss);
bss_end = bss_start + SIZEOF(.bss);

data_start = ADDR(.data);
data_end = data_start + SIZEOF(.data);

rom_data_start = LOADADDR(.data);
```

Linker Script

Although we would normally write the initialization code in assembly (since it's undefined behavior to execute C before initializing the `.bss` and `.data` sections), I've written it in C for illustrative purposes:

```c
#include <string.h>

extern char bss_start[];
extern char bss_end[];
extern char data_start[];
extern char data_end[];
extern char rom_data_start[];

void init_sections(void) {
  // Zero the .bss.
  memset(bss_start, 0, bss_end - bss_start);

  // Copy the .data values from ROM to RAM.
  memcpy(data_start, rom_data_start, data_end - data_start);
}
```

C

Misc Linker Script Features

Linker script includes a bunch of other commands that don't fit into a specific category:

- `ENTRY()` sets the program entry-point, either as a symbol or a raw address. The `-e` flag can be used to override it. The `ld` docs assert that there are fallbacks if an entry-point can't be found, but in my experience you can sometimes get errors here. `ENTRY(_start)` would use the `_start` symbol, for example[15].
- `INCLUDE "path/to/file.ld"` is `#include` but for linker script.
- `INPUT(foo.o)` will add `foo.o` as a linker input, as if it was passed at the commandline. `GROUP` is similar, but with the semantics of `--start-group`.
- `OUTPUT()` overrides the usual `a.out` default output name.
- `ASSERT()` provides static assertions.
- `EXTERN(sym)` causes the linker to behave as if an undefined reference to `sym` existed in an input object.

(Other commands are documented, but I've never needed them in practice.)

Real Linker Scripts

It may be useful to look at some real-life linker scripts.

If you wanna see what Clang, Rust, and the like all ultimately use, run `ld --verbose`. This will print the default linker script for your machine; this is a really intense script that uses basically every feature available in linker script (and, since it's GNU, is very poorly formatted).

The Linux kernel also has linker scripts, which are differently intense, because they use the C preprocessor. For example, the one for amd64: https://github.com/torvalds/linux/blob/master/arch/x86/kernel/vmlinux.lds.S.

Tock OS, a secure operating system written in Rust, has some pretty solid linker scripts, with lots of comments: https://github.com/tock/tock/blob/master/boards/kernel_layout.ld. I recommend taking a look to see what a "real" but not too wild linker script looks like. There's a fair bit of toolchain-specific stuff in there, too, that should give you an idea of what to expect.

Happy linking!

Appendix: A Linker Playground

tl;dr: If you don't wanna try out any examples, skip this section.

I want you to be able to try out the examples above, but there's no Godbolt for linker scripts (yet!). Unlike normal code, you can't just run linker script through a compiler, you're gonna need some objects to link, too! Let's set up a very small C project for testing your linker scripts.

> Note: I'm assuming you're on Linux, with x86_64, and using Clang. If you're on a Mac (even M1), you can probably make `ld64` do the right thing, but this is outside of what I'm an expert on.
>
> If you're on Windows, use WSL. I have no idea how MSCV does linker scripts at all.

First, we want a very simple static library:

```c
int lib_call(const char* str) {
  // Discard `str`, we just want to take any argument.
  (void)str;

  // This will go in `.bss`.
  static int count;
  return count++;
}
```

godbolt extern.c

Compile `extern.c` into a static library like so:

```shell
clang -c extern.c
ar rc libextern.a extern.o
```

Shell

We can check out that we got something reasonable by using `nm`. The `nm` program shows you all the symbols a library or object defines.

```
$ nm libextern.a
extern.o:
```

```
0000000000000000 T lib_call
0000000000000000 b lib_call.count
```
Terminal

This shows us the address, section type, and name of each symbol; `man nm` tells us that `T` means `.text` and `b` means `.bss`. Capital letters mean that the symbol is *exported*, so the linker can use it to resolve a symbol reference or a relocation. In C/C++, symbols declared `static` or in an unnamed namespace are "hidden", and can't be referenced outside of the object. This is sometimes called internal vs external linkage.

Next, we need a C program that uses the library:

```c
extern int lib_call(const char* str);

// We're gonna use a custom entrypoint. This code will never run anyways,
// just care about the linker output.
void run(void) {
  // This will go in `.data`, because it's initialized to non-zero.
  static int data = 5;

  // The string-constant will go into `.rodata`.
  data = lib_call("Hello from .rodata!");
}
```
godbolt run.c

Compile it with `clang -c run.c`. We can inspect the symbol table with `nm` as before:

```
$ nm run.o
                 U lib_call
0000000000000000 T run
0000000000000000 d run.data
```
Terminal

As you might guess, `d` is just `.data`. However, `U` is interesting: it's an undefined symbol, meaning the linker will need to perform a symbol resolution! In fact, if we ask Clang to link this for us (it just shells out to a linker like `ld`):

```
$ clang run.o
/usr/bin/ld: /somewhere/crt1.o: in function `_start':
(.text+0x20): undefined reference to `main'
/usr/bin/ld: run.o: in function `run':
run.c:(.text+0xf): undefined reference to `lib_call'
```
Terminal

The linker also complains that there's no `main()` function, and that some object we didn't provide called `crt1.o` wants it. This is the startup code for the C runtime; we can skip linking it with `-nostartfiles`. This will result in the linker picking an entry point for us.

We can resolve the missing symbol by linking against our library. `-lfoo` says to search for the library `libfoo.a`; `-L.` says to include the current directory for searching for libraries.

```
clang run.o -L. -lextern -nostartfiles
```
Shell

This gives us our binary, `a.out`, which we can now `objdump`:

```
$ objdump -d -Mintel a.out

a.out:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <run>:
  401000:  55                      push   rbp
  401001:  48 89 e5                mov    rbp,rsp
  401004:  48 bf 00 20 40 00 00    movabs rdi,0x402000
  40100b:  00 00 00
  40100e:  e8 0d 00 00 00          call   401020 <lib_call>
  401013:  89 04 25 00 40 40 00    mov    DWORD PTR ds:0x404000,eax
  40101a:  5d                      pop    rbp
  40101b:  c3                      ret
  40101c:  0f 1f 40 00             nop    DWORD PTR [rax+0x0]

0000000000401020 <lib_call>:
  401020:  55                      push   rbp
  401021:  48 89 e5                mov    rbp,rsp
  401024:  48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
  401028:  8b 04 25 04 40 40 00    mov    eax,DWORD PTR ds:0x404004
```

```
  40102f:   89 c1                   mov    ecx,eax
  401031:   83 c1 01                add    ecx,0x1
  401034:   89 0c 25 04 40 40 00    mov    DWORD PTR ds:0x404004,ecx
  40103b:   5d                      pop    rbp
  40103c:   c3                      ret
```

Terminal

Let's write up the simplest possible linker script for all this:

```
ENTRY(run)
SECTIONS {
  .text : { *(.text); *(.text.*) }
  .bss : { *(.bss); *(.bss.*) }
  .data : { *(.data); *(.data.*) }
  .rodata : { *(.rodata); *(.rodata.*) }
}
```

link.ld

Let's link! We'll also want to make sure that the system libc doesn't get in the way, using `-nostdlib`[16].

```
clang run.o -L. -lextern -nostartfiles -nostdlib -Wl,-T,link.ld
```

Shell

At this point, you can use `objdump` to inspect `a.out` at your leisure! You'll notice there are a few other sections, like `.eh_frame`. Clang adds these by default, but you can throw them out using `/DISCARD/`.

It's worth it to run the examples in the post through the linker using this "playground". You can actually control the sections Clang puts symbols into using the `__attribute__((section("blah")))` compiler extension. The Rust equivalent is `#[link_section = "blah"]`. ■

---

(1) Blame GCC for this. `-Wl` feeds arguments through to the linker, and `-T` is `ld`'s linker script input flag. Thankfully, rustc is far more sensible here: `-Clink-args=-Wl,-T,foo.ld` (when GCC/Clang is your linker frontend). ↺

(2) Correction, 2022-09-11. I have really been bothered by not knowing if this is actually true, and have periodically asked around about it. I asked Russ Cox, who was actually *at* Bell Labs back in the day, and he asked Ken Thompson, who confirms: it's genuinely `.s` for source, because it was the only source they had back then.

I am glad I got this from the horse's mouth. :) ↺

---

(3) And many other things, like Objective-C. ↺

---

(4) Completely and utterly unrelated to the objects of object-oriented programming. Best I can tell, the etymology is lost to time. ↺

---

(5) `a.out` is *also* an object file format, like ELF, but toolchains live and die by tradition, so that's the name given to the linker's output by default. ↺

---

(6) Rust does not compile each `.rs` file into an object, and its "crates" are much larger than the average C++ translation unit. However, the Rust compiler will nonetheless produce many object files for a single crate, precisely for the benefit of this compilation model. ↺

---

(7) Operating systems are loaded by a bootloader. Bootloaders are themselves loaded by other bootloaders, such as the BIOS. At the bottom of the turtles is the mask ROM, which is a tiny bootloader permanently burned into the device. ↺

(8) No idea on the etymology. This isn't ASCII text! ↺

(9) Back in the 50s, this stood for "block started by symbol". ↺

(10) Yes, yes, you can write `SORT_BY_NAME(*)(.text)` but that's not really something you ever wind up needing.

See https://sourceware.org/binutils/docs/ld/Input-Section-Wildcards.html for more information on this. ↺

(11) You only get `/* */` comment syntax because that's the lowest common denominator. ↺

(12) Well, `.` actually gets increased to the alignment of the section first. If you insist on an unaligned section, the syntax is, obviously,

```
SECTIONS {
  .unaligned .: {
    /* ... */
  }
}
```

Linker Script

(That was sarcasm. It must be stressed that this is not a friendly language.) ↺

(13) This symbol is also available in assembly files. `jmp .` is an overly-cute idiom for an infinity busy loop. It is even more terse in ARM and RISC-V, where it's written `b .` and `j .`, respectively.

Personally, I prefer the obtuse clarity of `loop_forever: j loop_forever`. ↺

---

(14) These are the same characters used to declare a section in assembly. If I wanted to place my code in a section named `.crt0` but wanted it to be placed into a readonly, executable memory block, use the the assembler directive `.section .crt0, rxal` ↺

---

(15) Note that the entry point is almost never a function called `main()`. In the default configuration of most toolchains, an object called `crt0.o` is provided as part of the libc, which provides a `_start()` function that itself calls `main()`. CRT stands for "C runtime"; thus, `crt0.o` initializes the C runtime.

This file contains the moral equivalent of the following C code, which varies according to target:

```c
extern int main(int argc, char** argv);
noreturn void _start() {
  init_libc();    // Initializes global libc state.
  run_ctors();    // Runs all library constructors.
  int ret = main(get_argc(), get_argv());
  run_dtors();    // Runs all library destructors.
  cleanup_libc(); // Deinitializes the libc.

  exit(ret); // Asks the OS to gracefully destroy the process.
}
```

This behavior can be disabled with `-nostartfiles` in Clang. The OSDev wiki has some on this topic: https://wiki.osdev.org/Creating_a_C_Library#Program_Initialization. ↺

(16) If you include libc, you will get bizarre errors involving something called "`gcc_s`". `libgcc` (and `libgcc_s`) is GCC's *compiler runtime* library. Where `libc` exposes high-level operations on the C runtime and utilities for manipulating common objects, `libgcc` provides even lower-level support, including:

- Polyfills for arithmetic operations not available on the target. For example, dividing two 64-bit integers on most 32-bit targets will emit a reference to the a symbol like `__udivmoddi4` (they all have utterly incomprehensible names like this one).
- Soft-float implementations, i.e., IEEE floats implemented in software for targets without an FPU.
- Bits of unwinding (e.g. exceptions and panics) support (the rest is in `libunwind`).
- Miscellaneous runtime support code, such as the code that calls C++ static initializers.

Clang's version, `libcompiler-rt`, is ABI-compatible with `libgcc` and provides various support for profiling, sanitizers, and many, many other things the compiler needs available for compiling code. ↺

Related Posts

- 2024-04-17 / The Rust Calling Convention We Deserve
- 2023-11-27 / Designing a SIMD Algorithm from Scratch

- 2023-09-29 / ***What is a Matrix? A Miserable Pile of Coefficients!***