# [How does C++ linking work in practice?](https://stackoverflow.com/questions/12122446/how-does-c-linking-work-in-practice)

https://stackoverflow.com/questions/12122446/how-does-c-linking-work-in-practice

This answer focuses on address relocation, which is one of the crucial functions of linking.

A minimal example will be used to clarify the concept.

## 0) Introduction

Summary: relocation edits the .text section of object files to translate:

* object file address
* into the final address of the executable

This must be done by the linker because the compiler only sees one input file at a time, but we must know about all object files at once to decide how to:

* resolve undefined symbols like declared undefined functions
* not clash multiple .text and .data sections of multiple object files

Prerequisites: minimal understanding of:

* x86-64 or IA-32 assembly
* global structure of an ELF file. I have made [a tutorial for that](http://www.cirosantilli.com/elf-hello-world/)

Linking has nothing to do with C or C++ specifically: compilers just generate the object files. The linker then takes them as input without ever knowing what language compiled them. It might as well be Fortran.

So to reduce the crust, let's study a NASM x86-64 ELF Linux hello world:

section .data

hello\_world db "Hello world!", 10

section .text

global \_start

\_start:

; sys\_write

mov rax, 1

mov rdi, 1

mov rsi, hello\_world

mov rdx, 13

syscall

; sys\_exit

mov rax, 60

mov rdi, 0

syscall

compiled and assembled with:

nasm -felf64 hello\_world.asm # creates hello\_world.o

ld -o hello\_world.out hello\_world.o # static ELF executable with no libraries

with NASM 2.10.09.

## 1) .text of .o

First we decompile the .text section of the object file:

objdump -d hello\_world.o

which gives:

0000000000000000 <\_start>:

0: b8 01 00 00 00 mov $0x1,%eax

5: bf 01 00 00 00 mov $0x1,%edi

a: 48 be 00 00 00 00 00 movabs $0x0,%rsi

11: 00 00 00

14: ba 0d 00 00 00 mov $0xd,%edx

19: 0f 05 syscall

1b: b8 3c 00 00 00 mov $0x3c,%eax

20: bf 00 00 00 00 mov $0x0,%edi

25: 0f 05 syscall

the crucial lines are:

a: 48 be 00 00 00 00 00 movabs $0x0,%rsi

11: 00 00 00

which should move the address of the hello world string into the rsi register, which is passed to the write system call.

But wait! How can the compiler possibly know where "Hello world!" will end up in memory when the program is loaded?

Well, it can't, specially after we link a bunch of .o files together with multiple .data sections.

Only the linker can do that since only he will have all those object files.

So the compiler just:

* puts a placeholder value 0x0 on the compiled output
* gives some extra information to the linker of how to modify the compiled code with the good addresses

This "extra information" is contained in the .rela.text section of the object file

## 2) .rela.text

.rela.text stands for "relocation of the .text section".

The word relocation is used because the linker will have to relocate the address from the object into the executable.

We can disassemble the .rela.text section with:

readelf -r hello\_world.o

which contains;

Relocation section '.rela.text' at offset 0x340 contains 1 entries:

Offset Info Type Sym. Value Sym. Name + Addend

00000000000c 000200000001 R\_X86\_64\_64 0000000000000000 .data + 0

The format of this section is fixed documented at: <http://www.sco.com/developers/gabi/2003-12-17/ch4.reloc.html>

Each entry tells the linker about one address which needs to be relocated, here we have only one for the string.

Simplifying a bit, for this particular line we have the following information:

* Offset = C: what is the first byte of the .text that this entry changes.

If we look back at the decompiled text, it is exactly inside the critical movabs $0x0,%rsi, and those that know x86-64 instruction encoding will notice that this encodes the 64-bit address part of the instruction.

* Name = .data: the address points to the .data section
* Type = R\_X86\_64\_64, which specifies what exactly what calculation has to be done to translate the address.

This field is actually processor dependent, and thus documented on the [AMD64 System V ABI extension](http://www.x86-64.org/documentation/abi.pdf) section 4.4 "Relocation".

That document says that R\_X86\_64\_64 does:

* + Field = word64: 8 bytes, thus the 00 00 00 00 00 00 00 00 at address 0xC
  + Calculation = S + A
    - S is value at the address being relocated, thus 00 00 00 00 00 00 00 00
    - A is the addend which is 0 here. This is a field of the relocation entry.

So S + A == 0 and we will get relocated to the very first address of the .data section.

## 3) .text of .out

Now lets look at the text area of the executable ld generated for us:

objdump -d hello\_world.out

gives:

00000000004000b0 <\_start>:

4000b0: b8 01 00 00 00 mov $0x1,%eax

4000b5: bf 01 00 00 00 mov $0x1,%edi

4000ba: 48 be d8 00 60 00 00 movabs $0x6000d8,%rsi

4000c1: 00 00 00

4000c4: ba 0d 00 00 00 mov $0xd,%edx

4000c9: 0f 05 syscall

4000cb: b8 3c 00 00 00 mov $0x3c,%eax

4000d0: bf 00 00 00 00 mov $0x0,%edi

4000d5: 0f 05 syscall

So the only thing that changed from the object file are the critical lines:

4000ba: 48 be d8 00 60 00 00 movabs $0x6000d8,%rsi

4000c1: 00 00 00

which now point to the address 0x6000d8 (d8 00 60 00 00 00 00 00 in little-endian) instead of 0x0.

Is this the right location for the hello\_world string?

To decide we have to check the program headers, which tell Linux where to load each section.

We disassemble them with:

readelf -l hello\_world.out

which gives:

Program Headers:

Type Offset VirtAddr PhysAddr

FileSiz MemSiz Flags Align

LOAD 0x0000000000000000 0x0000000000400000 0x0000000000400000

0x00000000000000d7 0x00000000000000d7 R E 200000

LOAD 0x00000000000000d8 0x00000000006000d8 0x00000000006000d8

0x000000000000000d 0x000000000000000d RW 200000

Section to Segment mapping:

Segment Sections...

00 .text

01 .data

This tells us that the .data section, which is the second one, starts at VirtAddr = 0x06000d8.

And the only thing on the data section is our hello world string.