Linker in a C compiler

Harshil Goyal

January 13, 2025

1 Introduction

To convert source code written in a high-level language into a low-level, machine-executable format, the code undergoes the following steps [1]:

$$SourceCode \xrightarrow{\text{.c File}} Preprocessor \xrightarrow{\text{.i File}} Compiler \xrightarrow{\text{.s File}} Assembler \xrightarrow{\text{.o file}} Linker \rightarrow ExecutableFile$$

Explanation of each produced file in the process:

- **.c File**: The source code written in a high-level language (in this case, we are assuming C). This is the initial point for the compilation process.
- **.i File**: The preprocessed file, produced by expanding macros and resolving `#include` directives.
- **.s File**: The assembly code generated by the compiler.
- **.o File**: The object code, a binary file produced by the assembler.
- **Executable File**: The final binary file produced by the linker, combining all essential object files and resolving symbols.

2 Linker

The process of collecting and combining multiple pieces of code or data into a single file that can be copied into memory and then executed is known as Linking. Linking can be performed at compile time, when the source code is translated into machine code, at load time, when the program is loaded into memory and executed by the loader, and even at run time. On early computer systems, linking was performed manually. On modern systems, linking is performed automatically by programs called linkers. [2]
Commonly used modern linkers are:

- **ld** in UNIX-like systems.
- **Microsoft Incremental Linker** in Windows

Whenever we compile source code written in C, the above mentioned steps are executed to get the final binary file. After the compiler generates the assembly code, the assembler compiles the produced (**.s file**) and generates object files. The object files generated in this step generally contain the program's compiled functions, data, and references to external symbols that are not yet resolved. These symbols may or may not refer to the functions or variables defined in the

other object files or possibly in the other libraries. These references are resolved by the linker making sure that all symbols are defined and their addresses are correct.
There are two types of linking [3]:

- **Static Linking**: In this type of linking, the linker copies the dependencies into the final executable.
- **Dynamic Linking**: In dynamic linking, the required libraries are linked during the run-time of the executable.

2.1 Example

Consider a program with two files: **main.c** and **fun.c**. Assume main.c calls a function, named **"foo"** defined in **fun.c**. After both files are compiled separately into **main.o** and **fun.o**, the linker or the link editor combines these files, resolving the reference to **"foo"** from **main.o** by linking it to the definition in **fun.o**. Additionally the linker incorporates the libraries specified by user in the source code.

References

[1] Linker - GeeksForGeeks. `https://www.geeksforgeeks.org/linker/`. [Accessed 12-01-2025].

[2] CMU Class Notes - Linking. `http://csapp.cs.cmu.edu/2e/ch7-preview.pdf`. [Accessed 12-01-2025].

[3] Static vs. Dynamic Linking — Baeldung on Computer Science — baeldung.com. `https://www.baeldung.com/cs/static-dynamic-linking-differences`. [Accessed 13-01-2025].