Menofia
Branch

Track

Professional Development and BI-infused CRM

Name

Ahmed Mamdouh Abd EL Ghany

Topic

Static & Dynamic linking

The Role of the Linker and the Linking Process

When a C++ program is compiled, each source file is first translated into an **object file** that contains machine code. However, these object files are not yet complete programs — they may depend on functions or variables defined elsewhere.

This is where the **linker** comes into play, as the linker's primary job is to **combine multiple object files** and **resolve symbol references** so that all function calls and variable accesses point to their correct locations in memory. It ensures that every symbol used in the program is matched with its definition, and it generates a final executable file that the operating system can run.

Static Linking

In static linking, the linker takes all the object files and the required libraries, then **copies the necessary code directly into the final executable file**. This means the executable becomes completely self-contained and does not depend on any external library files at runtime.

The advantage of this approach is that once compiled, the program can run on any compatible system without needing to install additional library files. This makes deployment easier and guarantees that the program will behave the same way everywhere, since it always uses the exact same version of every library.

However, static linking comes with a cost. Because the code from the libraries is physically included in the executable, the file size becomes significantly larger. In addition, if a library needs to be updated then every program that uses it must be recompiled to include the new version.

Although it consumes more storage and requires recompilation for updates, it provides maximum independence from external dependencies and ensures a stable, predictable runtime environment.

Dynamic Linking

Dynamic linking takes a different approach. Instead of embedding the library code inside the executable, the linker only records **references** to external shared libraries that contain the required functions.

When the program is executed, the **operating system's loader** takes responsibility for finding these shared libraries, loading them into memory, and connecting the program's references to the actual functions in the library.

This approach keeps the executable file much smaller, since it does not contain the full code of the libraries. Moreover, multiple programs can share the same library loaded in memory, saving a large amount of RAM and disk space.

Another major advantage is that libraries can be updated independently, so if a shared library is improved or patched, all programs using it automatically benefit from the changes without recompilation. On the other hand, dynamic linking introduces a dependency like if the required library is missing or incompatible, the program will fail to start. This makes deployment slightly more complex, as the correct versions of shared libraries must be available on the target system.

Comparison Between Static and Dynamic Linking

FEATURE	STATIC LINKING	DYNAMIC LINKING
LIBRARY INTEGRATION	Library code is copied directly into the executable.	Executable contains only references to external libraries.
EXECUTABLE SIZE	Larger, because it includes all required code.	Smaller, as code stays in shared libraries.
RUNTIME DEPENDENCY	No dependency on external libraries.	Requires shared libraries to be present at runtime.
MEMORY USAGE	Higher, each program has its own copy of the code.	Lower, since multiple programs share one copy in memory.
UPDATING LIBRARIES	Requires recompilation to apply updates.	Automatically benefits from updated shared libraries.
PORTABILITY	Fully self-contained and easy to distribute.	Requires correct library versions on the target system.