Separate Compilation

The most straightforward mechanism for separate compilation can be found in module-based languages such as Modula-2, Modula-3, and Ada, which allow a module to be divided into a **declaration** part (header) and an **implementation** part (body). The header contains all and only the information needed by users of the module (or needed by the compiler in order to compile such a user); the body contains the rest.

As a matter of software engineering practice, a design team will typically define module interfaces early in the lifetime of a project, and codify these interfaces in the form of module headers. Individual team members or subteams will then work to implement the module bodies. While doing so, they can compile their code successfully using the headers for the other modules.

In a simple implementation, only the body module needs to be compiled into runnable code: the compiler can read the header of module M when compiling the body of M, and also when compiling the body of any module that uses M. In a more sophisticated implementation, the compiler can avoid the overhead of a repeatedly scanning, parsing, and analyzing M's header by translating it into a symbol table, which is accessed directly when compiling their module headers.

As a practical matter, many languages allow the header of a module to be subdivided into a "public" part, which specifies the interface to the rest of the program, and a "private" part, which is not visible outside the module, but is needed by the compiler, for example todetermine the storage requirements of opaque types. Ideally, one would include in the header of a module only that information that the programmer needs to know to use the abstraction(s) that the module provides. Restricted exports, and the public and private portions of headers, are compromises introduced to allow the compiler to generate code in the face of separate compilation.

At some point prior to execution, modules that have been separately compiled must be "glued together" to form a single program. This job is the task of the **linker**. At the very least, the linker must resolve cross-module references (loads, stores, jumps) and *relocate* any instructions whose encoding depends on the location of certain modules in the final program. Typically it also checks to make sure that users and implementors of a given interface agree on the version of the header file used to define that interface. In some environments, the linker may perform additional tasks as well, including certain kinds of interprocedural (whole-program) code improvement.