The Compiler's Job

In computer science, the *compiler is responsible for transformation of source code to executeable machine code image.*
(`source.cpp`) → (`GNU gcc`) → (`executeable.o`)

Let f be a function that represents the compiler.
In this case $f(S^{(n)}) = M^{(m)}$ transformation applies.

$$\Rightarrow f : S^{(n)} \rightarrow M^{(m)}$$
$$(n < m \ [n, m \in Z])$$

where S is n units of source code (eg.: in `C++`) taken,
and M is m units of machine code (eg.: in `assembly`) produced - which are executeable set of CPU instructions.

A compiler can be considered as a processing pipeline, which can be conceptually decomposed to the following stages:

1. **Front-End** Takes plain source code as its input (eg. `C++` code) and transforms it to an itermediate representation (`IR`) as its output.
   Where the main phases are:

   - **Preprocessing** - Removal of any unneccesary text (eg. comments), macro substitution and conditional compilation.
   - **Lexical Analysis** (`tokenization`)- Breaking down the input source code into a sequence of lexical tokens.
     A token has 2 components: name and value - where value is being assigned to the name.
     The value has to be computed according to its category and type. Token categories are keywords, identifiers, separators.
   - **Syntax Analysis** (`parsing`) - Builds the parse tree from a linear sequence of tokens. This tree structure must obey to the formal grammar which defines the language's syntax.
   - **Semantic Analysis** - Adds semantic information to the parse tree and builds the symbol table. This step performs type checking (eg. identifying type errors) and object binding which associates variables and references with their definitions.

2. **Middle-End** - Takes the IR as its input and transforms it to Optimized Intermediate Representation (`OIR`) as its output. This stage can be skipped due to its space and time requirements (eg. `debug` mode or `release` mode).
   Performs optimizations on the IR to improve performance the produceable machine code. Some optimizations at this stage are general - aka. they are independent of the target CPU architecture.

   The main phases are:

   - **Analysis** - Performs analytics on data-flows, dependencies, pointers. Builds the control-flow graph and the call graph.
   - **Optimization** - IR transformation to an equivalent but more efficient form. IR reduction by eliminating dead code, inlining, constant propagation and loop transformation.

     The above two phases results in the OIR.

3. **Back-End** - Takes OIR as its input and transforms it to machine code (eg. `assembly`, `native machine language`) as its output that is executeable by the target CPU.
   The back end has 2 phases:

   - **Target Machine-Dependent Optimization** - Further optimization options that the target CPU allows to perform.
   - **Machine Code Generation** - Transformation from OIR to native machine language.

The Linker's Job

In computer science, *the linker is responsible for transforming a given set of executeable machine code images into a single executteable machine code image.* Possible outputs are:

- **Dynamically Linked Library** - `.a` or `.dll` (its parts can be loaded and executed by another executeable at runtime).
- **Statically Linked Library** - `.so` or `.lib` (it can be combined with another executeable by the linker to produce an executeable which incorporated this).
- **Executeable** - `.o` or `.exe` (it can be launched by the hosting environment [Operating System] and can be ran as a program).