Scope

What is scope ?
The textual region of the program in which a binding is active is its scope.

How Do we determine scope in C ?

In most modern languages, the scope of a binding is determined statically, that is, at compile time. In
C, for example, we introduce a new scope upon entry to a subroutine. We create bindings for local objects and deactivate bindings for global objects that are "hidden" by local objects of the same name. On subroutine exit, we destroy bindings for local variables and reactivate bindings for any global
objects that were hidden.

These manipulations of bindings may at first glance appear to be run-time operations, but they do not require the execution of any code: the portions of the program in which a binding is active are completely determined at compile time. We can look at a C program and know which names refer to which objects at which points in the program based on purely textual rules. For this reason, C is said to be statically scoped (some authors say lexically scoped).

In addition to talking about the "scope of a binding," we sometimes use the word scope as a noun all by itself, without a specific binding in mind.

Scope is a program region of maximal size in which no bindings change (or at least none are destroyed). Typically, a scope is the body of a module, class, subroutine, or structured control flow statement, sometimes called a block. In C family languages it would be delimited with {...} braces.

Which are Dynamically scoped Languages ?
Other languages, including APL, Snobol, and early dialects of Lisp, are dynamically scoped: their bindings depend on the flow of execution at run time.

What is elaboration ?

Algol 68 and Ada use the term elaboration to refer to the process by which declarations become active when control first enters a scope. Elaboration entails the creation of bindings. In many languages, it also entails the allocation of stack space for local objects, and possibly the assignment of initial values. In Ada it can entail a host of other things, including the execution of error-checking or heap-space-allocating code, the propagation of exceptions, and the creation of concurrently executing tasks.

What is referencing environment ?

At any given point in a program's execution, the set of active bindings is called the current referencing environment. The set is principally determined by static or dynamic scope rules.

A referencing environment generally corresponds to a sequence of scopes that can be examined (in order) to find the current binding for a given name.

In some cases, referencing environments also depend on binding rules. Specifically, when a reference to a subroutine S is stored in a variable, passed as a parameter to another subroutine, or returned as a function value, one needs to determine when the referencing environment for S is chosen—that is, when the binding between the reference to S and the referencing environment of S is made. The two principal options are deep binding, in which the choice is made when the reference is first created, and shallow binding, in which the choice is made when the reference is finally used.

Static Scoping ---

In a language with static (lexical) scoping, the bindings between names and objects can be determined at compile time by examining the text of the program, without consideration of the flow of control at run time.

Typically, the “current” binding for a given name is found in the matching declaration whose block most closely surrounds a given point in the program. There also many variants to this basic theme.

The simplest static scope rule is probably that of early versions of Basic Language, in which there was only a single, global scope. In fact, there were only a few hundred possible names, each of which consisted of a letter optionally followed by a digit. There were no explicit declarations; variables were declared implicitly by virtue of being used.

Scope rules are somewhat more complex in (pre-Fortran 90) Fortran, as it distinguishes between global and local variables. The scope of a local variable is limited to the subroutine in which it appears; it is not visible elsewhere. Variable declarations are optional. If a variable is not declared, it is assumed to be local to the current subroutine and to be of type integer if its name begins with the letters I–N, or real otherwise. (Different conventions for implicit declarations can be specified by the programmer. In Fortran 90, the programmer can also turn off implicit declarations, so that use of an undeclared variable becomes a compile-time error.)

Global variables in Fortran may be partitioned into common blocks, which are then “imported” by subroutines. Common blocks are designed for separate compilation: they allow a subroutine to import only the sets of variables it needs. Unfortunately, Fortran requires each subroutine to declare the names and types of the variables in each of the common blocks it uses, and there is no standard mechanism to ensure that the declarations in different subroutines are the same.

Semantically, the lifetime of a local Fortran variable (both the object itself and the name-to-object binding) encompasses a single execution of the variable’s subroutine. Programmers can override this rule by using an explicit save statement.