3 Names, Scopes, and Bindings

3.4 Implementing Scope

For both static and dynamic scoping, a language implementation must keep track of the name-to-object bindings in effect at each point in the program. The principal difference is *time*: with static scope the compiler uses a *symbol table* to track bindings at compile time; with dynamic scoping the interpreter or run-time system uses an *association list* or *central reference table* to track bindings at run time.

3.4.1 Symbol Tables

In a language with static scoping, the compiler uses an insert operation to place a name-to-object binding into the symbol table for each newly encountered declaration. When it encounters the use of a name that should already have been declared, the compiler uses a lookup operation to search for an existing binding. It is tempting to try to accommodate the visibility rules of static scoping by performing a remove operation to delete a name from the symbol table at the end of its scope. Unfortunately, several factors make this straightforward approach impractical:

- The ability of inner declarations to hide outer ones in most languages with nested scopes means that the symbol table has to be able to contain an arbitrary number of mappings for a given name. The lookup operation must return the innermost mapping, and outer mappings must become visible again at end of scope.
- Records (structures) and classes have some of the properties of scopes, but do not share their nicely nested structure. When it sees a record declaration, the semantic analyzer must remember the names of the record's fields (recursively, if records are nested). At the end of the declaration, the field names must become invisible. Later, however, whenever a variable of the record type appears in the program text (as in my_rec.field_name), the record fields must suddenly become visible again for the part of the reference after the dot. In object-oriented languages, member (field and method) names must become visible throughout

the methods of the class, even if (as in C++) the code for the methods can appear outside the class declaration.

- As noted in Section 3.3.3, names are sometimes used before they are declared. Algol and C, for example, permit *forward references* to labels. Pascal permits forward references in pointer declarations. Most object-oriented languages permit forward references to class members. Modula-3 permits forward references of all kinds.
- As noted in Section 3.3.3, C, C++, and Ada distinguish between the declaration of an object and its definition. Pascal has a similar mechanism for mutually recursive subroutines. When it sees a declaration, the compiler must remember any nonvisible details so that it can check the eventual definition for consistency. This operation is similar to remembering the field names of records and classes.
- While it may be desirable to forget names at the end of their scope, and even to reclaim the space they occupy in the symbol table, information about them may need to be saved for use by a *symbolic debugger* (Section 16.3.2). A debugger allows the user to manipulate a running program: starting it, stopping it, and reading and writing its data. In order to parse high-level commands, the debugger must have access to the compiler's symbol table, which the compiler typically saves in a hidden portion of the final machine-language program.

EXAMPLE 3.45

The LeBlanc-Cook symbol table

To accommodate these concerns, most compilers never delete anything from the symbol table. Instead, they manage visibility using enter_scope and leave_scope operations. Implementations vary from compiler to compiler; the approach described here is due to LeBlanc and Cook [CL83].

Each scope, as it is encountered, is assigned a serial number. The outermost scope (the one that contains the predefined identifiers) is given number 0. The scope containing programmer-declared global names is given number 1. Additional scopes are given successive numbers as they are encountered. All serial numbers are distinct; they do not represent the level of lexical nesting, except in as much as nested subroutines naturally end up with numbers higher than those of surrounding scopes. If language rules specify that a declaration should be visible only in the remainder of the current code block (not the preceding portion), we can even allocate a serial number for each such declaration, to capture the scope that is the remainder of the block.

All names, regardless of scope, are entered into a single large hash table, keyed by name. Each entry in the table then contains the symbol name, its category (variable, constant, type, procedure, field name, parameter, etc.), scope number, type (a pointer to another symbol table entry), and additional, category-specific fields.

In addition to the hash table, the symbol table has a *scope stack* that indicates, in order, the scopes that compose the current referencing environment. As the semantic analyzer scans the program, it pushes and pops this stack whenever it enters or leaves a scope, respectively. Entries in the scope stack contain the scope number, an indication of whether the scope is closed, and in some cases further information.

```
procedure lookup(name)
    pervasive := best := null
    apply hash function to name to find appropriate chain
    foreach entry e on chain
        if e.name = name        -- not something else with same hash value
            if e.scope = 0
                pervasive := e
            else
                foreach scope s on scope stack, top first
                    if s.scope = e.scope
                        best := e        -- closer instance
                        exit inner loop
                    elsif best ≠ null and then s.scope = best.scope
                        exit inner loop        -- won't find better
                    if s.closed
                        exit inner loop        -- can't see farther
    if best ≠ null
        while best is an import or export entry
            best := best.real_entry
        return best
    elsif pervasive ≠ null
        return pervasive
    else
        return null        -- name not found
```

Figure 3.17 LeBlanc-Cook symbol table lookup operation.

To look up a name in the table, we scan down the appropriate hash chain looking for entries that match the name we are trying to find. For each matching entry, we scan down the scope stack to see if the scope of that entry is visible. We look no deeper in the stack than the top-most closed scope. Imports and exports are made visible outside their normal scope by creating additional entries in the table; these extra entries contain pointers to the real entries. We don't have to examine the scope stack at all for entries with scope number 0: they are pervasive. Pseudocode for the lookup algorithm appears in Figure C-3.17.

EXAMPLE 3.46
Symbol table for a sample program

The lower right portion of Figure C-3.18 contains the skeleton of a C++ program. The remainder of the figure shows the configuration of the symbol table for the referencing environment of the grey arrow shown in function `F2`. At this point in the code, the scope stack contains four entries, representing, respectively, the (anonymous) type of structure `S`, function `F2`, namespace (module) `M2`, and the global scope. The scope for the anonymous type indicates the specific variable (i.e., `S`) to which names (fields) in this scope belong. The outermost, pervasive scope is not explicitly represented.

All of the entries for a given name appear on the same hash chain, since the table is keyed on name. In this example, we assume that hash collisions have placed `M2` on the same chain as the `J`s, and the anonymous structure type (which will

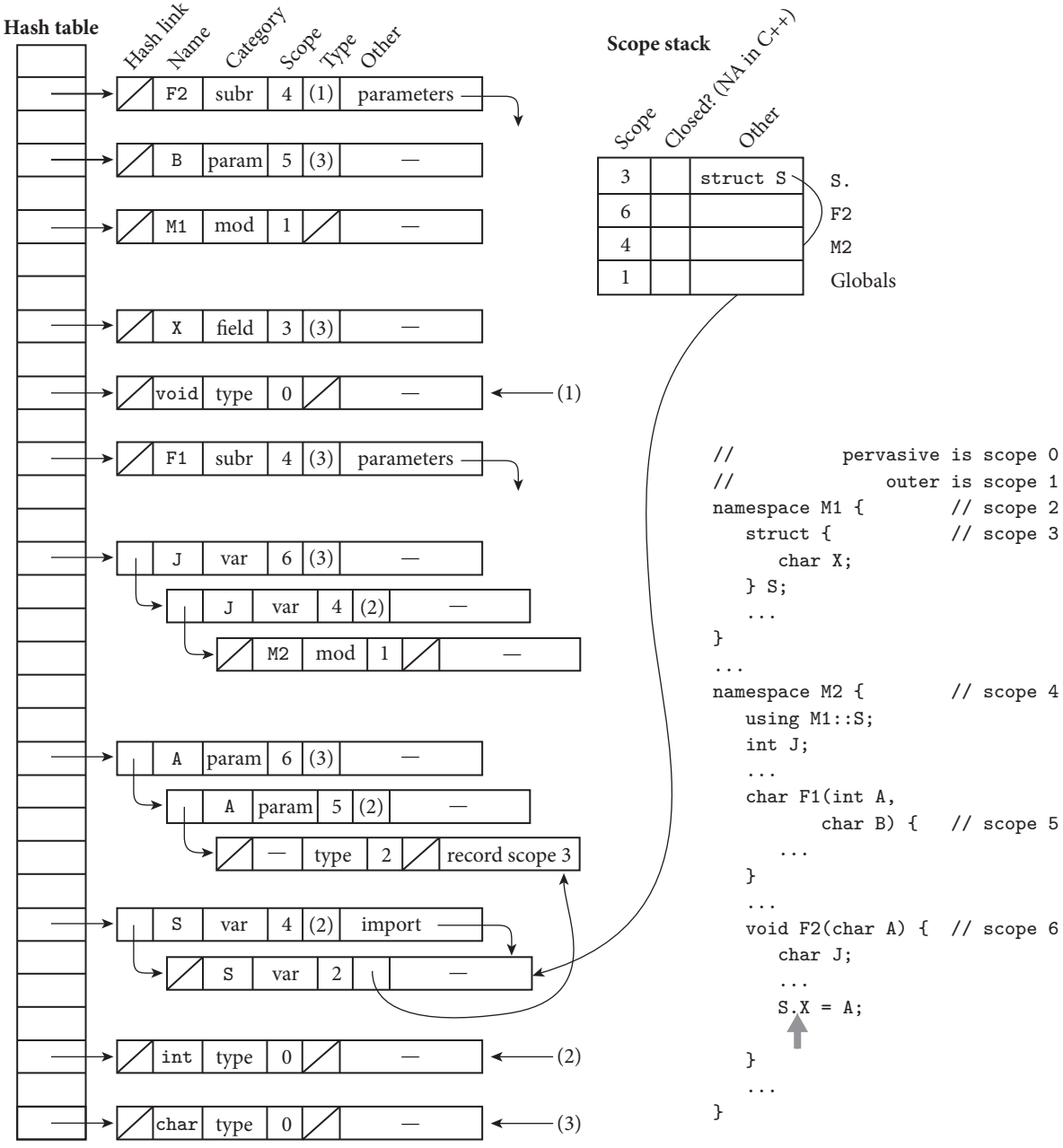

Figure 3.18 LeBlanc-Cook symbol table for an example program in a language like C++. The scope stack represents the referencing environment at the grey arrow shown in function `F2`. For the sake of clarity, the many pointers from type fields to the symbol table entries for `void`, `int`, and `char` are shown as parenthesized (1)s, (2)s, and (3)s, rather than as arrows.

have some arbitrary internal name) on the same chain as the `A`s. Variable `S` has an extra entry, to make it directly visible inside `M2`, as requested by the `using` statement. When we are inside `F2`, a lookup operation on `J` will find `F2`'s `J`; the `J` in `M2` will be hidden by virtue of `F2` being above `M2` on the scope stack. The entry for the anonymous `struct` type indicates the scope number to be pushed onto the scope stack while resolving references to fields within objects of that type. The entry for each function contains the head pointer of a list that links together the subroutine's parameters, for use in analyzing calls (additional links of these chains are not shown). During code generation, many symbol table entries would contain additional fields, for such information as size and run-time address.

The second column of the scope stack is intended to indicate closed scopes. While C++ doesn't have any of these, we can imagine how they would work. For example, if `M2` were closed, then names in the global scope, which appears below `M2` in the scope stack, would not be visible at the indicated point in the code. Anything imported into `M2` *would* be visible, because it would have an extra entry (like that of `S`) within `M2`'s own scope.[1] If our language had exports (again, C++ does not), we would create extra entries in the *outer* scope, analogous to the ones we create in inner scopes for imports.

Classes, which we did not use in Figure C-3.18, would be handled much like a combination of namespaces and structures. Field and method names of the class would be visible to the class's methods, much as objects in a namespace are visible to all the namespace's code. Moreover, the entry for the class—like that of a structure type—would indicate the scope to be pushed onto the scope stack when the compiler has parsed a dot (`.`) or arrow (`->`) token and expects the next token to name a field or method of the class.

It is tempting to suggest extending a LeBlanc-Cook style symbol table to handle the visibility specifications common in object-oriented languages (e.g., the `public`, `private`, `protected` keywords of C++, to which we will return in Section 10.2.2), but this is probably a mistake. For one thing, such an extension would likely be quite messy. It is easy to make all the names in a scope visible, by pushing that scope onto the scope stack. It is also relatively easy to make a small number of names visible, by creating extra entries in other scopes, as we did for imports and exports. An intermediate option does not immediately present itself. More significantly, when the programmer attempts to use a field or method inappropriately, we probably want to generate an error message along the lines of "method `m` is private in class `foo`" instead of just "method name `foo` not found." This observation suggests employing a traditional lookup mechanism for class members, followed by a separate check for visibility in the current context. We consider this possibility in Exercise C-3.27.

---

1 Recall that the `using` statement in C++ isn't an import in the usual sense: it just gives a simpler (unqualified) name to an already-visible object.

3.4.2 Association Lists and Central Reference Tables

Pictorial representations of the two principal implementations of dynamic scoping appear in Figures C-3.19 and C-3.20. Association lists (*A-lists*) are simple and elegant, but can be very inefficient. Central reference tables resemble a simplified LeBlanc-Cook symbol table, without the separate scope stack; they require more work at scope entry and exit than do association lists, but they make lookup operations fast.

EXAMPLE 3.47
A-list lookup in Lisp

A-lists are widely used for dictionary abstractions in Lisp; they are supported by a rich set of built-in functions in most Lisp dialects. It is therefore natural for a simple Lisp interpreter to use an A-list to keep track of name-value bindings, and even to make this list explicitly visible to the running program. Since bindings are created when entering a scope, and destroyed when leaving or returning from a scope, the A-list functions as a stack. When execution enters a scope at run time, the interpreter pushes bindings for names declared in that scope onto the top of the A-list. When execution finally leaves a scope, these bindings are removed. To look up the meaning of a name in an expression, the interpreter searches from the top of the list until it finds an appropriate binding (or reaches the end of the list, in which case an error has occurred). Each entry in the list contains whatever information is needed to perform semantic checks (e.g., type checking, which we will consider in Section 7.2) and to find variables and other objects that occupy memory locations. In the left half of Figure C-3.19, the first (top) entry on the A-list represents the most recently encountered declaration: the I in procedure P. The second entry represents the J in procedure Q. Below these are the global symbols, Q, P, J, and I, and (not shown explicitly) any predefined names provided by the Lisp interpreter. ■

EXAMPLE 3.48
Central reference table

The problem with using an association list to represent a program's referencing environment is that it can take a long time to find a particular entry in the list, particularly if it represents an object declared in a scope encountered early in the program's execution, and now buried deep in the list. A central reference table is designed for faster access. It has one slot for every distinct name in the program. The table slot in turn contains a list (stack) of declarations encountered at run time, with the most recent occurrence at the beginning of the list. Looking up a name is now easy: the current meaning is found at the beginning of the list in the appropriate slot in the table. In the upper part of Figure C-3.20, the first entry on the I list is the I in procedure P; the second is the global I. If the program is compiled and the set of names is known at compile time, then each name can have a statically assigned slot in the table, which the compiled code can refer to directly. If the program is not compiled, or the set of names is not statically known, then a hash function will need to be used at run time to find the appropriate slot. ■

When control enters a new scope at run time, entries must be pushed onto the beginning of every list in the central reference table whose name is (re)declared in that scope. When control leaves a scope for the final time, these entries must be popped. The work involved is somewhat more expensive than pushing and popping an A-list, but not dramatically more so, and lookup operations are now

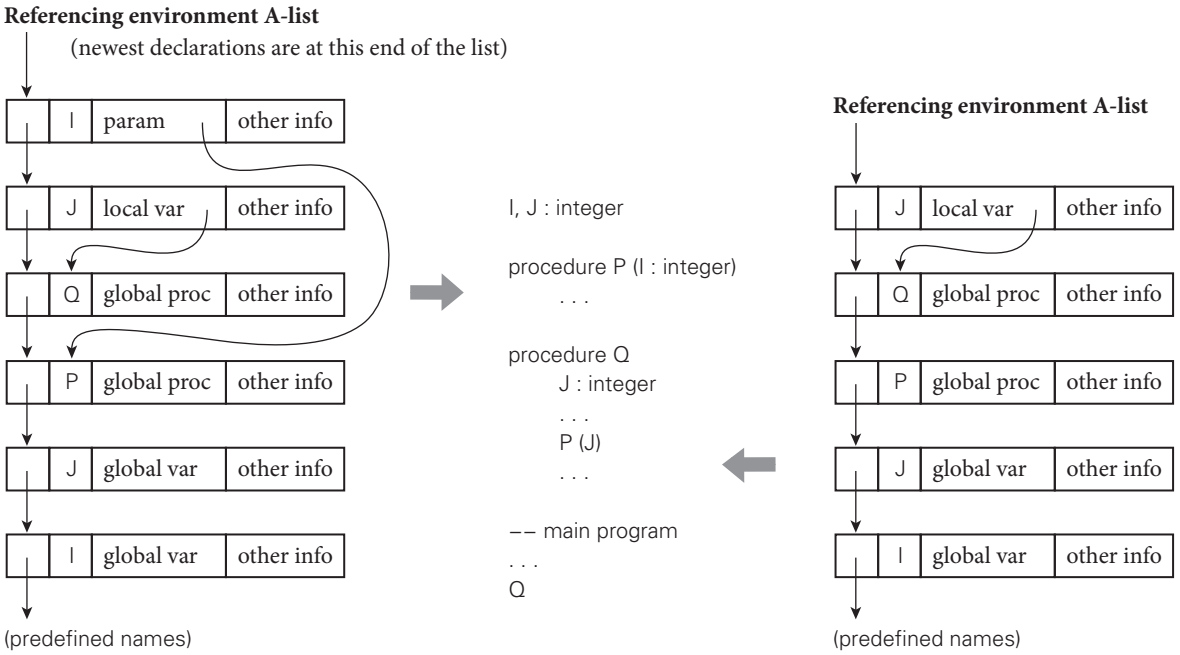

Figure 3.19 Dynamic scoping with an association list. The left side of the figure shows the referencing environment at the point in the code indicated by the adjacent grey arrow: after the main program calls Q and it in turn calls P. When searching for I, one will find the parameter at the beginning of the A-list. The right side of the figure shows the environment at the other grey arrow: after P returns to Q. When searching for I, one will find the global definition.

much faster. In contrast to the symbol table of a compiler for a language with static scoping, central reference table entries for a given scope do not need to be saved when the scope completes execution; the space can be reclaimed.

Within the Lisp community, implementation of dynamic scoping via an association list is sometimes called *deep binding*, because the lookup operation may need to look arbitrarily deep in the list. Implementation via a central reference table is sometimes called *shallow binding*, because it finds the current association at the head of a given reference chain. Unfortunately, the terms "deep and shallow binding" are also more widely used for a completely different purpose, discussed in Section 3.6. To avoid potential confusion, some authors use "deep and shallow access" [Seb19] or "deep and shallow search" [Fin96] for the implementations of dynamic scoping.

Closures with Dynamic Scoping

(This subsection is best read after Section 3.6.1.)

EXAMPLE 3.49
A-list closures

If an association list is used to represent the referencing environment of a program with dynamic scoping, the referencing environment in a closure can be represented by a top-of-stack (beginning of A-list) pointer (Figure C-3.21). When

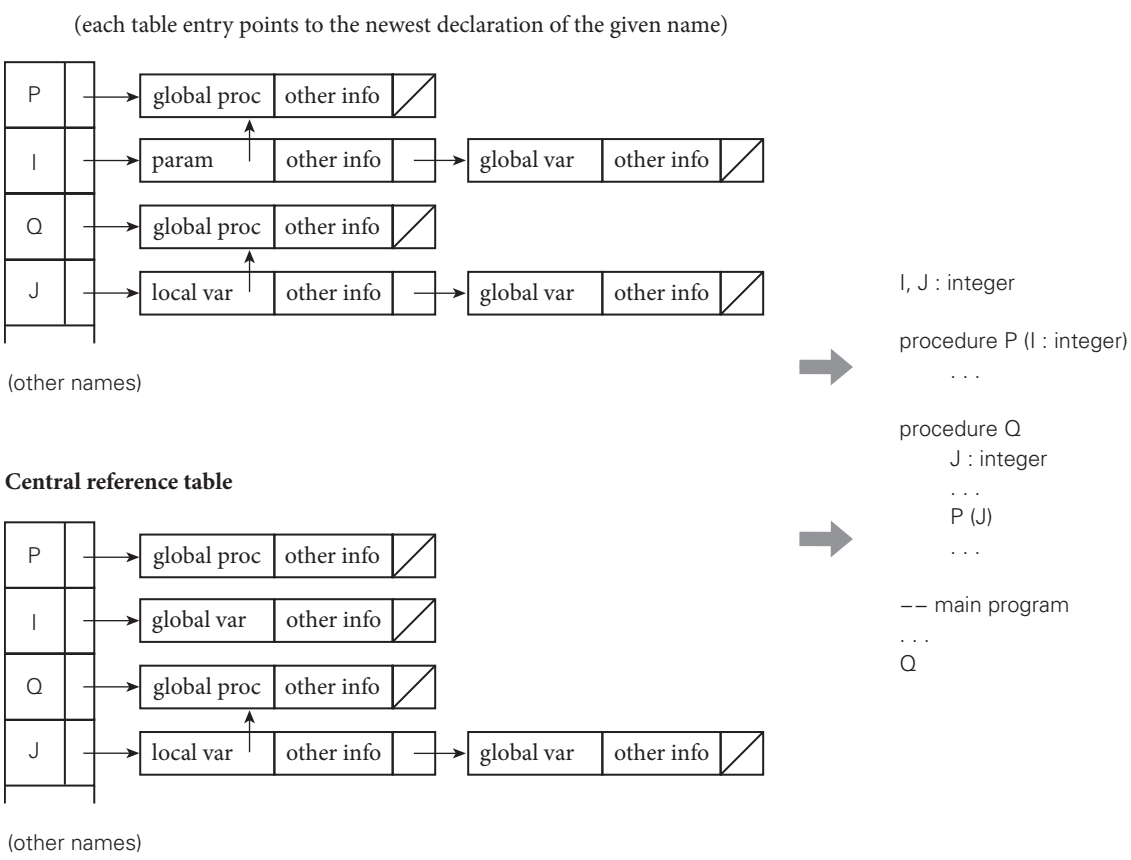

Figure 3.20 **Dynamic scoping with a central reference table.** The upper half of the figure shows the referencing environment at the point in the code indicated by the upper grey arrow: after the main program calls Q and it in turn calls P. When searching for I, one will find the parameter at the beginning of the chain in the I slot of the table. The lower half of the figure shows the environment at the lower grey arrow: after P returns to Q. When searching for I, one will find the global definition.

a subroutine is called through a closure, the main pointer to the referencing environment A-list is temporarily replaced by the pointer from the closure, making any bindings created since the closure was created (P's I and J in the figure) temporarily invisible. New bindings created *within* the subroutine (or in any subroutine it calls) are pushed using the temporary pointer. Because the A-list is represented by pointers (rather than an array), the effect is to have two lists—one representing the caller's referencing environment and the other the temporary referencing environment resulting from use of the closure—that share their older entries. When Q returns to P in our example, the original head of the A-list will be restored, making P's I and J visible again.

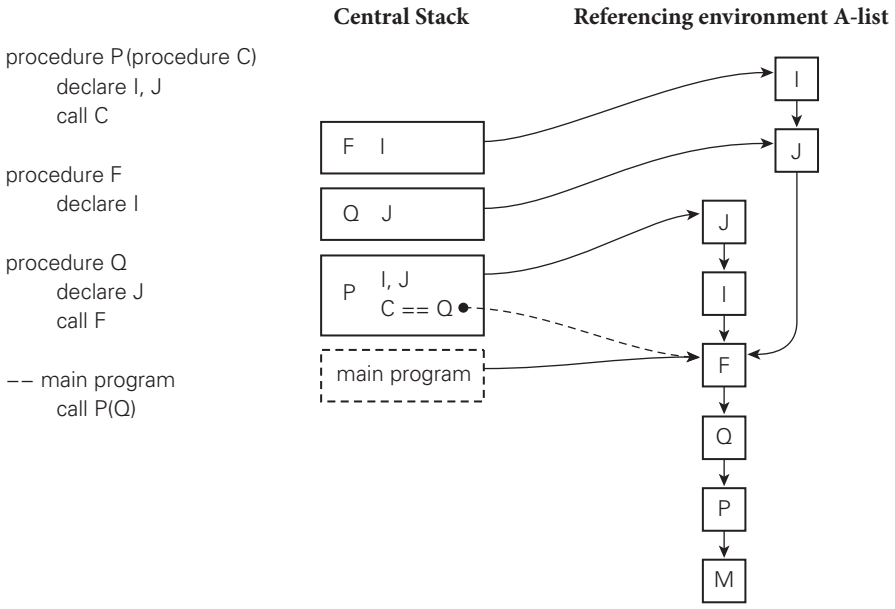

Figure 3.21 **Capturing the A-list in a closure.** Each frame in the stack has a pointer to the current beginning of the A-list, which the run-time system uses to look up names. When the main program passes Q to P with deep binding, it bundles its A-list pointer in Q's closure (dashed arrow). When P calls C (which is Q), it restores the bundled pointer. When Q elaborates its declaration of J (and F elaborates its declaration of I), the A-list is temporarily bifurcated.

With a central reference table implementation of dynamic scoping, the creation of a closure is more complicated. In the general case, it may be necessary to copy the entire main array of the central table and the first entry on each of its lists. Space and time overhead may be reduced if the compiler or interpreter is able to determine that only some of the program's names will be used by the subroutine in the closure (or by things that the subroutine may call). In this case, the environment can be saved by copying the first entries of the lists for only the names that will be used. When the subroutine is called through the closure, these entries can then be pushed onto the beginnings of the appropriate lists in the central reference table. Additional code must be executed to remove them again after the subroutine returns.

CHECK YOUR UNDERSTANDING

43. List the basic operations provided by a symbol table.
44. Outline the implementation of a LeBlanc-Cook style symbol table.
45. Why don't compilers generally remove names from the symbol table at the ends of their scopes?

46. Describe the *association list* (*A-list*) and *central reference table* data structures used to implement dynamic scoping. Summarize the tradeoffs between them.

47. Explain how to implement deep binding by capturing the referencing environment A-list in a closure. Why are closures harder to build with a central reference table?

3 Names, Scopes, and Bindings

3.8 Separate Compilation

Probably the most straightforward mechanisms 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 (or *header*) and an implementation part (or *body*). As we noted in Section 3.3.4, 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. Using preliminary copies of the bodies, they may also be able to undertake a certain amount of testing.

In a simple implementation, only the body of a 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 repeatedly scanning, parsing, and analyzing *M*'s header by translating it into a symbol table, which is then accessed directly when compiling the bodies of *M* and its users. Most Ada implementations compile their module headers. Implementations of Modula-2 and 3 vary: some work one way, some the other.

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 to determine 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 compro-

mises 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. We will return to the subject of linking in Chapters 15 and 16.

3.8.1 Separate Compilation in C

The initial version of C was designed at Bell Laboratories around 1970. It has evolved considerably over the years, but not, for the most part, in the area of separate compilation. Here the language remains comparatively primitive. In particular, there is in general no way for the compiler or the linker to detect inconsistencies among declarations or uses of a name in different files. The C89 standards committee introduced a new explanation of separate compilation based on the notion of *linkage*, but this served mainly to clarify semantics, not to change them. The current rules can be summarized as follows (certain details and special cases are omitted):

- If the declaration of a global object (variable or function) contains the word `static`, then the object has *internal linkage*, and is identified with (linked to) any other internally linked declaration of the same name in the same file.
- If the declaration of a function does not contain the keyword `static`, then it has *external linkage*, and is identified with any other (nonstatic) declaration of the same function in any file of the program. (A function declaration may consist of just the header.)
- If the declaration of a variable contains the keyword `extern`, then the variable has the same linkage as any visible, internally or externally linked declaration of the same name appearing earlier in the file. If there is no earlier declaration, then the variable has external linkage, and is identified with any other declaration of the same external variable in any file of the program. In other words, files in the same program that contain matching external variable declarations actually share the same variable. A global variable also has external linkage if its declaration says neither `static` nor `extern`.
- If an object is declared with both internal and external linkage, the behavior of the program is undefined.
- An object (variable or function) that is externally linked must have a *definition* in exactly one file of a program. A variable is defined when it is given an initial

value, or is declared at the global level without the `extern` keyword. A function is defined when its body (code) is given.

Many C implementations prior to C89 relaxed the final rule to permit zero or one definitions of an external variable; some permitted more than one. In these implementations, the linker unified multiple definitions, and created an implicit definition for any variable (or set of linked variables) for which the program contained only declarations.

The "linkage" rules of C89 provide a way to associate names in one file with names in another file. The rules are most easily understood in terms of their implementation. Most language-independent linkers are designed to deal with *symbols*: character-string names for locations in a machine-language program. The linker's job is to assign every symbol a location in the final program, and to embed the address of the symbol in every machine-language instruction that makes a reference to it. To do this job, the linker needs to know which symbols can be used to resolve unbound references in other files, and which are local to a given file. C89 rules suffice to provide this information. For the programmer, however, there is no formal notion of *interface*, and no mechanism to make a name visible in some, but not all files. Moreover, nothing ensures that the declarations of an external object found in different files will be compatible: it is entirely possible, for example, to declare an external variable as a multifield record in one file and as a floating-point number in another. The compiler is not required to catch such errors, and the resulting bugs can be very difficult to find.

Header Files

Fortunately, C programmers have developed conventions on the use of external declarations that tend to minimize errors in practice. These conventions rely on the *file inclusion* facility of a macro preprocessor. The programmer creates files in pairs that correspond roughly to the interface and the implementation of a module. The name of an interface file ends with `.h`; the name of the corresponding implementation file ends with `.c`. Every object *defined* in the `.c` file is *declared* in the `.h` file. At the beginning of the `.c` file, the programmer inserts a directive that is treated as a special form of comment by the compiler, but that causes the preprocessor to include a verbatim copy of the corresponding `.h` file. This inclusion operation has the effect of placing "forward" declarations of all the module's objects at the beginning of its implementation file. Any inconsistencies with definitions later in the file will result in error messages from the compiler. The programmer also instructs the preprocessor at the top of each `.c` file to include a copy of the `.h` files for all of the modules on which the `.c` file depends. As long as the preprocessor includes identical copies of a given `.h` file in all the `.c` files that use its module, no inconsistent declarations will occur. Unfortunately, it is easy to forget to recompile one or more `.c` files when a `.h` file is changed, and this can lead to very subtle bugs. Tools like Unix's `make` utility help minimize such errors by keeping track of the dependences among modules.

Namespaces

Even with the convention of header files, C89 still suffers from the lack of scoping beyond the level of an individual file. In particular, all global names must be distinct, across all files of a program, and all libraries to which it links. Some coding standards encourage programmers to embed a module's name in the name of each of its external objects (e.g., `scanner_nextSym`), but this practice can be awkward, and is far from universal.

To address this limitation, C++ introduced a `namespace` mechanism that generalizes the scoping already provided for classes and functions, breaks the tie between module and compilation unit, and strengthens the interface conventions of `.h` files. Any collection of names can be declared inside a `namespace`:

EXAMPLE 3.50
Namespaces in C++

```
namespace foo {
    class foo_type_1;           // declaration
    ...
}
```

Actual definitions of the objects within `foo` can then appear in any file:

```
class foo::foo_type_1 { ...     // full definition
```

Definitions of objects declared in different namespaces can appear in the same file if desired.

A C++ programmer can access the objects in a namespace using *fully qualified* names, or by *importing* (`using`) them explicitly:

EXAMPLE 3.51
Using names from another namespace

```
foo::foo_type_1 my_first_obj;
```

or

```
using foo::foo_type_1;
...
foo_type_1 my_first_obj;
```

or

```
using namespace foo;    // import everything from foo
...
foo_type_1 my_first_obj;
```

There is no notion of export; all objects with external linkage in a namespace are visible elsewhere if imported or accessed with their qualified name. Note that linkage remains the foundation for separate compilation: `.h` files are merely a convention.

3.8.2 Packages and Automatic Header Inference

EXAMPLE 3.52
Packages in Java

The separate compilation facilities of Java and C# eliminate `.h` files. Java introduces a formal notion of module, called a *package*. Every *compilation unit*, which may be a file or (in some implementations) a record in a database, belongs to exactly one package, but a package may consist of many compilation units, each of which begins with an indication of the package to which it belongs:

```
package foo;
public class FooType1 { ...
```

Unless explicitly declared as `public`, a class in Java is visible in all and only those compilation units that belong to the same package.

EXAMPLE 3.53
Using names from another package

As in C++, a compilation unit that needs to use classes from another package can access them using fully qualified names, or via name-at-a-time or package-at-a-time import:

```
foo.FooType1 myFirstObj;
```

or

```
import foo.FooType1;
...
FooType1 myFirstObj;
```

or

```
import foo.*;           // import everything from foo
...
FooType1 myFirstObj;
```

When asked to import names from package *M*, the Java compiler will search for *M* in a standard (but implementation-dependent) set of places, and will recompile it if appropriate (i.e., if only source code is found, or if the target code is out of date). The compiler will then *automatically* extract the information that would have been needed in a C++ `.h` file or an Ada or Modula-3 header. If the compilation of *M* requires other packages, the compiler will search for them as well, recursively.

C# follows Java's lead in extracting header information automatically from complete class definitions. Its module-level syntax, however, is based on the namespaces of C++, which allow a single file to contain fragments of multiple namespaces. There is also no notion of standard search path in C#: to build a complete program, the programmer must provide the compiler with a complete list of all the files required.

To mimic the software engineering practice of early header file construction, a Java or C# design team can create skeleton versions of (the public classes of) its packages or namespaces, which can then be used, concurrently and independently, by the programmers responsible for the full versions.

3.8.3 Module Hierarchies

In Modula and Ada, the programmer can create a hierarchy of modules within a single compilation unit by means of lexical nesting (module `C`, for example, may be declared inside of module `B`, which in turn is declared inside of module `A`). In a similar vein, the Ada 95, Java, or C# programmer can create a hierarchy of separately compiled modules by means of *multipart names:*

EXAMPLE 3.54
Multipart package names

```
package A.B is ...          -- Ada 95

package A.B; ...            // Java

namespace A.B { ...         // C#
```

In these examples package `A.B` is said to be a *child* of package `A`. In Ada 95 and C# the child behaves as though it had been nested inside of the parent, so that all the names in the parent are automatically visible. In Java, by contrast, multipart names work by convention only: there is no special relationship between packages `A` and `A.B`. If `A.B` needs to refer to names in `A`, then `A` must make them public, and `A.B` must import them. Child packages in Ada 95 are reminiscent of derived classes in C++, except that they support a module-as-manager style of abstraction, rather than a module-as-type style (more on this in sidebar 10.3).

CHECK YOUR UNDERSTANDING

48. What purpose(s) does separate compilation serve?
49. What does it mean for an external variable to be *linked* in C?
50. Summarize the C conventions for use of `.h` and `.c` files.
51. Describe the difference between a compilation unit and a C++ or C# *namespace*.
52. Explain why Ada and similar languages separate the header of a module from its body. Explain how Java and C# get by without.

DESIGN & IMPLEMENTATION

3.12 Separate compilation

The evolution of separate compilation mechanisms from early C and Fortran, through C++, Modula-3, Ada, and finally Java and C#, reflects a move from an implementation-centric viewpoint to a more programmer-centric viewpoint. Interestingly, the ability to have zero definitions of an externally linked variable in certain early implementations of C is inherited from Fortran: the assembly language mnemonic corresponding to a declaration without a definition is `.common` (for a mechanism known as `common` blocks in Fortran).

3 Names, Scopes, and Bindings

3.10 Exercises

3.24 Assuming a LeBlanc-Cook style symbol table, explain how the compiler finds the symbol table information (e.g., the type) of a complicated reference such as `my_firm->revenues[1999]`.

3.25 Show the contents of a LeBlanc-Cook style symbol table that captures the referencing environment of

(a) function `F1` in Figure 3.4.

(b) procedure `set_seed` in Figure 3.7.

3.26 In Example C-3.45 we suggested that the implementor of a language in which declarations are visible only in the remainder of the current code block might choose to introduce a new nested scope for every declaration. This would, of course, lead to a very deep scope stack. If that turned out to be a performance problem for the compiler, explain how you might layer a caching mechanism on top of the standard lookup algorithm to eliminate most of the slow-down.

3.27 Consider the visibility of class members (fields and methods) in an object-oriented language, as discussed near the end of Section C-3.4.1. Describe a mechanism that could be used to check visibility after first locating the member in a more traditional symbol table. (You may want to look ahead to Section 10.2.2.)

3.28 Show a trace of the contents of the referencing environment A-list during execution of the program in

(a) Figure 3.9. Assume that a positive value is read at line 8.

(b) Exercise 3.14.

3.29 Repeat the previous exercise for a central reference table.

3.30 Consider the following tiny program in C:

```
void hello() {
    printf("Hello, world\n");
}

int main() {
    hello();
}
```

(a) Split the program into two separately compiled files, `tiny.c` and `hello.c`. Be sure to create a header file `hello.h` and include it correctly in `tiny.c`.

(b) Reconsider the program as C++ code. Put the `hello` function in a separate namespace, and include an appropriate `using` declaration in `tiny.c`.

(c) Rewrite the program in Java, with `main` and `hello` in separate packages.

3.31 Consider the following file from some larger C program:

```
int a;
extern int b;
static int c;

void foo() {
    int a;
    static int b;
    extern int c;
    extern int d;
}

static int b;
extern int c;
```

For each variable declaration, indicate whether the variable has external linkage, internal (file-level) linkage, or no linkage (i.e., is local).

3.32 Modula-2 provides no way to divide the header of a module into a public part and a private part: everything in the header is visible to the users of the module. Is this a major shortcoming? Are there disadvantages to the public/private division (e.g., as in Ada)? (For hints, see Section 10.2.)

3 Names, Scopes, and Bindings

3.11 Explorations

3.44 Using your favorite compiler, generate assembly language for some simple programs with debugger support enabled (on a Unix system, this will probably require the `-g` and `-S` command-line switches). Look through the result for debugger information. Can you decipher any of it? You may want to look ahead to Section 16.3.2, and to consult a manual for your system's object file format (on a modern Unix system, look for documentation on DWARF).

3.45 Learn about the *reflection* mechanisms of Java, C#, Prolog, Perl, PHP, Tcl, Python, or Ruby, all of which allow a program to inspect and reason about its own symbol table at run time. How complete are these mechanisms? (For example, can a program inspect symbols that aren't currently in scope?) What is reflection good for? What uses should be considered good or bad programming practice? For more ideas, see Section 16.3.1.

3.46 Learn about the `typeglob` mechanism of Perl, which allows a program to modify its own symbol table at run time. What are `typeglobs` good for? (See Sidebar 14.9 for some initial pointers.)

3.47 Create a C program in which a variable is exported from one file and imported by another, but the declarations in the files disagree with respect to type. You should be able to arrange for the program to compile and link successfully, but behave incorrectly. Try the same thing in Ada or C++. What happens?

3.48 Investigate the use of module hierarchies in the standard libraries of C++, Java, and C#. How is each organized? How fine grain is the division into separate headers or packages? Can you suggest an explanation for any major differences you find?