Chapter 6

The Procedure Abstraction

CHAPTER OVERVIEW

Procedures play a critical role in the development of software systems. They provide abstractions for control flow and naming. They provide basic information hiding. They are the building block on which systems provide interfaces. They are one of the principal forms of abstraction in Algol-like languages; object-oriented languages rely on procedures to implement their methods or code members.

This chapter provides an in-depth look at the implementation of procedures and procedure calls, from the perspective of a compiler writer. Along the way, it highlights the implementation similarities and differences between Algol-like languages and object-oriented languages.

Keywords: Procedure Calls, Parameter Binding, Linkage Conventions

6.1 INTRODUCTION

The procedure is one of the central abstractions in most modern programming languages. Procedures create a controlled execution environment; each procedure has its own private named storage. Procedures help define interfaces between system components; cross-component interactions are typically structured through procedure calls. Finally, procedures are the basic unit of work for most compilers. A typical compiler processes a collection of procedures and produces code for them that will link and execute correctly with other collections of compiled procedures.

This latter feature, often called separate compilation, allows us to build large software systems. If the compiler needed the entire text of a program for each compilation, large software systems would be untenable. Imagine recompiling a multimillion line application for each editing change made during

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00006-2
Copyright © 2012, Elsevier Inc. All rights reserved.

development! Thus, procedures play as critical a role in system design and engineering as they do in language design and compiler implementation. This chapter focuses on how compilers implement the procedure abstraction.

Conceptual Roadmap

To translate a source-language program into executable code, the compiler must map all of the source-language constructs that the program uses into operations and data structures on the target processor. The compiler needs a strategy for each of the abstractions supported by the source language. These strategies include both algorithms and data structures that are embedded into the executable code. These runtime algorithms and data structures combine to implement the behavior dictated by the abstraction. These runtime strategies also require support at compile time in the form of algorithms and data structures that run inside the compiler.

This chapter explains the techniques used to implement procedures and procedure calls. Specifically, it examines the implementation of control, of naming, and of the call interface. These abstractions encapsulate many of the features that make programming languages usable and that enable construction of large-scale systems.

Overview

The procedure is one of the central abstractions that underlie most modern programming languages. Procedures create a controlled execution environment. Each procedure has its own private named storage. Statements executed inside the procedure can access the private, or local, variables in that private storage. A procedure executes when it is invoked, or called, by another procedure (or the operating system). The callee may return a value to its caller, in which case the procedure is termed a *function*. This interface between procedures lets programmers develop and test parts of a program in isolation; the separation between procedures provides some insulation against problems in other procedures.

Callee
In a procedure call, we refer to the procedure that is invoked as the *callee*.

Caller
In a procedure call, we refer to the calling procedure as the *caller*.

Procedures play an important role in the way that programmers develop software and that compilers translate programs. Three critical abstractions that procedures provide allow the construction of nontrivial programs.

1. *Procedure Call Abstraction* Procedural languages support an abstraction for procedure calls. Each language has a standard mechanism to invoke a procedure and map a set of arguments, or parameters, from the caller's name space to the callee's name space. This abstraction typically includes a mechanism to return control to the

caller and continue execution at the pointimmediately after the call. Most languages allow a procedure to return one or more values to the caller. The use of standard linkage conventions, sometimes referred to as *calling sequences*, lets the programmer invoke code written and compiled by other people and at other times; it lets the application invoke library routines and system services.

Linkage convention
an agreement between the compiler and operating system that defines the actions taken to call a procedure or function

2. *Name Space* In most languages, each procedure creates a new and protected name space. The programmer can declare new names, such as variables and labels, without concern for the surrounding context. Inside the procedure, those local declarations take precedence over any earlier declarations for the same names. The programmer can create parameters for the procedure that allow the caller to map values and variables in the caller's name space into formal parameters in the callee's name space. Because the procedure has a known and separate name space, it can function correctly and consistently when called from different contexts. Executing a call instantiates the callee's name space. The call must create storage for the objects declared by the callee. This allocation must be both automatic and efficient—a consequence of calling the procedure.

Actual parameter
A value or variable passed as a parameter at a call site is an *actual parameter* of the call.

Formal parameter
A name declared as a parameter of some procedure *p* is a *formal parameter* of *p*.

3. *External Interface* Procedures define the critical interfaces among the parts of large software systems. The linkage convention defines rules that map names to values and locations, that preserve the caller's runtime environment and create the callee's environment, and that transfer control from caller to callee and back. It creates a context in which the programmer can safely invoke code written by other people. The existence of uniform calling sequences allows the development and use of libraries and system calls. Without a linkage convention, both the programmer and the compiler would need detailed knowledge about the implementation of the callee at each procedure call.

Thus, the procedure is, in many ways, the fundamental abstraction that underlies Algol-like languages. It is an elaborate facade created collaboratively by the compiler and the underlying hardware, with assistance from the operating system. Procedures create named variables and map them to virtual addresses; the operating system maps virtual addresses to physical addresses. Procedures establish rules for visibility of names and addressability; the hardware typically provides several variants of load and store operations. Procedures let us decompose large software systems into components; linkers and loaders knit these together into an executable program that the hardware can execute by advancing its program counter and following branches.

A WORD ABOUT TIME

This chapter deals with both compile-time and runtime mechanisms. The distinction between events that occur at compile time and those that occur at runtime can be confusing. The compiler generates all the code that executes at runtime. As part of the compilation process, the compiler analyzes the source code and builds data structures that encode the results of the analysis. (Recall the discussion of lexically scoped symbol tables in Section 5.5.3.) The compiler determines much of the storage layout that the program will use at runtime. It then generates the code needed to create that layout, to maintain it during execution, and to access both data objects and code in memory. When the compiled code runs, it accesses data objects and calls procedures or methods. All of the code is generated at compile time; all of the accesses occur at runtime.

A large part of the compiler's task is putting in place the code needed to realize the various pieces of the procedure abstraction. The compiler must dictate the layout of memory and encode that layout in the generated program. Since it may compile the different components of the program at different times, without knowing their relationships to one another, this memory layout and all the conventions that it induces must be standardized and uniformly applied. The compiler must also use the various interfaces provided by the operating system, to handle input and output, manage memory, and communicate with other processes.

This chapter focuses on the procedure as an abstraction and the mechanisms that the compiler uses to establish its control abstraction, name space, and interface to the outside world.

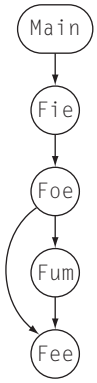
6.2 PROCEDURE CALLS

In Algol-like languages (ALLs), procedures have a simple and clear call/return discipline. A procedure call transfers control from the call site in the caller to the start of the callee; on exit from the callee, control returns to the point in the caller that immediately follows its invocation. If the callee invokes other procedures, they return control in the same way. Figure 6.1a shows a Pascal program with several nested procedures, while Figures 6.1b and 6.1c show the program's call graph and its *execution history*, respectively.

The call graph shows the set of potential calls among the procedures. Executing `Main` can result in two calls to `Fee`: one from `Foe` and another from `Fum`. The execution history shows that both calls occur at runtime. Each

```
program Main(input, output);
   var x,y,z: integer;
   procedure Fee;
      var x: integer;
      begin { Fee }
         x := 1;
         y := x * 2 + 1
      end;
   procedure Fie;
      var y: real;
      procedure Foe;
         var z: real;
            procedure Fum;
               var y: real;
               begin { Fum }
                  x := 1.25 * z;
                  Fee;
                  writeln('x = ',x)
               end;
         begin { Foe }
            z := 1;
            Fee;
            Fum
         end;
      begin { Fie }
         Foe;
         writeln('x = ',x)
      end;
   begin { Main }
      x := 0;
      Fie
   end.
```

(a) Example Pascal Program



(b) Call Graph

1. `Main` calls `Fie`
2. `Fie` calls `Foe`
3. `Foe` calls `Fee`
4. `Fee` returns to `Foe`
5. `Foe` calls `Fum`
6. `Fum` calls `Fee`
7. `Fee` returns to `Fum`
8. `Fum` returns to `Foe`
9. `Foe` returns to `Fie`
10. `Fie` returns to `Main`

(c) Execution History

■ **FIGURE 6.1** Nonrecursive Pascal Program and Its Execution History.

of these calls creates a distinct instance, or *activation*, of `Fee`. By the time that `Fum` is called, the first instance of `Fee` is no longer active. It was created by the call from `Foe` (event 3 in the execution history), and destroyed after it returned control back to `Foe` (event 4). When control returns to `Fee`, from the call in `Fum` (event 6), it creates a new activation of `Fee`. The return from `Fee` to `Fum` destroys that activation.

Activation
A call to a procedure *activates* it; thus, we call an instance of its execution an *activation*.

```
(define (fact k)
  (cond
    [(<= k 1) 1]
    [else (* (fact (sub1 k)) k)]
  ))
```

■ **FIGURE 6.2** Recursive Factorial Program in Scheme.

When the program executes the assignment `x:=1` in the first invocation of `Fee`, the active procedures are `Fee`, `Foe`, `Fie`, and `Main`. These all lie on a path in the call graph from `Main` to `Fee`. Similarly, when it executes the second invocation of `Fee`, the active procedures (`Fee`, `Fum`, `Foe`, `Fie`, and `Main`) lie on a path from `Main` to `Fee`. Pascal's call and return mechanism ensures that, at any point during execution, the procedure activations instantiate some rooted path through the call graph.

When the compiler generates code for calls and returns, that code must preserve enough information so that calls and returns operate correctly. Thus, when `Foe` calls `Fum`, the code must record the address in `Foe` to which `Fum` should return control. `Fum` may *diverge*, or not return, due to a runtime error, an infinite loop, or a call to another procedure that does not return. Still, the call mechanism must preserve enough information to allow execution to resume in `Foe` if `Fum` returns.

Diverge
A computation that does not terminate normally is said to *diverge*.

Return address
When *p* calls *q*, the address in *p* where execution should continue after *p*'s return is called its *return address*.

The call and return behavior of ALLs can be modelled with a stack. As `Fie` calls `Foe`, it pushes the return address in `Fie` onto the stack. When `Foe` returns, it pops that address off the stack and jumps to the address. If all procedures use the same stack, popping a return address exposes the next one.

The stack mechanism handles recursion as well. The call mechanism, in effect, unrolls the cyclic path through the call graph and creates a distinct activation for each call to a procedure. As long as the recursion terminates, this path will be finite and the stack of return addresses will correctly capture the program's behavior.

To make this concrete, consider the recursive factorial computation shown in Figure 6.2. When invoked to compute `(fact 5)`, it generates a series of recursive calls: `(fact 5)` calls `(fact 4)` calls `(fact 3)` calls `(fact 2)` calls `(fact 1)`. At that point, the `cond` statement executes the clause for `(<= k 1)`, terminating the recursion. The recursion unwinds in the reverse order, with the call to `(fact 1)` returning the value 1 to `(fact 2)`. It, in turn, returns the value 2 to `(fact 3)`, which returns 6 to `(fact 4)`. Finally, `(fact 4)` returns 24 to `(fact 5)`, which multiplies 24 times 5 to return the answer 120. The recursive program exhibits last-in, first-out behavior, so the stack mechanism correctly tracks all of the return addresses.

Control Flow in Object-Oriented Languages

From the perspective of procedure calls and returns, object-oriented languages (OOLs) are similar to ALLs. The primary differences between procedure calls in an OOL and an ALL lie in the mechanism used to name the callee and in the mechanisms used to locate the callee at runtime.

More Complex Control Flow

Following Scheme, many programming languages allow a program to encapsulate a procedure and its runtime context into an object called a *closure*. When the closure is invoked, the procedure executes in the encapsulated runtime context. A simple stack is inadequate to implement this control abstraction. Instead, the control information must be saved in some more general structure that can represent the more complex control-flow relationship. Similar problems arise if the language allows references to local variables that outlast a procedure's activation.

Closure
a procedure and the runtime context that defines its free variables

SECTION REVIEW

In Algol-like languages, procedures are invoked with a call and they terminate in a return, unless the procedure diverges. To translate calls and returns, the compiler must arrange for the code to record, at each call, the appropriate return address and to use, at each return, the return address that corresponds to the correct call. Using a stack to hold return addresses correctly models the last-in, first-out behavior of return addresses.

One key data structure used to analyze caller–callee relationships is the call graph. It represents the set of calls between procedures, with an edge from `Foe` to `Fum` for each call site in `Foe` that invokes `Fum`. Thus, it captures the static relationship between callers and callees defined by the source code. It does not capture the dynamic, or runtime, relationship between procedures; for example, it cannot tell how many times the recursive factorial program in Figure 6.2 calls itself.

Review Questions

1. Many programming languages include a direct transfer of control, often called a `goto`. Compare and contrast a procedure call and a `goto`.
2. Consider the factorial program shown in Figure 6.2. Write down the execution history of a call to `(fact 5)`. Explicitly match up the calls and returns. Show the value of `k` and of the return value.

6.3 NAME SPACES

Scope
In an Algol-like language, *scope* refers to a name space. The term is often used in discussions of the visibility of names.

In most procedural languages, a complete program will contain multiple name spaces. Each name space, or *scope*, maps a set of names to a set of values and procedures over some set of statements in the code. This range might be the whole program, some collection of procedures, a single procedure, or a small set of statements. The scope may inherit some names from other scopes. Inside a scope, the programmer can create names that are inaccessible outside the scope. Creating a name, `fee`, inside a scope can obscure definitions of `fee` in surrounding scopes, in effect making them inaccessible inside the scope. Thus, scope rules give the programmer control over access to information.

6.3.1 Name Spaces of Algol-like Languages

Most programming languages inherit many of the conventions that were defined for Algol 60. This is particularly true of the rules that govern the visibility of names. This section explores the notion of naming that prevails in ALLs, with particular emphasis on the hierarchical scope rules that apply in such languages.

Nested Lexical Scopes

Lexical scope
Scopes that nest in the order that they are encountered in the program are often called *lexical scopes*.

In lexical scoping, a name refers to the definition that is lexically closest to its use—that is, the definition in the closest surrounding scope.

Most ALLs allow the programmer to nest scopes inside one another. The limits of a scope are marked by specific terminal symbols in the programming language. Typically, each new procedure defines a scope that covers its entire definition. Pascal demarcated scopes with a `begin` at the start and an `end` at the finish. C uses curly braces, { and }, to begin and end a *block*; each block defines a new scope.

Pascal popularized nested procedures. Each procedure defines a new scope, and the programmer can declare new variables and procedures in each scope. It uses the most common scoping discipline, called *lexical scoping*. The general principle behind lexical scoping is simple:

> *In a given scope, each name refers to its lexically closest declaration.*

Thus, if *s* is used in the current scope, it refers to the *s* declared in the current scope, if one exists. If not, it refers to the declaration of *s* that occurs in the closest enclosing scope. The outermost scope contains global variables.

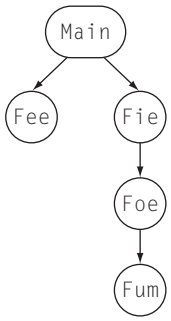
To make lexical scoping concrete, consider the Pascal program shown in Figure 6.3. It contains five distinct scopes, one corresponding to the program `Main` and one for each of the procedures `Fee`, `Fie`, `Foe`, and `Fum`. Each procedure declares some set of variables drawn from the set of names `x`, `y`, and `z`.

```
program Main0(input, output);
  var x1,y1,z1: integer;
  procedure Fee1;
     var x2: integer;
     begin { Fee1 }
        x2 := 1;
        y1 := x2 * 2 + 1
     end;
  procedure Fie1;
     var y2: real;
     procedure Foe2;
        var z3: real;
           procedure Fum3
              var y4: real;
              begin { Fum3 }
                 x1 := 1.25 * z3;
                 Fee1;
                 writeln('x = ',x1)
              end;
        begin { Foe2 }
           z3 := 1;
           Fee1;
           Fum3
        end;
     begin { Fie1 }
        Foe2;
        writeln('x = ',x1)
     end;
  begin { Main0 }
     x1 := 0;
     Fie1
  end.
```

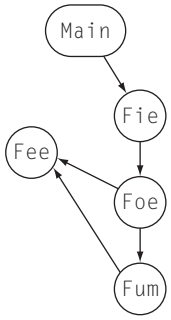
(a) Pascal Program

Scope	x	y	z
Main	⟨1,0⟩	⟨1,4⟩	⟨1,8⟩
Fee	⟨2,0⟩	⟨1,4⟩	⟨1,8⟩
Fie	⟨1,0⟩	⟨2,0⟩	⟨1,8⟩
Foe	⟨1,0⟩	⟨2,0⟩	⟨3,0⟩
Fum	⟨1,0⟩	⟨4,0⟩	⟨3,0⟩

(b) Static Coordinates



(c) Nesting Relationships



(d) Calling Relationships

■ **FIGURE 6.3** Nested Lexical Scopes in Pascal.

The figure shows each name with a subscript that indicates its level number. Names declared in a procedure always have a level that is one more than the level of the procedure name. Thus, if `Main` has level 0, as shown, names declared directly in `Main`, such as `x`, `y`, `z`, `Fee`, and `Fie` all have level 1.

To represent names in a lexically scoped language, the compiler can use the *static coordinate* for each name. The static coordinate is a pair ⟨*l*,*o*⟩, where

Static coordinate
For a name x declared in scope *s*, its static coordinate is a pair ⟨*l*,*o*⟩ where *l* is the lexical nesting level of *s* and *o* is the offset where x is stored in the scope's data area.

DYNAMIC SCOPING

The alternative to lexical scoping is dynamic scoping. The distinction between lexical and dynamic scoping only matters when a procedure refers to a variable that is declared outside the procedure's own scope, often called a *free variable*.

With lexical scoping, the rule is simple and consistent: a free variable is bound to the declaration for its name that is lexically closest to the use. If the compiler starts in the scope containing the use, and checks successive surrounding scopes, the variable is bound to the first declaration that it finds. The declaration always comes from a scope that encloses the reference.

With dynamic scoping, the rule is equally simple: a free variable is bound to the variable by that name that was most recently created at runtime. Thus, when execution encounters a free variable, it binds that free variable to the most recent instance of that name. Early implementations created a runtime stack of names, on which every name was pushed as its declaration was encountered. To bind a free variable, the running code searched the name stack from its top downward until a variable with the right name was found. Later implementations are more efficient.

While many early Lisp systems used dynamic scoping, lexical scoping has become the dominant choice. Dynamic scoping is easy to implement in an interpreter and somewhat harder to implement efficiently in a compiler. It can create bugs that are difficult to detect and hard to understand. Dynamic scoping still appears in some languages; for example, Common Lisp still allows the program to specify dynamic scoping.

l is the name's lexical nesting level and o is the its offset in the data area for level l. To obtain l, the front end uses a lexically scoped symbol table, as described in Section 5.5.3. The offset, o, should be stored with the name and its level in the symbol table. (Offsets can be assigned when declarations are processed during context-sensitive analysis.) The table on the right side of Figure 6.3 shows the static coordinate for each variable name in each procedure.

The second part of name translation occurs during code generation. The compiler must use the static coordinate to locate the value at runtime. Given a coordinate $\langle l,o \rangle$, the code generator must emit code that translates l into the runtime address of the appropriate data area. Then, it can use the offset o to compute the address for the variable corresponding to $\langle l,o \rangle$. Section 6.4.3 describes two different ways to accomplish this task.

Scope Rules across Various Languages

Programming language scope rules vary idiosyncratically from language to language. The compiler writer must understand the specific rules of a source language and must adapt the general translation schemes to work with these specific rules. Most ALLs have similar scope rules. Consider the rules for the languages FORTRAN, C, and Scheme:

- FORTRAN has a simple name space. A FORTRAN program creates a single global scope, along with a local scope for each procedure or function. Global variables are grouped together in a "common block"; each common block consists of a name and a list of variables. The global scope holds the names of procedures and common blocks. Global names have lifetimes that match the lifetime of the program. A procedure's scope holds parameter names, local variables, and labels. Local names obscure global names if they conflict. Names in the local scope have, by default, lifetimes that match an invocation of the procedure, The programmer can give a local variable the lifetime of a global variable by listing it in a `save` statement.

  Separate compilation makes it hard for FORTRAN compilers to detect different declarations for a common block in distinct files. Thus, the compiler must translate common-block references into ⟨*block*, *offset*⟩ pairs to produce correct behavior.

- C has more complex rules. A C program has a global scope for procedure names and global variables. Each procedure has a local scope for variables, parameters, and labels. The language definition does not allow nested procedures, although some compilers have implemented this feature as an extension. Procedures can contain blocks (set off with left and right braces) that create separate local scopes; blocks can be nested. Programmers often use a block-level scope to create temporary storage for code generated by a preprocessor macro or to create a local variable whose scope is the body of a loop.
  C introduces another scope: the file-level scope. This scope includes names declared as `static` that not enclosed in a procedure. Thus, `static` procedures and functions are in the file-level scope, as are any `static` variables declared at the outermost level in the file. Without the `static` attribute, these names would be global variables. Names in the file-level scope are visible to any procedure in the file, but are not visible outside the file. Both variables and procedures can be declared static.

  Static name
  A variable declared as *static* retains its value across invocations of its defining procedure.

  Variables that are not static are called *automatic*.

- Scheme has a simple set of scope rules. Almost all objects in Scheme reside in a single global space. Objects can be data or executable expressions. System-provided functions, such as `cons`, live alongside user-written code and data items. Code, which consists of an executable expression, can create private objects by using a `let` expression. Nesting `let` expressions inside one another can create nested lexical scopes of arbitrary depth.

6.3.2 Runtime Structures to Support Algol-like Languages

Activation record
a region of storage set aside to hold control information and data storage associated with a single instance of a single procedure

To implement the twin abstractions of procedure calls and scoped name spaces, the translation must establish a set of runtime structures. The key data structure involved in both control and naming is the *activation record* (AR), a private block of memory associated with a specific invocation of a specific procedure. In principle, every procedure call gives rise to a new AR.

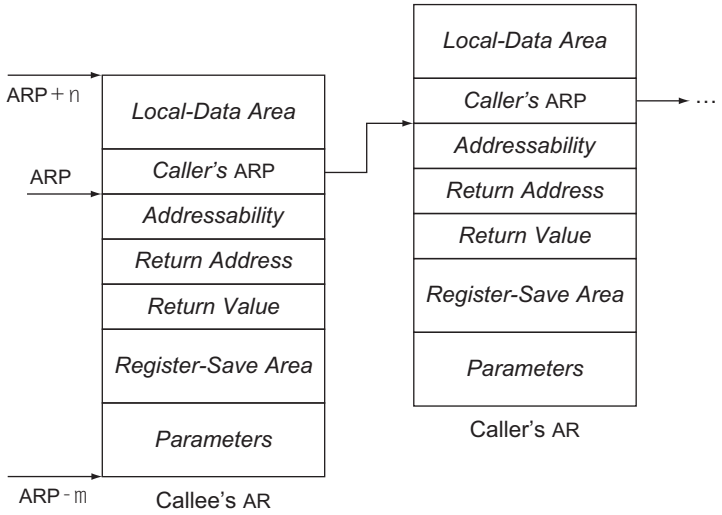
- The compiler must arrange for each call to store the return address where the callee can find it. The return address goes into the AR.
- The compiler must map the actual parameters at the call site into the formal parameter names by which they are known in the callee. To do so, it stores ordered parameter information in the AR.
- The compiler must create storage space for variables declared in the callee's local scope. Since these values have lifetimes that match the lifetime of the return address, it is convenient to store them in the AR.
- The callee needs other information to connect it to the surrounding program, and to allow it to interact safely with other procedures. The compiler arranges to store that information in the callee's AR.

Since each call creates a new AR, when multiple instances of a procedure are active, each has its own AR. Thus, recursion gives rise to multiple ARs, each of which holds the local state for a different invocation of the recursive procedure.

Activation record pointer
To locate the current AR the compiler arranges to keep a pointer to the AR, the *activation record pointer*, in a designated register.

Figure 6.4 shows how the contents of an AR might be laid out. The entire AR is addressed through an *activation record pointer* (ARP), with various fields in the AR found at positive and negative offsets from the ARP. The ARs in Figure 6.4 have a number of fields.

- The parameter area holds actual parameters from the call site, in an order that corresponds to their order of appearance at the call.
- The register save area contains enough space to hold registers that the procedure must preserve due to procedure calls.
- The return-value slot provides space to communicate data from the callee back to the caller, if needed.
- The return-address slot holds the runtime address where execution should resume when the callee terminates.
- The "addressability" slot holds information used to allow the callee to access variables in surrounding lexical scopes (not necessarily the caller).
- The slot at the callee's ARP stores the caller's ARP. The callee needs this pointer so that it can restore the caller's environment when it terminates.
- The local data area holds variables declared in the callee's local scope.



■ **FIGURE 6.4** Typical Activation Records.

For the sake of efficiency, some of the information shown in Figure 6.4 may be kept in dedicated registers.

Local Storage

The AR for an invocation of procedure *q* holds the local data and state information for that invocation. Each separate call to *q* generates a separate AR. All data in the AR is accessed through the ARP. Because procedures typically access their AR frequently, most compilers dedicate a hardware register to hold the ARP of the current procedure. In ILOC, we refer to this dedicated register as r_{arp}.

The ARP always points to a designated location in the AR. The central part of the AR has a static layout; all the fields have known fixed lengths. This ensures that the compiled code can access those items at fixed offsets from the ARP. The ends of the AR are reserved for storage areas whose sizes may change from one invocation to another; typically one holds parameter storage while the other holds local data.

Reserving Space for Local Data

Each local data item may need space in the AR. The compiler should assign each such item an appropriately sized area and record the current lexical level and its offset from the ARP in the symbol table. This pair, the lexical level and offset, become the item's static coordinate. Then, the variable can be accessed using an operation like `loadAO`, with r_{arp} and the offset as its arguments, to provide efficient access to local variables.

The compiler may not know the sizes of some local variables at compile time. For example, the program might read the size of an array from external media or determine it from work done in an earlier phase of the computation. For such variables, the compiler can leave space in the local data area for a pointer to the actual data or to a descriptor for an array (see Section 7.5.3 on page 362). The compiler arranges to allocate the actual storage elsewhere, at runtime, and to fill the reserved slot with the address of the dynamically allocated memory. In this case, the static coordinate leads the compiler to the pointer's location, and the actual access either uses the pointer directly or uses the pointer to calculate an appropriate address in the variable-length data area.

Initializing Variables

If the source language allows the program to specify an initial value for a variable, the compiler must arrange for that initialization to occur. If the variable is allocated statically—that is, it has a lifetime that is independent of any procedure—and the initial value is known at compile time, the data can be inserted directly into the appropriate locations by the loader. (Static variables are usually stored outside all ARs. Having one instance of such a variable provides the needed semantics—a single value preserved across all the calls. Using a separate static data area—either one per procedure or one for the entire program—lets the compiler use the initialization features commonly found in loaders.)

Local variables, on the other hand, must be initialized at runtime. Because a procedure may be invoked multiple times, the only feasible way to set initial values is to generate instructions that store the necessary values to the appropriate locations. In effect, these initializations are assignments that execute before the procedure's first statement, each time it is invoked.

Space for Saved Register Values

When p calls q, one of them must save the register values that p needs after the call. It may be necessary to save all the register values; on the other hand, a subset may suffice. On return to p, these saved values must be restored. Since each activation of p stores a distinct set of values, it makes sense to store these saved registers in the AR of either p or q, or both. If the callee saves a register, its value is stored in the callee's register save area. Similarly, if the caller saves a register, its value is stored in the caller's register save area. For a caller p, only one call inside p can be active at a time. Thus, a single register save area in p's AR suffices for all the calls that p can make.

Allocating Activation Records

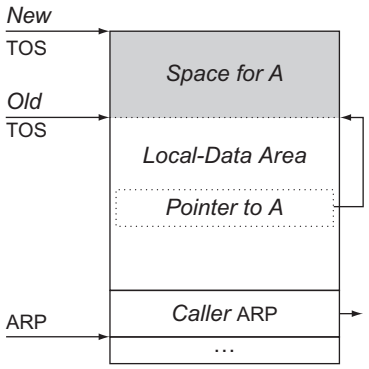
When p calls q at runtime, the code that implements the call must allocate an AR for q and initialize it with the appropriate values. If all the fields shown in Figure 6.4 are stored in memory, then the AR must be available to the caller, p, so that it can store the actual parameters, return address, caller's ARP, and addressability information. This forces allocation of q's AR into p, where the size of its local data area may not be known. On the other hand, if these values are passed in registers, actual allocation of the AR can be performed in the callee, q. This lets q allocate the AR, including any space required for the local data area. After allocation, it may store into its AR some of the values passed in registers.

The compiler writer has several options for allocating activation records. This choice affects both the cost of procedure calls and the cost of implementing advanced language features, such as building a closure. It also affects the total amount of memory needed for activation records.

Stack Allocation of Activation Records

In many cases, the contents of an AR are only of interest during the lifetime of the procedure whose activation causes the AR's creation. In short, most variables cannot outlive the procedure that creates them, and most procedure activations cannot outlive their callers. With these restrictions, calls and returns are balanced; they follow a last-in, first-out (LIFO) discipline. A call from p to q eventually returns, and any returns that occur between the call from p to q and the return from q to p must result from calls made (either directly or indirectly) by q. In this case, the activation records also follow the LIFO ordering; thus, they can be allocated on a stack. Pascal, C, and Java are typically implemented with stack-allocated ARs.

Keeping activation records on a stack has several advantages. Allocation and deallocation are inexpensive; each requires one arithmetic operation on the value that marks the stack's top. The caller can begin the process of setting up the callee's AR. It can allocate all the space up to the local data area. The callee can extend the AR to include the local data area by incrementing the top-of-stack (TOS) pointer. It can use the same mechanism to extend the current AR incrementally to hold variable-size objects, as shown in Figure 6.5. Here, the callee has copied the TOS pointer into the local data area slot for A and then incremented the TOS pointer by the size of A. Finally, with stack-allocated ARs, a debugger can walk the stack from its top to its base to produce a snapshot of the currently active procedures.



■ **FIGURE 6.5** Stack Allocation of a Dynamically Sized Array.

Heap Allocation of Activation Records

If a procedure can outlive its caller, the stack discipline for allocating ARs breaks down. Similarly, if a procedure can return an object, such as a closure, that includes, explicitly or implicitly, references to its local variables, stack allocation is inappropriate because it will leave behind dangling pointers. In these situations, ARs can be kept in heap storage (see Section 6.6). Implementations of Scheme and ML typically use heap-allocated ARs.

A modern memory allocator can keep the cost of heap allocation low. With heap-allocated ARs, variable-size objects can be allocated as separate objects on the heap. If heap objects need explicit deallocation, then the code for procedure return must free the AR and its variable-size extensions. With implicit deallocation (see Section 6.6.2), the garbage collector frees them when they are no longer useful.

Static Allocation of Activation Records

Leaf procedure
a procedure that contains no calls

If a procedure q calls no other procedures, then q can never have multiple active invocations. We call q a *leaf procedure* since it terminates a path through a graph of the possible procedure calls. The compiler can statically allocate activation records for leaf procedures. This eliminates the runtime costs of AR allocation. If the calling convention requires the caller to save its own registers, then q's AR needs no register save area.

If the language does not allow closures, the compiler can do better than allocating a static AR for each leaf procedure. At any point during execution, only one leaf procedure can be active. (To have two such procedures active, the first one would need to call another procedure, so it would not be a leaf.) Thus, the compiler can allocate a single static AR for use by all of the leaf procedures. The static AR must be large enough to accommodate any of the

program's leaf procedures. The static variables declared in any of the leaf procedures can be laid out together in that single AR. Using a single static AR for leaf procedures reduces the space overhead of separate static ARs for each leaf procedure.

Coalescing Activation Records

If the compiler discovers a set of procedures that are always invoked in a fixed sequence, it may be able to combine their activation records. For example, if a call from *p* to *q* always results in calls to *r* and *s*, the compiler may find it profitable to allocate the ARs for *q*, *r*, and *s* at the same time. Combining ARs can save on the costs of allocation; the benefits will vary directly with allocation costs. In practice, this optimization is limited by separate compilation and the use of function-valued parameters. Both limit the compiler's ability to determine the calling relationships that actually occur at runtime.

6.3.3 Name Spaces of Object-Oriented Languages

Much has been written about object-oriented design, object-oriented programming, and object-oriented languages. Languages such as Simula, Smalltalk, C++, and Java all support object-oriented programming. Many other languages have extensions that provide them with features to support object-oriented programming. Unfortunately, the term *object-oriented* has been given so many different meanings and implementations that it has come to signify a wide range of language features and programming paradigms.

As we shall see, not all OOLs can be compiled, in the traditional sense of a translation that finalizes all of the details about the executable program. Features of some OOLs create name spaces that cannot be understood until runtime. Implementations of these languages rely on runtime mechanisms that run from interpretation to runtime compilation (so-called *just-in-time compilers* or JITs). Because interpreters and JITs use many of the same structures as a compiler, we describe the problem as it might be implemented in a traditional compiler.

Just-in-time compiler
Schemes that perform some of the tasks of a traditional compiler at runtime are often called *just-in-time* compilers or JITs.

In a JIT, compile time becomes part of runtime, so JITs place an emphasis on compile-time efficiency.

From the compiler's perspective, OOLs reorganize the program's name space. Most OOLs retain the procedure-oriented lexical scoping conventions of an ALL for use within procedural code. They augment this classic naming scheme with a second set of conventions for naming, one organized around the layout of data—specifically, the definitions of objects. This data-centric naming discipline leads to a second hierarchy of scopes and a second mechanism for resolving names—that is, for mapping a source-language name into a runtime address so that the compiled code can access the data associated with that name.

TERMINOLOGY FOR OBJECT-ORIENTED LANGUAGES

The diversity of object-oriented languages has led to some ambiguity in the terms that we use to discuss them. To make the discussion in this chapter concrete, we will use the following terms:

1. *Object* An object is an abstraction with one or more members. Those members can be data items, code that manipulates those data items, or other objects. An object with code members is a *class*. Each object has internal state—data whose lifetimes match the object's lifetime.
2. *Class* A class is a collection of objects with the same abstract structure and characteristics. A class defines the set of data members in each *instance* of the class and defines the code members (*methods*) that are local to that class. Some methods are *public*, or externally visible, others are *private*, or invisible outside the class.
3. *Inheritance* Inheritance refers to a relationship among classes that defines a partial order on the name scopes of classes. Each class may have a *superclass* from which it inherits both code and data members. If *a* is the superclass of *b*, *b* is a subclass of *a*. Some languages allow a class to have multiple superclasses.
4. *Receiver* Methods are invoked relative to some object, called the method's receiver. The receiver is known by a designated name, such as `this` or `self`, inside the method.

The complexity and the power of an OOL arise, in large part, from the organizational possibilities presented by its multiple name spaces.

The syntax and terminology used to specify subclasses varies between languages. In Java, a subclass *extends* its superclass, while in C++, a subclass is *derived* from its superclass.

Inheritance imposes an ancestor relation on the classes in an application. Each class has, by declaration, one or more parent classes, or superclasses. Inheritance changes both the name space of the application and the mapping of method names to implementations. If α is a superclass of β, then β is a subclass of α and any method defined in α must operate correctly on an object of class β, if it is visible in β. The converse is not true; a method declared in class β cannot be applied to an object of its superclass α, as the method from β may need fields present in an object of class β that are absent from an object of class α.

Visibility

When a method runs, it can reference names defined in multiple scope hierarchies. The method is a procedure, with its own name space defined by the set of lexical scopes in which it is declared; the method can access names in those scopes using the familiar conventions defined for ALLS. The method was invoked relative to some receiver; it can access that object's own members. The method is defined in the receiver's class. The method can access

the members of that class and, by inheritance, of its superclasses. Finally, the program creates some global name space and executes in it. The running method can access any names that are contained in that global name space.

To make these issues concrete, consider the abstracted example shown in Figure 6.6. It defines a class, `Point`, of objects with integer fields `x` and `y` and methods `draw` and `move`. `ColorPoint` is a subclass of `Point` that extends `Point` with an additional field `c` of type `Color`. It uses `Point`'s method for `move`, overrides its method for `draw` and defines a new method `test` that performs some computation and then invokes `draw`. Finally, class `C` defines local fields and methods and uses `ColorPoint`.

Now, consider the names that are visible inside method `m` of class `C`. Method `m` maps `x` and `y` to their declarations in `C`. It expressly references the class names `Point` and `ColorPoint`. The assignment `y=p.x` takes its right-hand side from the field `x` in the object `p`, which `p` has by inheritance from class `Point`. The left-hand side refers to `m`'s local variable `y`. The call to `draw` maps to the method defined in `ColorPoint`. Thus, `m` refers to definitions from all three classes in the example.

```
class Point {
   public int x, y;
   public void draw() {...};
   public void move() {...};
}

class ColorPoint extends Point {        // inherits x, y, & move()
   Color c;                             // local field of ColorPoint
   public void draw() {...};            // hide Point's draw()
   public void test() {...; draw();};   // local method
}

class C {
   int x, y;                            // local fields
   public void m() {                    // local method
      int y;                            // local variable of m
      Point p = new ColorPoint();       // uses ColorPoint and, by
      y = p.x                           // inheritance, Point
      p.draw()
   }
}
```

■ **FIGURE 6.6** Definitions for `Point` and `ColorPoint`.

To translate this example, the compiler must track the hierarchy of names and scopes established both by the scope rules inside methods and classes and by the hierarchy of classes and superclasses established by `extends`. Name resolution in this environment depends on both the details of the code definitions and the class structure of the data definitions. To translate an OOL, the compiler needs to model both the name space of the code and the name spaces associated with the class hierarchy. The complexity of that model depends on details of the specific OOL.

In Java, `public` makes a name visible everywhere while `private` makes the name visible only within its own class.

To add a final complication, some OOLs provide attributes for individual names that change their visibility. For example, a Java name can have the attributes `public` or `private`. Similarly, some OOLs provide a mechanism to reference names obscured by nesting. In C++, the `::` operator allows the code to name a scope while in Java the programmer can use a fully qualified name.

Naming in the Class Hierarchy

The class hierarchy defines a set of nested name scopes, just as a set of nested procedures and blocks does in an ALL. In an ALL, lexical position defines the relationship between those name scopes—if procedure *d* is declared inside procedure *c*, then *d*'s name space is nested inside *c*'s name space. In an OOL, the class declarations can be lexically disjoint and the subclass relation is specified by explicit declarations.

Direct superclass
If class α extends β, then β is α's direct superclass. If β has a superclass γ, then γ is, by transitivity, a superclass of α, but it is not α's direct superclass.

To find the declaration of a name, the compiler must search the lexical hierarchy, the class hierarchy, and the global name space. For a name `x` in a method `m`, the compiler first searches the lexical scopes that surround the reference in `m`. If that lookup fails, it searches the class hierarchy for the class that contains `m`. Conceptually, it searches `m`'s declared class, then `m`'s direct superclass, then that class' direct superclass, and so on until it finds the name or exhausts the class hierarchy. If the name is not found in either the lexical hierarchy or the class hierarchy, the compiler searches the global name space.

To support the more complex naming environment of an OOL, the compiler writer uses the same basic tools used with an ALL: a linked set of symbol tables (see Section 5.5.3). In an OOL, the compiler simply has more tables than in an ALL and it must use those tables in a way that reflects the naming environment. It can link the tables together in the appropriate order, or it can keep the three kinds of tables separate and search them in the appropriate order.

The major complication that arises with some OOLs derives not from the presence of a class hierarchy, but rather from when that hierarchy is defined. If the OOL requires that class definitions be present at compile time and that

TRANSLATING JAVA

The Java programming language was designed to be portable, to be secure, and to have a compact representation for transmission over networks. These design goals led directly to a two-stage translation scheme for Java that is followed in almost all Java implementations.

Java code is first compiled, in the traditional sense, from Java source into an IR called Java bytecode. Java bytecode is compact. Java bytecode forms the instruction set for the Java Virtual Machine (JVM). The JVM has been implemented with an interpreter that can be compiled on almost any target platform, providing portability. Because Java code executes inside the JVM, the JVM can control interactions between the Java code and the system, limiting the ability of a Java program to gain illicit access to system resources—a strong security feature.

This design implies a specific translation scheme. Java code is first compiled into Java bytecode. The bytecode is then interpreted by the JVM. Because interpretation adds runtime overhead, many JVM implementations include a just-in-time compiler that translates heavily used bytecode sequences into native code for the underlying hardware. As a result, Java translation is a combination of compilation and interpretation.

class definitions not change after compile time, then name resolution inside methods can be performed at compile time. We say that such a language has a *closed class structure*. On the other hand, if the language allows the running program to change its class structure, either by importing classes as in Java or by editing classes as in Smalltalk, then the language has an *open class structure*.

Closed class structure
If the class structure of an application is fixed at compile time, the OOL has a *closed hierarchy*.

Open class structure
If an application can change its class structure at runtime, it has an *open hierarchy*.

Given a method `m`, the compiler can map a name that appears in `m` to either a declaration in some nested scope of `m`, or to the class definition that contains `m`. If the name is declared in a superclass, the compiler's ability to determine which superclass declares the name depends on whether the class structure is open or closed. With a closed class structure, the compiler has the complete class hierarchy, so it can resolve all names back to their declarations and, with appropriate runtime structures to support naming, can generate code to access any name. With an open class structure, the compiler may not know the class structure until runtime. Such languages require runtime mechanisms to resolve names in the class hierarchy; that requirement, in turn, typically leads to implementations that rely on interpretation or runtime compilation. Similar situations can arise from explicit or implicit conversions in a language with a closed class structure; for example virtual functions in C++ may require runtime support.

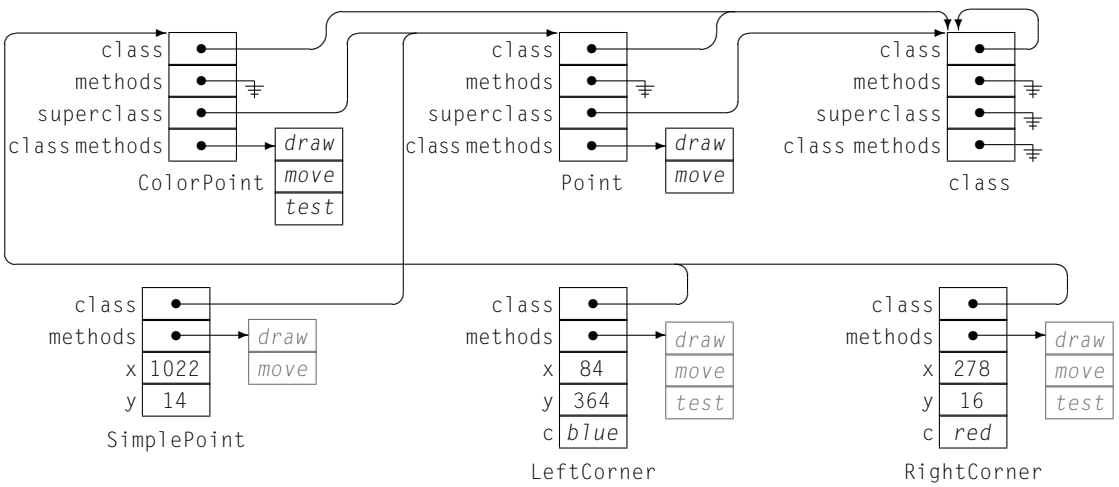
C++ has a closed class structure. Any functions, other than virtual functions, can be resolved at compile time. Virtual functions require runtime resolution.

6.3.4 Runtime Structures to Support Object-Oriented Languages

Just as Algol-like languages need runtime structures to support their lexical name spaces, so too do object-oriented languages need runtime structures to support both their lexical hierarchy and their class hierarchy. Some of those structures are identical to the ones found in an ALL. For example, the control information for methods, as well as storage for method-local names, is stored in ARs. Other structures are designed to address specific problems introduced by the OOL. For example, object lifetimes need not match the invocation of any particular method, so their persistent state cannot be stored in some AR. Thus, each object needs its own object record (OR) to hold its state. The ORs of classes instantiate the inheritance hierarchy; they play a critical role in translation and execution.

The amount of runtime support that an OOL needs depends heavily on features of the OOL. To explain the range of possibilities, we will begin with the structures that might be generated for the definitions in Figure 6.6, assuming a language with single inheritance and an open class structure. From that base case, we will explore the simplifications and optimizations that a closed class structure allows.

Figure 6.7 shows the runtime structures that might result from instantiating three objects using the definitions from Figure 6.6. `SimplePoint` instantiates `Point`, while both `LeftCorner` and `RightCorner` instantiate `ColorPoint`. Each of object has its own OR, as do the classes `Point` and



■ **FIGURE 6.7** Runtime Structures for the ColorPoint Example.

`ColorPoint`. For completeness, the diagram shows an OR for class `class`. Depending on the language, an implementation may avoid representing some of these fields, method vectors, and pointers.

The OR for a simple object, such as `LeftCorner`, contains a pointer to the class that defined `LeftCorner`, a pointer to the method vector for that class, and space for its fields, `x`, `y`, and `c`, Notice that the inherited fields in a `ColorPoint` and in its method vector have the same offset that they would in the base class `Point`. The OR for `ColorPoint` literally extends the OR for `Point`. The resulting consistency allows a superclass method such as `Point.move` to operate correctly on a subclass object, such as `LeftCorner`.

The OR for a class contains a pointer to its class, `class`, a pointer to the method vector for `class`, and its local fields which include `superclass` and `class methods`. In the figure, all method vectors are drawn as complete method vectors—that is, they include all of the methods for the class, both local and inherited. The `superclass` field records the inheritance hierarchy, which may be necessary in an open class structure. The `class methods` field points to the method vector used members of the class.

To avoid a confusing tangle of lines in the figure, we have simplified the method vectors in several ways. The drawing shows separate method vectors rather than pointers to a shared copy of the class methods vectors. The copies are drawn in gray. Class `class` has null pointers for both its `methods` and its `class methods` fields. In a real implementation, these would likely have some methods, which would, in turn, cause non-null pointers in the `methods` field of both `Point` and `ColorPoint`.

Method Invocation

How does the compiler generate code to invoke a method such as `draw`? Methods are always invoked relative to an object, say `RightCorner`, as receiver. For the invocation to be legal, `RightCorner` must be visible at the point of the call, so the compiler can discover how to find `RightCorner` with a symbol-table lookup. The compiler first looks in the method's lexical hierarchy, then in the class hierarchy, and, finally, in the global scope. That lookup provides enough information to let the compiler emit code to obtain a pointer to `RightCorner`'s OR.

Once the compiler has emitted code to obtain the OR pointer, it locates the method vector pointer at offset 4 in the OR pointer. It uses `draw`'s offset, which is 0 relative to the method vector pointer, to obtain a pointer to the desired implementation of `draw`. It uses that code pointer in a standard procedure call, with one twist—it passes `RightCorner`'s OR pointer as the implicit first parameter to `draw`. Because it located `draw` from `RightCorner`'s OR,

which contains a pointer to `ColorPoint`'s `class methods` vector, the code sequence locates the proper implementation of `draw`. If the invocation had been `SimplePoint.draw`, the same process would have found `Point`'s method vector pointer and called `Point.draw`.

The example assumes that each class has a complete method vector. Thus, the slot for `move` in `ColorPoint`'s method vector points to `Point.move` while the slot for `draw` points to `ColorPoint.draw`. This scheme produces the desired result—an object of class `x` invokes the implementation of a method that is visible inside the definition for class `x`. The alternative scheme would represent only `ColorPoint`'s locally defined methods in its `class method` vector, and would locate an inherited method by chasing ORs up the superclass chain in a manner analogous to access links for lexical scoping and ARs.

Object-Record Layout

"Implementation" might be a compiler, an interpreter, or a JIT. The layout problem is the same.

One subtle point in the example is that an implementation must maintain consistent offsets, by name, up and down the superclass hierarchy. Fields, such as `x` and `y`, must appear at the same offset in an OR of class `Point` and `ColorPoint` for a method such as `move` to operate correctly on ORs of either its class or its superclasses. For the same reason, methods must appear at the same offsets in the method vectors of related classes.

Without inheritance, the compiler can assign offsets in arbitrary order to the class' fields and methods. It compiles those offsets directly into the code. The code uses the receiver's pointer (e.g. `this`) and the offsets to locate any desired field in the OR or any method in the method vector.

With single inheritance, OR layout is straightforward. Since each class has only one direct superclass, the compiler appends the new fields to the end of the superclass OR layout, extending the OR layout. This approach, called *prefixing*, ensures consistent offsets up and down the superclass hierarchy. When an object is cast to one of its superclasses, the fields in the OR are in their expected locations. The ORs in Figure 6.7 follow this scheme.

In a language with a closed class structure, object-record layout can be done at compile time, as soon as all the superclasses are known. In a language with an open class structure, object-record layout must be done between the time when the superclass structure is known and the time when ORs are allocated. If the class structure is unknown at compile time but cannot change at runtime, these issues can be resolved at linktime or at the start of execution. If the class structure can change at runtime, as in either Java or Smalltalk, then the runtime environment must be prepared to adjust object layouts and the class hierarchy.

- If classes change infrequently, the overhead for adjusting object-record layouts can be small. The runtime environment, either an interpreter or a JIT and an interpreter, can compute object record layouts and build method vectors for each affected class when the class structure changes.
- If classes change often, the compiler must still compute object-record layouts and adjust them. However, it may be more efficient for the implementation to use incomplete method vectors and search rather than rebuilding class method vectors at each change. (See the next subsection.)

In Java, for example, classes only change when the class loader runs. Thus, the class loader could trigger the rebuilding process.

As a final issue, consider what happens if the language allows changes to the structure of a class that has instantiated objects. Adding a field or a method to a class with instantiated objects necessitates visiting those objects, building them new ORs, and connecting those ORs back into the runtime environment in a seamless way. (Typically, the latter requirement requires an extra level of indirection on references to ORs.) To avoid these complications, most languages forbid changes to classes that already have instantiated objects.

Static versus Dynamic Dispatch

Dispatch
The process of calling a method is often called *dispatch*, a term derived from the message-passing model of OOLs such as Smalltalk.

The runtime structures shown in Figure 6.7 suggest that every method call requires one or more load operations to locate the method's implementation. In a language with a closed class structure, the compiler can avoid this overhead for most calls. In C++, for example, the compiler can resolve any method to a concrete implementation at compile time, unless the method is declared as a *virtual method*—meaning, essentially, that the programmer wants to locate the implementation relative to the receiver's class.

With a virtual method, dispatch is done through the appropriate method vector. The compiler emits code to locate the method's implementation at runtime using the object's `method` vector, a process called *dynamic dispatch*. If, however, the C++ compiler can prove that some virtual method call has a known invariant receiver class, it can generate a direct call, sometimes called *static dispatch*.

Languages with open class structures may need to rely on dynamic dispatch. If the class structure can change at runtime, the compiler cannot resolve method names to implementations; instead, it must defer this process to runtime. The techniques used to address this problem range from recomputing method vectors at each change in the class hierarchy to runtime name resolution and search in the class hierarchy.

- If the class hierarchy changes infrequently, the implementation may simply rebuild method vectors for the affected classes after each change. In this scheme, the runtime system must traverse the superclass

METHOD CACHES

To support an open class hierarchy, the compiler may need to produce a search key for each method name and retain a mapping of keys to implementations that it can search at runtime. The map from method name to search key can be simple—using the method name or a hash index for that name—or it can be complex—assigning each method name an integer from a compact set using some link-time mechanism. In either case, the compiler must include tables that can be searched at runtime to locate the implementation of a method in the most recent ancestor of the receiver's class.

To improve method lookup in this environment, the runtime system can implement a *method cache*—a software analog of the hardware data cache found in most processors. The method cache has a small number of entries, say 1000. Each cache entry consists of a key, a class, and a pointer to a method implementation. A dynamic dispatch begins with a lookup in the method cache; if it finds an entry with the receiver's class and method key, it returns the cached method pointer. If the lookup fails, the dispatch performs a complete search up the superclass chain, starting with the receiver's class. It caches the result that it finds and returns the method pointer.

Of course, creating a new entry may force eviction of some other cache entry. Standard cache replacement policies, such as least recently used or round robin, can select the method to evict. Larger caches retain more information, but require more memory and may take longer to search. When the class structure changes, the implementation can clear the method cache to prevent incorrect results on future lookups.

To capture type locality at individual calls, some implementations use an *inline method cache*, a single entry cache located at the actual call site. The cache stores the receiver's class and the method pointer from the last invocation at that site. If the current receiver class matches the previous receiver class, the call uses the cached method pointer. A change to the class hierarchy must invalidate the cache, either by changing the class' tag or by overwriting the class tags at each inline cache. If the current class does not match the cached class, a full lookup is used, and that lookup writes its results into the inline cache.

hierarchy to locate method implementations and build subclass method vectors.

- If the class hierarchy changes often, the implementor may choose to keep incomplete method vectors in each class—record just the local methods. In this scheme, a call to a superclass method triggers a runtime search in the class hierarchy for the first method of that name.

Either of these schemes requires that the language runtime retain lookup tables of method names—either source level names or search keys derived from those names. Each class needs a small dictionary in its OR. Runtime name resolution looks up the method name in the dictionaries through the hierarchy, in a manner analogous to the chain of symbol tables described in Section 5.5.3.

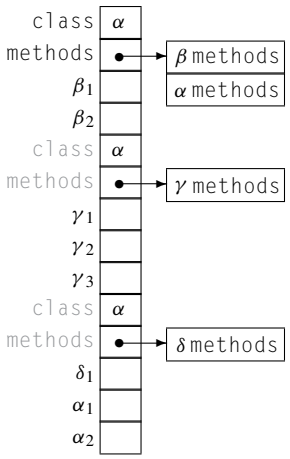
OOL implementations try to reduce the cost of dynamic dispatch through one of two general strategies. They can perform analysis to prove that a given method invocation always uses a receiver of the same known class, in which case they can replace dynamic dispatch with static dispatch. For calls where they cannot discover the receiver's class, or where the class varies at runtime, the implementations can cache search results to improve performance. In this scheme, the search consults a method cache before it searches the class hierarchy. If the method cache contains the mapping for the receiver's class and the method name, the call uses the cached method pointer and avoids the search.

Multiple Inheritance

Some OOLs allow multiple inheritance, meaning a new class may inherit from several superclasses that have inconsistent object layouts. This situation poses a new problem: the compiled code for a superclass method uses offsets based on the OR layout for that superclass. Of course, different immediate superclasses may assign conflicting offsets to their fields. To reconcile these competing offsets, the compiler must adopt a slightly more complex scheme: it must use different OR pointers with methods from different superclasses.

Consider a class α that inherits from multiple superclasses, β, γ, and δ. To lay out the OR for an object of class α, the implementation must first impose an order on α's superclasses—say β, γ, δ. It lays out the OR for class α as the entire OR, including class pointer and method vector, for β, followed by the entire OR for γ, followed by the entire OR for δ. To this layout, it appends any fields declared locally in the declaration of α. It constructs the method vector for α by appending α's methods to the method vector for the first superclass.

The drawing in the margin shows the resulting OR layout for class α, We assume that α defines two local fields, α_1 and α_2, and that the fields of β, γ, and δ are named similarly. The OR for α divides into four logical sections: the OR for β, the OR for γ, the OR for δ, and the space for fields declared in α. Methods declared in α are appended to the method vector for the first section. The "shadow" class pointers and method vectors, whose



Object Record for α

labels appear in gray, exist to allow those superclass methods to receive the environment that they expect—the OR layout of the corresponding superclass.

The remaining complication involved in multiple inheritance lies in the fact that the OR pointer must be adjusted when a superclass method is invoked using one of the shadow class pointers and method vectors. The invocation must adjust the pointer by the distance between the class pointer at the top of the OR and the shadow class pointer. The simplest mechanism to accomplish this adjustment is to insert a *trampoline function* between the method vector and the actual method. The trampoline adjusts the OR pointer, invokes the method with all of the parameters, and readjusts the OR pointer on return.

SECTION REVIEW

Algol-like languages typically use lexical scoping, in which names spaces are properly nested and new instances of a name obscure older ones. To hold data associated with its local scope, a procedure has an activation record for each invocation. In contrast, while object-oriented languages may use lexical scopes for procedure-local names, they also rely on a hierarchy of scopes defined by the data—by the hierarchy of class definitions. This dual-hierarchy name space leads to more complex interactions among names and to more complex implementations.

Both styles of naming require runtime structures that both reflect and implement the naming hierarchy. In an ALL, the activation records can capture the structure of the name space, provide the necessary storage for most values, and preserve the state necessary for correct execution. In an OOL, the activation records of running code still capture the lexically scoped portion of the name space and the state of execution; however, the implementation also needs a hierarchy of object records and class records to model the object-based portion of the name space.

Review Questions

1. In C, `setjmp` and `longjmp` provide a mechanism for interprocedural transfer of control. `setjmp` creates a data structure; invoking `longjmp` on the data structure created by a `setjmp` causes execution to continue immediately after the `setjmp`, with the context present when the `setjmp` executed. What information must `setjmp` preserve? How does the implementation of `setjmp` change between stack-allocated and heap-allocated ARs?

2. Consider the example from Figure 6.7. If the compiler encounters a reference to `LeftCorner` with a cast to class `Point`, which implementation of the method `draw` would that cast reference invoke? How could the programmer refer to the other implementation of `draw`?

6.4 COMMUNICATING VALUES BETWEEN PROCEDURES

The central notion underlying the concept of a procedure is abstraction. The programmer abstracts common operations relative to a small set of names, or formal parameters, and encapsulates those operations in a procedure. To use the procedure, the programmer invokes it with an appropriate *binding* of values, or actual parameters, to those formal parameters. The callee executes, using the formal parameter names to access the values passed as actual parameters. If the programmer desires, the procedure can return a result.

6.4.1 Passing Parameters

Parameter binding maps the actual parameters at a call site to the callee's formal parameters. It lets the programmer write a procedure without knowledge of the contexts in which it will be called. It lets the programmer invoke the procedure from many distinct contexts without exposing details of the procedure's internal operation in each caller. Thus, parameter binding plays a critical role in our ability to write abstract, modular code.

Most modern programming languages use one of two conventions for mapping actual parameters to formal parameters: *call-by-value* binding and *call-by-reference* binding. These techniques differ in their behavior. The distinction between them may be best explained by understanding their implementations.

Call by Value

Consider the following procedure, written in C, and several call sites that invoke it:

```
int fee(int x, int y) {
  x = 2 * x;
  y = x + y;
  return y;
}
```

```
c = fee(2,3);
a = 2;
b = 3;
c = fee(a,b);
a = 2;
b = 3;
c = fee(a,a);
```

Call by value
a convention where the caller evaluates the actual parameters and passes their values to the callee

Any modification of a value parameter in the callee is not visible in the caller.

CALL-BY-NAME PARAMETER BINDING

Algol introduced another parameter-binding mechanism, *call by name*. In call-by-name binding, a reference to a formal parameter behaves exactly as if the actual parameter had been textually substituted in its place, with appropriate renaming. This simple rule can lead to complex behavior. Consider the following artificial example in Algol 60:

```
begin comment Simple array example;
  procedure zero(Arr,i,j,u1,u2);
    integer Arr;
    integer i,j,u1,u2;
    begin;
      for i := 1 step 1 until u1 do
        for j := 1 step 1 until u2 do
          Arr := 0;
      end;
  integer array Work[1:100,1:200];
  integer p, q, x, y, z;
  x := 100;
  y := 200;
  zero(Work[p,q],p,q,x,y);
end
```

The call to `zero` assigns zero to every element of the array `Work`. To see this, rewrite `zero` with the text of the actual parameters.

While call-by-name binding was easy to define, it was difficult to implement and to understand. In general, the compiler must produce, for each formal parameter, a function that evaluates the actual parameter to return a pointer. These functions are called *thunks*. Generating competent thunks was complex; evaluating a thunk for each parameter access was expensive. In the end, these disadvantages overcame any advantages that call-by-name parameter binding offered.

The R programming language, a domain-specific tool for statistical analysis, implements a lazy form of call-by-value binding. The implementation creates and passes thunks that are invoked the first time that the parameter value is actually referenced. The thunk, or *promise*, stores its result for subsequent references.

With *call-by-value* parameter passing, as in C, the caller copies the value of an actual parameter into the appropriate location for the corresponding formal parameter—either a register or a parameter slot in the callee's AR. Only one name refers to that value—the name of the formal parameter. Its value is an initial condition, determined by evaluating the actual parameter

at the time of the call. If the callee changes its value, that change is visible inside the callee, but not in the caller.

The three invocations produce the following results when invoked using call-by-value parameter binding:

Call by Value	a		b		Return Value
	in	out	in	out	
fee(2,3)	-	-	-	-	7
fee(a,b)	2	2	3	3	7
fee(a,a)	2	2	3	3	6

With call by value, the binding is simple and intuitive.

One variation on call-by-value binding is *call-by-value-result* binding. In the value-result scheme, the values of formal parameters are copied back into the corresponding actual parameters as part of the process of returning control from the callee to the caller. The programming language Ada includes value-result parameters. The value-result mechanism also satisfies the rules of the FORTRAN 77 language definition.

Call by Reference

Call by reference
a convention where the compiler passes an address for the formal parameter to the callee

If the actual parameter is a variable (rather than an expression), then changing the formal's value also changes the actual's value.

With *call-by-reference* parameter passing, the caller stores a pointer in the AR slot for each parameter. If the actual parameter is a variable, it stores the variable's address in memory. If the actual parameter is an expression, the caller evaluates the expression, stores the result in the local data area of its own AR, and then stores a pointer to that result in the appropriate parameter slot in the callee's AR. Constants should be treated as expressions to avoid any possibility of the callee changing the value of a constant. Some languages forbid passing expressions as actual parameters to call-by-reference formal parameters.

Inside the callee, each reference to a call-by-reference formal parameter needs an extra level of indirection. Call by reference differs from call by value in two critical ways. First, any redefinition of a reference formal parameter is reflected in the corresponding actual parameter. Second, any reference formal parameter might be bound to a variable that is accessible by another name inside the callee. When this happens, we say that the names are *aliases*, since they refer to the same storage location. Aliasing can create counterintuitive behavior.

Consider the earlier example, rewritten in PL/I, which uses call-by-reference parameter binding.

```
fee: procedure (x,y)                    c = fee(2,3);
       returns fixed binary;            a = 2;
     declare x, y fixed binary;         b = 3;
     x = 2 * x;                         c = fee(a,b);
     y = x + y;                         a = 2;
     return y;                          b = 3;
     end fee;                           c = fee(a,a);
```

With call-by-reference parameter binding, the example produces different results. The first call is straightforward. The second call redefines both `a` and `b`; those changes would be visible in the caller. The third call causes `x` and `y` to refer to the same location, and thus, the same value. This *alias* changes `fee`'s behavior. The first assignment gives `a` the value 4. The second assignment then gives `a` the value 8, and `fee` returns 8, where `fee(2,2)` would return 6.

Alias
When two names can refer to the same location, they are said to be *aliases*.

In the example, the third call creates an alias between `x` and `y` inside `fee`.

Call by	a		b		Return
Reference	in	out	in	out	Value
fee(2,3)	-	-	-	-	7
fee(a,b)	2	4	3	7	7
fee(a,a)	2	8	3	3	8

Space for Parameters

The size of the representation for a parameter has an impact on the cost of procedure calls. Scalar values, such as variables and pointers, are stored in registers or in the parameter area of the callee's AR. With call-by-value parameters, the actual value is stored; with call-by-reference parameters, the address of the parameter is stored. In either case, the cost per parameter is small.

Large values, such as arrays, records, or structures, pose a problem for call by value. If the language requires that large values be copied, the overhead of copying them into the callee's parameter area will add significant cost to the procedure call. (In this case, the programmer may want to model call by reference and pass a pointer to the object rather than the object.) Some languages allow the implementation to pass such objects by reference. Others include provisions that let the programmer specify that passing a particular

parameter by reference is acceptable; for example, the `const` attribute in C assures the compiler that a parameter with the attribute is not modified.

6.4.2 Returning Values

To return a value from a function the compiler must set aside space for the returned value. Because the return value, by definition, is used after the callee terminates, it needs storage outside the callee's AR. If the compiler writer can ensure that the return value is of small fixed size, then it can store the value either in the caller's AR or in a designated register.

With call-by-value parameters, linkage conventions often designate the register reserved for the first parameter as the register to hold the return value.

All of our pictures of the AR have included a slot for a returned value. To use this slot, the caller allocates space for the returned value in its own AR, and stores a pointer to that space in the return slot of its own AR. The callee can load the pointer from the caller's return-value slot (using the copy of the caller's ARP that it has in the callee's AR). It can use the pointer to access the storage set aside in the caller's AR for the returned value. As long as both caller and callee agree about the size of the returned value, this works.

If the caller cannot know the size of the returned value, the callee may need to allocate space for it, presumably on the heap. In this case, the callee allocates the space, stores the returned value there, and stores the pointer in the return-value slot of the caller's AR. On return, the caller can access the return value using the pointer that it finds in its return-value slot. The caller must free the space allocated by the callee.

If the return value is small—the size of the return-value slot or less—then the compiler can eliminate the indirection. For a small return value, the callee can store the value directly into the return value slot of the caller's AR. The caller can then use the value directly from its AR. This improvement requires, of course, that the compiler handle the value in the same way in both the caller and the callee. Fortunately, type signatures for procedures can ensure that both compiles have the requisite information.

6.4.3 Establishing Addressability

As part of the linkage convention, the compiler must ensure that each procedure can generate an address for each variable that it needs to reference. In an ALL, a procedure can refer to global variables, local variables, and any variable declared in a surrounding lexical scope. In general, the address calculation consists of two portions: finding the *base address* of the appropriate *data area* for the scope that contains the value, and finding the correct offset within that data area. The problem of finding base addresses divides into two

Data area
The region in memory that holds the data for a specific scope is called its *data area*.

Base address
The address of the start of a data area is often called a *base address*.

cases: data areas with static base addresses and those whose address cannot be known until runtime.

Variables with Static Base Addresses

Compilers typically arrange for global data areas and static data areas to have static base addresses. The strategy to generate an address for such a variable is simple: compute the data area's base address into a register and add its offset to the base address. The compiler's IR will typically include address modes to represent this calculation; for example, in ILOC, `loadAI` represents a "register + immediate offset" mode and `loadAO` represents a "register + register" mode.

To generate the runtime address of a static base address, the compiler attaches a symbolic, assembly-level label to the data area. Depending on the target machine's instruction set, that label might be used in a load immediate operation or it might be used to initialize a known location, in which case it can be moved into a register with a standard load operation.

Name mangling
The process of constructing a unique string from a source-language name is called *name mangling*.

If `&fee.` is too long for an immediate load, the compiler may need to use multiple operations to load the address.

The compiler constructs the label for a base address by *mangling* the name. Typically, it adds a prefix, a suffix, or both to the original name, using characters that are legal in the assembly code but not in the source language. For example, mangling the global variable name `fee` might produce the label `&fee.`; the label is then attached to an assembly-language pseudo-operation that reserves space for `fee`. To move the address into a register, the compiler might emit an operation such as `loadI &fee.` $\Rightarrow r_i$. Subsequent operations can then use r_i to access the memory location for `fee`. The label becomes a relocatable symbol for the assembler and the loader, which convert it into a runtime virtual address.

Global variables may be labelled individually or in larger groups. In FORTRAN, for example, the language collects global variables into common blocks. A typical FORTRAN compiler establishes one label for each common block. It assigns an offset to each variable in each common block and generates `load` and `store` operations relative to the common block's label. If the data area is larger than the offset allowed in a "register + offset" operation, it may be advantageous to have multiple labels for parts of the data area.

Similarly, the compiler may combine all the static variables in a single scope into one data area. This reduces the likelihood of an unexpected naming conflict; such conflicts are discovered during linking or loading and can be confusing to the programmer. To avoid such conflicts, the compiler can base the label on a globally visible name associated with the scope. This strategy decreases the number of base addresses in use at any time, reducing demand

for registers. Using too many registers to hold base addresses may adversely affect overall runtime performance.

Variables with Dynamic Base Addresses

As described in Section 6.3.2, local variables declared within a procedure are typically stored in the procedure's AR. Thus, they have dynamic base addresses. To access these values, the compiler needs a mechanism to find the addresses of various ARs. Fortunately, lexical scoping rules limit the set of ARs that can be accessed from any point in the code to the current AR and the ARs of lexically enclosing procedures.

Local Variable of the Current Procedure

Accessing a local variable of the current procedure is trivial. Its base address is simply the address of the current AR, which is stored in the ARP. Thus, the compiler can emit code that adds its offset to the ARP and uses the result as the value's address. (This offset is the same value as the offset in the value's static coordinate.) In ILOC the compiler might use a `loadAI` (an "address + immediate offset" operation) or a `loadAO` (an "address + offset" operation). Most processors provide efficient support for these common operations.

In some cases, a value is not stored at a constant offset from the ARP. The value might reside in a register, in which case loads and stores are not needed. If the variable has an unpredictable or changing size, the compiler will store it in an area reserved for variable-size objects, either at the end of the AR or in the heap. In this case, the compiler can reserve space in the AR for a pointer to the variable's actual location and generate one additional load to access the variable.

Local Variables of Other Procedures

To access a local variable of some enclosing lexical scope, the compiler must arrange for the construction of runtime data structures that map a static coordinate, produced using a lexically-scoped symbol table in the parser, into a runtime address.

For example, assume that procedure `fee`, at lexical level m, references variable `a` from `fee`'s lexical ancestor `fie`, at level n. The parser converts this reference into a static coordinate $\langle n,o \rangle$, where o is `a`'s offset in the AR for `fie`. The compiler can compute the number of lexical levels between `fee` and `fie` as $m-n$. (The coordinate $\langle m–n,o \rangle$ is sometimes called the *static-distance coordinate* of the reference.)

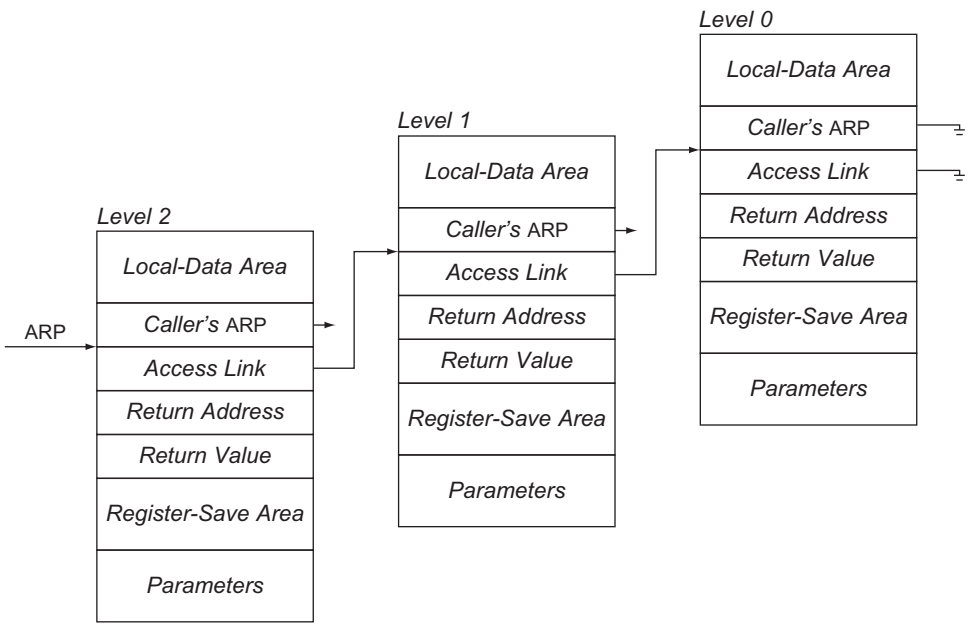
The compiler needs a mechanism to convert $\langle n,o \rangle$ into a runtime address. In general, that scheme will use runtime data structures to find the ARP of

the most recent level n procedure, and use that ARP as the base address in its computation. It adds the offset o to that base address to produce a run-time address for the value whose static coordinate is $\langle n,o \rangle$. The complication lies in building and traversing the runtime data structures to find the base address. The following subsections examine two common methods: use of access links and use of a global display.

Access Links

The intuition behind access links is simple. The compiler ensures that each AR contains a pointer, called an *access link* or a *static link*, to the AR of its immediate lexical ancestor. The access links form a chain that includes all the lexical ancestors of the current procedure, as shown in Figure 6.8. Thus, any local variable of another procedure that is visible to the current procedure is stored in an AR on the chain of access links that begins in the current procedure.

To access a value $\langle n,o \rangle$ from a level m procedure, the compiler emits code to walk the chain of links and find the level n ARP. Next, it emits a load that uses the level n ARP and o. To make this concrete, consider the program represented by Figure 6.8. Assume that m is 2 and that the access link is stored at an offset of -4 from the ARP. The following table shows a set of



■ **FIGURE 6.8** Using Access Links.

three different static coordinates alongside the ILOC code that a compiler might generate for them. Each sequence leaves the result in r_2.

Coordinate	Code
⟨2,24⟩	`loadAI` r_{arp}, 24 ⇒ r_2
⟨1,12⟩	`loadAI` r_{arp}, -4 ⇒ r_1 `loadAI` r_1, 12 ⇒ r_2
⟨0,16⟩	`loadAI` r_{arp}, -4 ⇒ r_1 `loadAI` r_1, -4 ⇒ r_1 `loadAI` r_1, 16 ⇒ r_2

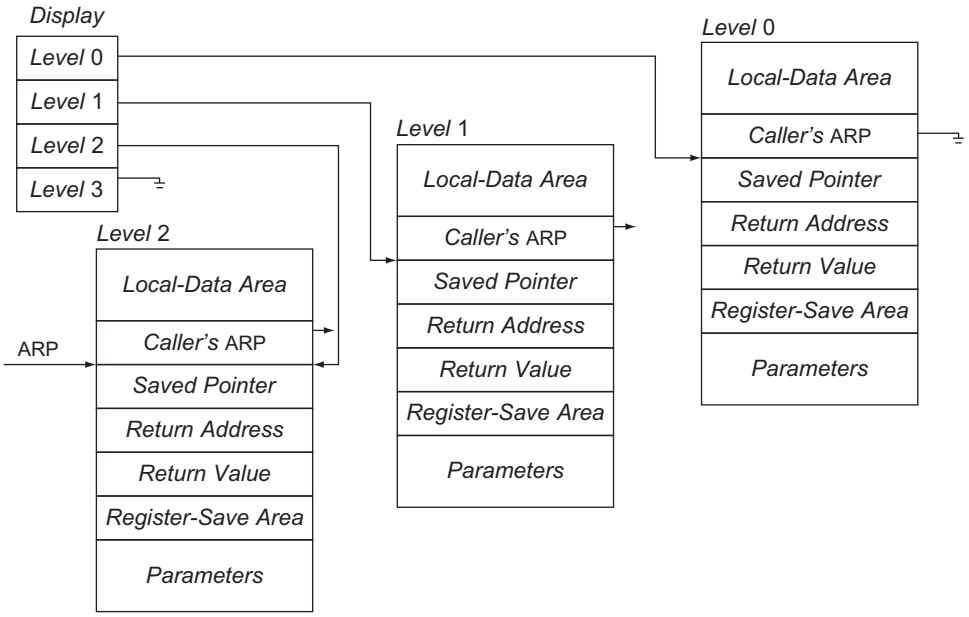
Since the compiler has the static coordinate for each reference, it can compute the static distance ($m-n$). The distance tells it how many chain-following loads to generate, so the compiler can emit the correct sequence for each nonlocal reference. The cost of the address calculation is proportional to the static distance. If programs exhibit shallow lexical nesting, the difference in cost between accessing two variables at different levels will be fairly small.

To maintain access links, the compiler must add code to each procedure call that finds the appropriate ARP and stores it as the callee's access link. For a caller at level m and a callee at level n, three cases arise. If $n = m + 1$, the callee is nested inside the caller, and the callee can use the caller's ARP as its access link. If $n = m$, the callee's access link is the same as the caller's access link. Finally, if $n < m$, the callee's access link is the level $n - 1$ access link for the caller. (If n is zero, the access link is null.) The compiler can generate a sequence of $m - n + 1$ loads to find this ARP and store that pointer as the callee's access link.

Global Display

In this scheme, the compiler allocates a single global array, called the *display*, to hold the ARP of the most recent activation of a procedure at each lexical level. All references to local variables of other procedures become indirect references through the display. To access a variable ⟨*n*,*o*⟩, the compiler uses the ARP from element n of the display. It uses o as the offset and generates the appropriate load operation. Figure 6.9 shows this situation.

Returning to the static coordinates used in the discussion of access links, the following table shows code that the compiler might emit for a display-based



■ **FIGURE 6.9** Using a Global Display.

implementation. Assume that the current procedure is at lexical level 2, and that the label `_disp` gives the address of the display.

Coordinate	Code
⟨2,24⟩	`loadAI` r_{arp}`,24` ⇒ r_2
⟨1,12⟩	`loadI _disp` ⇒ r_1 `loadAI` r_1`,4` ⇒ r_1 `loadAI` r_1`,12` ⇒ r_2
⟨0,16⟩	`loadI _disp` ⇒ r_1 `loadAI` r_1`,16` ⇒ r_2

With a display, the cost of nonlocal access is fixed. With access links, the compiler generates a series of $m-n$ loads; with a display, it uses $n \times l$ as offset into the display, where l is the length of a pointer (4 in the example). Local access is still cheaper than nonlocal access, but with a display, the penalty for nonlocal access is constant, rather than variable.

Of course, the compiler must insert code where needed to maintain the display. Thus, when procedure p at level n calls some procedure q at level $n+1$, p's ARP becomes the display entry for level n. (While p is executing, that

entry is unused.) The simplest way to keep the display current is to have *p* update the level *n* entry when control enters *p* and to restore it on exit from *p*. On entry, *p* can copy the level *n* display entry to the reserved addressability slot in its AR and store its own ARP in the level *n* slot of the display.

Many of these display updates can be avoided. The only procedures that can use the ARP stored by a procedure *p* are procedures *q* that *p* calls (directly or indirectly), where *q* is nested inside *p*'s scope. Thus, any *p* that does not call a procedure nested inside itself need not update the display. This eliminates all updates in leaf procedures, as well as many other updates.

SECTION REVIEW

If the fundamental purpose of a procedure is abstraction, then the ability to communicate values between procedures is critical to their utility. The flow of values between procedures occurs with two different mechanisms: the use of parameters and the use of values that are visible in multiple procedures. In each of these cases, the compiler writer must arrange access conventions and runtime structures to support the access. For parameter binding, two particular mechanisms have emerged as the common cases: call by value and call by reference. For nonlocal accesses, the compiler must emit code to compute the appropriate base addresses. Two mechanisms have emerged as the common cases: access links and a display.

The most confusing aspect of this material is the distinction between actions that happen at compile time, such as the parser finding static coordinates for a variable, and those that happen at runtime, such as the executing program tracing up a chain of access links to find the ARP of some surrounding scope. In the case of compile-time actions, the compiler performs the action directly. In the case of runtime actions, the compiler emits code that will perform the action at runtime.

Review Questions

1. An early FORTRAN implementation had an odd bug. The short program in the margin would print, as its result, the value 16. What did the compiler do that led to this result? What should it have done instead? (FORTRAN uses call-by-reference parameter binding.)
2. Compare and contrast the costs involved in using access links versus global displays to establish addresses for references to variables declared in surrounding scopes. Which would you choose? Do language features affect your choice?

```
subroutine change(n)
  integer n
  n = n * 2
end

program test
  call change(2)
  print *, 2 * 2
end
```

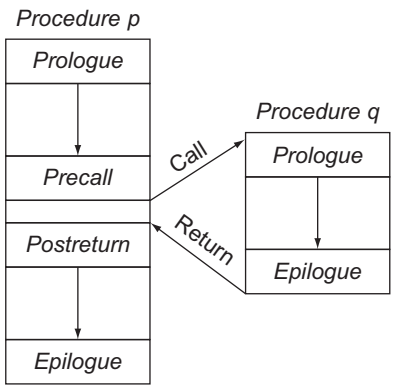
6.5 STANDARDIZED LINKAGES

The procedure linkage is a contract between the compiler, the operating system, and the target machine that clearly divides responsibility for naming, allocation of resources, addressability, and protection. The procedure linkage ensures interoperability of procedures between the user's code, as translated by the compiler, and code from other sources, including system libraries, application libraries, and code written in other programming languages. Typically, all of the compilers for a given combination of target machine and operating system use the same linkage, to the extent possible.

The linkage convention isolates each procedure from the different environments found at call sites that invoke it. Assume that procedure p has an integer parameter x. Different calls to p might bind x to a local variable stored in the caller's stack frame, to a global variable, to an element of some static array, and to the result of evaluating an integer expression such as $y + 2$. Because the linkage convention specifies how to evaluate the actual parameter and store its value, as well as how to access x in the callee, the compiler can generate code for the callee that ignores the differences between the runtime environments at the different calls sites. As long as all the procedures obey the linkage convention, the details will mesh to create the seamless transfer of values promised by the source-language specification.

The linkage convention is, of necessity, machine dependent. For example, it depends implicitly on information such as the number of registers available on the target machine and the mechanisms for executing a call and a return.

Figure 6.10 shows how the pieces of a standard procedure linkage fit together. Each procedure has a *prologue sequence* and an *epilogue sequence*. Each call site includes both a *precall sequence* and a *postreturn sequence*.



■ **FIGURE 6.10** A Standard Procedure Linkage.

- *Precall Sequence* The precall sequence begins the process of constructing the callee's environment. It evaluates the actual parameters, determines the return address, and, if necessary, the address of space reserved to hold a return value. If a call-by-reference parameter is currently allocated to a register, the precall sequence needs to store it into the caller's AR so that it can pass that location's address to the callee.

  Many of the values shown in the diagrams of the AR can be passed to the callee in registers. The return address, an address for the return value, and the caller's ARP are obvious candidates. The first k actual parameters can be passed in registers as well—a typical value for k might be 4. If the call has more than k parameters, the remaining actual parameters must be stored in either the callee's AR or the caller's AR.

- *Postreturn Sequence* The postreturn sequence undoes the actions of the precall sequence. It must restore any call-by-reference and call-by-value-result parameters that need to be returned to registers. It restores any caller-saved registers from the register save area. It may need to deallocate all or part of the callee's AR.

- *Prologue Sequence* The prologue for a procedure completes the task of creating the callee's runtime environment. It may create space in the callee's AR to store some of the values passed by the caller in registers. It must create space for local variables and initialize them, as necessary. If the callee references a procedure-specific static data area, it may need to load the label for that data area into a register.

- *Epilogue Sequence* The epilogue for a procedure begins the process of dismantling the callee's environment and reconstructing the caller's environment. It may participate in deallocating the callee's AR. If the procedure returns a value, the epilogue may be responsible for storing the value into the address specified by the caller. (Alternatively, the code generated for a return statement may perform this task.) Finally, it restores the caller's ARP and jumps to the return address.

This framework provides general guidance for building a linkage convention. Many of the tasks can be shifted between caller and callee. In general, moving work into the prologue and epilogue code produces more compact code. The precall and postreturn sequences are generated for each call, while the prologue and epilogue occur once per procedure. If procedures are called, on average, more than once, then there are fewer prologue and epilogue sequences than precall and postreturn sequences.

MORE ABOUT TIME

In a typical system, the linkage convention is negotiated between the compiler implementors and the operating-system implementors at an early stage of the system's development. Thus, issues such as the distinction between caller-saves and callee-saves registers are decided at design time. When the compiler runs, it must emit procedure prologue and epilogue sequences for each procedure, along with precall and postreturn sequences for each call site. This code executes at runtime. Thus, the compiler cannot know the return address that it should store into a callee's AR. (Neither can it know, in general, the address of that AR.) It can, however, include a mechanism that will generate the return address at link time (using a relocatable assembly language label) or at runtime (using some offset from the program counter) and store it into the appropriate location in the callee's AR.

Similarly, in a system that uses a display to provide addressability for local variables of other procedures, the compiler cannot know the runtime addresses of the display or the AR. Nonetheless, it emits code to maintain the display. The mechanism that achieves this requires two pieces of information: the lexical nesting level of the current procedure and the address of the global display. The former is known at compile time; the latter can be determined at link time by using a relocatable assembly language label. Thus, the prologue can simply load the current display entry for the procedure's level (using a `loadAO` from the display address) and store it into the AR (using a `storeAO` relative to the ARP). Finally, it stores the address of the new AR into the display slot for the procedure's lexical level.

Saving Registers

Caller-saves registers
The registers designated for the caller to save are *caller-saves registers*.

Callee-saves registers
The registers designated for the callee to save are *callee-saves registers*.

At some point in the call sequence, any register values that the caller expects to survive across the call must be saved into memory.Either the caller or the callee can perform the actual save; there is an advantage to either choice. If the caller saves registers, it can avoid saving values that it knows are not useful across the call; that knowledge might allow it to preserve fewer values. Similarly, if the callee saves registers, it can avoid saving values of registers that it does not use; again, that knowledge might result in fewer saved values.

In general, the compiler can use its knowledge of the procedure being compiled to optimize register save behavior. For any specific division of labor between caller and callee, we can construct programs for which it works well and programs for which it does not. Most modern systems take a middle ground and designate a portion of the register set for caller-saves treatment and a portion for callee-saves treatment. In practice, this seems to work well. It encourages the compiler to put long-lived values in callee-saves registers, where they will be stored only if the callee actually needs the register. It

encourages the compiler to put short-lived values in caller-saves registers, where it may avoid saving them at a call.

Allocating the Activation Record

In the most general case, both the caller and the callee need access to the callee's AR. Unfortunately, the caller cannot know, in general, how large the callee's AR must be (unless the compiler and linker can contrive to have the linker paste the appropriate values into each call site).

With stack-allocated ARs, a middle ground is possible. Since allocation consists of incrementing the stack-top pointer, the caller can begin the creation of the callee's AR by bumping the stack top and storing values into the appropriate places. When control passes to the callee, it can extend the partially built AR by incrementing the stack top to create space for local data. The postreturn sequence can then reset the stack-top pointer, performing the entire deallocation in a single step.

With heap-allocated ARs, it may not be possible to extend the callee's AR incrementally. In this situation, the compiler writer has two choices.

1. The compiler can pass the values that it must store in the callee's AR in registers; the prologue sequence can then allocate an appropriately sized AR and store the passed values in it. In this scheme, the compiler writer reduces the number of values that the caller passes to the callee by arranging to store the parameter values in the caller's AR. Access to those parameters uses the copy of the caller's ARP that is stored in the callee's AR.
2. The compiler writer can split the AR into multiple distinct pieces, one to hold the parameter and control information generated by the caller and the others to hold space needed by the callee but unknown to the caller. The caller cannot, in general, know how large to make the local data area. The compiler can store this number for each callee using mangled labels; the caller can then load the value and use it. Alternatively, the callee can allocate its own local data area and keep its base address in a register or in a slot in the AR created by the caller.

Heap-allocated ARs add to the overhead cost of a procedure call. Care in the implementation of the calling sequence and the allocator can reduce those costs.

Managing Displays and Access Links

Either mechanism for managing nonlocal access requires some work in the calling sequence. Using a display, the prologue sequence updates the display record for its own level and the epilogue sequence restores it. If the

procedure never calls a more deeply nested procedure, it can skip this step. Using access links, the precall sequence must locate the appropriate first access link for the callee. The amount of work varies with the difference in lexical level between caller and callee. As long as the callee is known at compile time, either scheme is reasonably efficient. If the callee is unknown (if it is, for example, a function-valued parameter), the compiler may need to emit special-case code to perform the appropriate steps.

SECTION REVIEW

The procedure linkage ties together procedures. The linkage convention is a social contract between the compiler, the operating system, and the underlying hardware. It governs the transfer of control between procedures, the preservation of the caller's state and the creation of the callee's state, and the rules for passing values between them.

Standard procedure linkages allow us to assemble executable programs from procedures that have different authors, that are translated at different times, and that are compiled with different compilers. Procedure linkages allow each procedure to operate safely and correctly. The same conventions allow application code to invoke system and library calls. While the details of the linkage convention vary from system to system, the basic concepts are similar across most combinations of target machine, operating system, and compiler.

Review Questions

1. What role does the linkage convention play in the construction of large programs? Of interlanguage programs? What facts would the compiler need to know in order to generate code for an interlanguage call?
2. If the compiler knows, at a procedure call, that the callee does not, itself, contain any procedure calls, what steps might it omit from the calling sequence? Are there any fields in the AR that the callee would never need?

6.6 ADVANCED TOPICS

The compiler must arrange for the allocation of space to hold the various runtime structures discussed in Section 6.3. For some languages, those structures have lifetimes that do not fit well into the first-in first-out discipline of a stack. In such cases, the language implementation allocates space in the runtime heap—a region of memory set aside for such objects and managed by routines in a runtime support library. The compiler must also arrange storage for other objects that have lifetimes unrelated to the flow of

control, such as many lists in a Scheme program or many objects in a Java program.

We assume a simple interface to the heap, namely, a routine `allocate(size)` and a routine `free(address)`. The `allocate` routine takes an integer argument `size` and returns the address of a block of space in the heap that contains at least `size` bytes. The `free` routine takes the address of a block of previously allocated space in the heap and returns it to the pool of free space. The critical issues that arise in designing algorithms for explicitly managing the heap are the speeds of both `allocate` and `free` and the extent to which the pool of free space becomes fragmented into small blocks.

This section sketches the algorithms involved in allocation and reclamation of space in a runtime heap. Section 6.6.1 focuses on techniques for explicit management of the heap. Along the way, it describes how to implement `free` for each of the schemes. Section 6.6.2 examines implicit deallocation—techniques that avoid the need for `free`.

6.6.1 Explicit Heap Management

Most language implementations include a runtime system that provides support functions for the code generated by the compiler. The runtime system typically includes provision for management of a runtime heap. The actual routines that implement the heap may be language specific, as in a Scheme interpreter or a Java virtual machine, or they may be part of the underlying operating system, as in the Posix implementations of `malloc` and `free`.

While many techniques have been proposed to implement `allocate` and `free`, most of those implementations share common strategies and insights. This section explores a simple strategy, *first-fit allocation*, that exposes most of the issues, and then shows how a strategy such as first fit is used to implement a modern allocator.

First-Fit Allocation

The goal of a first-fit allocator is to allocate and free space in the heap quickly. First fit emphasizes speed over memory utilization. Every block in the heap has a hidden field that holds its size. In general, the size field is located in the word preceding the address returned by `allocate`, as shown in Figure 6.11a. Blocks available for allocation reside on a list called the *free list*. In addition to the mandatory size field, blocks on the free list have additional fields, as shown in Figure 6.11b. Each free block has a pointer to the next block on the free list (set to null in the last block) and a pointer to the block itself in the last word of the block. To initialize the heap, the allocator creates a free list that contains a single large unallocated block.

■ **FIGURE 6.11** Blocks in a First-Fit Allocator.

A call `allocate(`k`)` causes the following sequence of events: The `allocate` routine walks the free list until it discovers a block with size greater than or equal to k plus one word for the `size` field. Assume it finds an appropriate block, b_i. It removes b_i from the free list. If b_i is larger than necessary, `allocate` creates a new free block from the excess space at the end of b_i and places that block on the free list. The `allocate` routine returns a pointer to the second word of b_i.

If `allocate` fails to find a large enough block, it tries to extend the heap. If it succeeds in extending the heap, it returns a block of appropriate size from this newly allocated portion of the heap. If extending the heap fails, `allocate` reports failure (typically by returning a null pointer).

To deallocate a block, the program calls `free` with the address of the block, b_j. The simplest implementation of `free` adds b_j to the head of the free list and returns. This produces a fast `free` routine. Unfortunately, it leads to an allocator that, over time, fragments memory into small blocks.

To overcome this flaw, the allocator can use the pointer at the end of a freed block to coalesce adjacent free blocks. The `free` routine loads the word preceding b_j's size field, which is the end-of-block pointer for the block that immediately precedes b_j in memory. If that word contains a valid pointer, and it points to a matching block header (one whose address plus size field points to the start of b_j), then both b_j and its predecessor are free. The `free` routine can combine them by increasing the predecessor's size field and storing the appropriate pointer at the end of b_j. Combining these blocks lets `free` avoid updating the free list.

To make this scheme work, `allocate` and `free` must maintain the end-of-block pointers. Each time that `free` processes a block, it must update that pointer with the address of the head of the block. The `allocate` routine must invalidate either the `next` pointer or the end-of-block pointer to prevent `free` from coalescing a freed block with an allocated block in which those fields have not been overwritten.

The `free` routine can also try to combine b_j with its successor in memory, b_k. It can use b_j's size field to locate the start of b_k. It can use b_k's size field and end-of-block pointer to determine if b_k is free. If b_k is free, then `free`

ARENA-BASED ALLOCATION

Inside the compiler itself, the compiler writer may find it profitable to use a specialized allocator. Compilers have phase-oriented activity. This lends itself well to an arena-based allocation scheme.

With an arena-based allocator, the program creates an arena at the beginning of an activity. It uses the arena to hold allocated objects that are related in their use. Calls to allocate objects in the arena are satisfied in a stacklike fashion; an allocation involves incrementing a pointer to the arena's high-water mark and returning a pointer to the newly allocated block. No call is used to deallocate individual objects; they are freed when the arena that contains them is deallocated.

The arena-based allocator is a compromise between traditional allocators and garbage-collecting allocators. With an arena-based allocator, the calls to `allocate` can be made lightweight (as in the modern allocator). No freeing calls are needed; the program frees the entire arena in a single call when it finishes the activity for which the arena was created.

can combine the two blocks, removing b_k from the free list, adding b_j to the free list, and updating b_j's size field and end-of-block pointer appropriately. To make the free-list update efficient, the free list should be a doubly linked list. Of course, the pointers are stored in unallocated blocks, so the space overhead is irrelevant. Extra time required to update the doubly linked free list is minimal.

As described, the coalescing scheme depends on the fact that the relationship between the final pointer and the size field in a free block are absent in an allocated block. While it is extremely unlikely that the allocator will identify an allocated block as free, this can happen. To ensure against this unlikely event, the implementor can make the end-of-block pointer a field that exists in both allocated and free blocks. On allocation, the pointer is set to contain an address outside the heap, such as zero. On freeing, the pointer is set to the block's own address. The cost of this added assurance is an extra field in each allocated block and an extra store for each allocation.

Many variations on first-fit allocation have been tried. They trade off the cost of `allocate`, the cost of `free`, the amount of fragmentation produced by a long series of allocations, and the amount of space wasted by returning blocks larger than requested.

Multipool Allocators

Modern allocators are derived from first-fit allocation but simplified by a couple of observations about the behavior of programs. As memory sizes

grew in the early 1980s, it became reasonable to waste some space if doing so led to faster allocation. At the same time, studies of program behavior suggested that real programs allocate memory frequently in a few common sizes and infrequently in large or unusual sizes.

Modern allocators use separate memory pools for several common sizes. Typically, selected sizes are powers of two, starting with a small block size (such as 16 bytes) and running up to the size of a virtual-memory page (typically 4096 or 8192 bytes). Each pool has only one size of block, so `allocate` can return the first block on the appropriate free list, and `free` can simply add the block to the head of the appropriate free list. For requests larger than a page, a separate first-fit allocator is used. Allocators based on these ideas are fast. They work particularly well for heap allocation of activation records.

These changes simplify both `allocate` and `free`. The `allocate` routine must check for an empty free list and adds a new page to the free list if it is empty. The `free` routine inserts the freed block at the head of the free list for its size. A careful implementation could determine the size of a freed block by checking its address against the memory segments allocated for each pool. Alternative schemes include using a size field as before, and, if the allocator places all the storage on a page into a single pool, storing the size of the blocks in a page in the first word of the page.

Debugging Help

Programs written with explicit allocation and deallocation are notoriously difficult to debug. It appears that programmers have difficulty deciding when to free heap-allocated objects. If the allocator can quickly distinguish between an allocated object and a free object, then the heap-management software can provide the programmer with some help in debugging.

For example, to coalesce adjacent free blocks, the allocator needs a pointer from the end of a block back to its head. If an allocated block has that pointer set to an invalid value, then the deallocation routine can check that field and report a runtime error when the program attempts to deallocate a free block or an illegal address—a pointer to anything other than the start of an allocated block.

For a modest additional overhead, heap-management software can provide additional help. By linking together allocated blocks, the allocator can create an environment for memory-allocation debugging tools. A snapshot tool can walk the list of allocated blocks. Tagging blocks by the call site that created them lets the tool expose memory leaks. Timestamping them allows the tool

to provide the programmer with detailed information about memory use. Tools of this sort can provide invaluable help in locating blocks that are never deallocated.

6.6.2 Implicit Deallocation

Many programming languages support implicit deallocation of heap objects. The implementation deallocates memory objects automatically when they are no longer in use. This requires some care in the implementation of both the allocator and the compiled code. To perform implicit deallocation, or *garbage collection*, the compiler and runtime system must include a mechanism for determining when an object is no longer of interest, or *dead*, and a mechanism for reclaiming and recycling the dead space.

Garbage collection
the implicit deallocation of objects that reside on the runtime heap

The work associated with garbage collection can be performed incrementally, for individual statements, or it can be performed as a batch-oriented task that runs on demand, when the free-space pool is exhausted. Reference counting is a classic way to perform incremental garbage collection. Mark-sweep collection is a classic approach to performing batch-oriented collection.

Reference Counting

This technique adds a counter to each heap-allocated object. The counter tracks the number of outstanding pointers that refer to the object. When the allocator creates the object, it sets the reference count to one. Each assignment to a pointer variable adjusts two reference counts. It decrements the reference count of the pointer's preassignment value and increments the reference count of the pointer's postassignment value. When an object's reference count drops to zero, no pointer exists that can reach the object, so the system may safely free the object. Freeing an object can, in turn, discard pointers to other objects. This must decrement the reference counts of those objects. Thus, discarding the last pointer to an abstract syntax tree should free the entire tree. When the root node's reference count drops to zero, it is freed and its descendant's reference counts are decremented. This, in turn, should free the descendants, decrementing the counts of their children. This process continues until the entire AST has been freed.

The presence of pointers in allocated objects creates problems for reference-counting schemes, as follows:

1. The running code needs a mechanism to distinguish pointers from other data. It may either store extra information in the header field for each object or limit the range of pointers to less than a full word and use the

remaining bits to "tag" the pointer. Batch collectors face the same problem and use the same solutions.

2. The amount of work done for a single decrement can grow quite large. If external constraints require bounded deallocation times, the runtime system can adopt a more complex protocol that limits the number of objects deallocated for each pointer assignment. By keeping a queue of objects that must be freed and limiting the number handled on each reference-count adjustment, the system can distribute the cost of freeing objects over a larger set of operations. This amortizes the cost of freeing over the set of all assignments to heap-allocated objects and bounds the work done per assignment.
3. The program might form cyclic graphs with pointers. The reference counts for a cyclic data structure cannot be decremented to zero. When the last external pointer is discarded, the cycle becomes both unreachable and nonrecyclable. To ensure that all such objects are freed, the programmer must break the cycle before discarding the last pointer to the cycle. (The alternative, to perform reachability analysis on the pointers at runtime, would make reference counting prohibitively expensive.) Many categories of heap-allocated objects, such as variable-length strings and activation records, cannot be involved in such cycles.

Reference counting incurs additional cost on every pointer assignment. The amount of work done for a specific pointer assignment can be bounded; in any well-designed scheme, the total cost can be limited to some constant factor times the number of pointer assignments executed plus the number of objects allocated. Proponents of reference counting argue that these overheads are small enough and that the pattern of reuse in reference-counting systems produces good program locality. Opponents of reference counting argue that real programs do more pointer assignments than allocations, so that garbage collection achieves equivalent functionality with less total work.

Batch Collectors

If the collector cannot free any space, then it must request additional space from the system. If none is available, allocation fails.

Batch collectors consider deallocation only when the free-space pool has been exhausted. When the allocator fails to find needed space, it invokes the batch collector. The collector pauses the program's execution, examines the pool of allocated memory to discover unused objects, and reclaims their space. When the collector terminates, the free-space pool is usually nonempty. The allocator can finish its original task and return a newly allocated object to the caller. (As with reference counting, schemes exist that perform collection incrementally to amortize the cost over longer periods of execution.)

Logically, batch collectors proceed in two phases. The first phase discovers the set of objects that can be reached from pointers stored in program variables and compiler-generated temporaries. The collector conservatively assumes that any object reachable in this manner is live and that the remainder are dead. The second phase deallocates and recycles dead objects. Two commonly used techniques are *mark-sweep* collectors and *copying* collectors. They differ in their implementation of the second phase of collection—recycling.

Identifying Live Data

Collecting allocators discover live objects by using a marking algorithm. The collector needs a bit for each object in the heap, called a *mark bit*. This bit can be stored in the object's header, alongside tag information used to record pointer locations or object size. Alternatively, the collector can create a dense bit map for the heap when needed. The initial step clears all the mark bits and builds a worklist that contains all the pointers stored in registers and in variables accessible to current or pending procedures. The second phase of the algorithm walks forward from these pointers and marks every object that is reachable from this set of visible pointers.

Figure 6.12 presents a high-level sketch of a marking algorithm. It is a simple fixed-point computation that halts because the heap is finite and the marks prevent a pointer contained in the heap from entering the `Worklist` more than once. The cost of marking is, in the worst case, proportional to the number of pointers contained in program variables and temporaries plus the size of the heap.

The marking algorithm can be either precise or conservative. The difference lies in how the algorithm determines that a specific data value is a pointer in the final line of the `while` loop.

- In a precise collector, the compiler and runtime system know the type and layout of each object. This information can be recorded in object headers, or it can be known implicitly from the type system. Either way, the marking phase only follows real pointers.
- In a conservative marking phase, the compiler and runtime system may be unsure about the type and layout of some objects. Thus, when an object is marked, the system considers each field that may be a possible pointer. If its value might be a pointer, it is treated as a pointer. Any value that does not represent a word-aligned address might be excluded, as might values that fall outside the known boundaries of the heap.

Conservative collectors have limitations. They fail to reclaim some objects that a precise collector would find. Nonetheless, conservative collectors have

```
Clear all marks
Worklist ← { pointer values from activation records & registers }

while (Worklist ≠ Ø)
   remove p from the Worklist
   if (p→object is unmarked)
      mark p→object
      add pointers from p→object to Worklist
```

■ **FIGURE 6.12** A Simple Marking Algorithm.

been successfully retrofitted into implementations for languages such as C that do not normally support garbage collection.

When the marking algorithm halts, any unmarked object must be unreachable from the program. Thus, the second phase of the collector can treat that object as dead. Some objects marked as live may also be dead. However, the collector lets them survive because it cannot prove them dead. As the second phase traverses the heap to collect the garbage, it can reset the mark fields to "unmarked." This lets the collector avoid the initial traversal of the heap in the marking phase.

Mark-Sweep Collectors

Mark-sweep collectors reclaim and recycle objects by making a linear pass over the heap. The collector adds each unmarked object to the free list (or one of the free lists), where the allocator will find it and reuse it. With a single free list, the same collection of techniques used to coalesce blocks in the first-fit allocator applies. If compaction is desirable, it can be implemented by incrementally shuffling live objects downward during the sweep, or with a postsweep compaction pass.

Copying Collectors

Copying collectors divide memory into two pools, an *old* pool and a *new* pool. The allocator always operates from the old pool. The simplest type of copying collector is called *stop and copy*. When an allocation fails, a stop and copy collector copies all the live data from the old pool into the new pool and swaps the identities of the old and new pools. The act of copying live data compacts it; after collection, all the free space is in a single contiguous block. Collection can be done in two passes, like mark sweep, or it can be done incrementally, as live data is discovered. An incremental scheme can mark objects in the old pool as it copies them to avoid copying the same object multiple times.

An important family of copying collectors are the *generational collectors*. These collectors capitalize on the observation that an object that survives

one collection is more likely to survive subsequent collections. To capitalize on this observation, generational collectors periodically repartition their "new" pool into a "new" and an "old" pool. In this way, successive collections examine only newly allocated objects. Generational schemes vary in how often they declare a new generation, freezing the surviving objects and exempting them from the next collection, and whether or not they periodically re-examine the older generations.

Comparing the Techniques

Garbage collection frees the programmer from needing to worry about when to release memory and from tracking down the inevitable storage leaks that result from attempting to manage allocation and deallocation explicitly. The individual schemes have their strengths and weaknesses. In practice, the benefits of implicit deallocation outweigh the disadvantages of either scheme for most applications.

Reference counting distributes the cost of deallocation more evenly across program execution than does batch collection. However, it increases the cost of every assignment that involves a heap-allocated value—even if the program never runs out of free space. In contrast, batch collectors incur no cost until the allocator fails to find needed space. At that point, however, the program incurs the full cost of collection. Thus, any allocation can provoke a collection.

Mark-sweep collectors examine the entire heap, while copying collectors only examine the live data. Copying collectors actually move every live object, while mark-sweep collectors leave them in place. The tradeoff between these costs will vary with the application's behavior and with the actual costs of various memory references.

Reference-counting implementations and conservative batch collectors have problems recognizing cyclic structures, because they cannot distinguish between references from within the cycle and those from without. The mark-sweep collectors start from an external set of pointers, so they discover that a dead cyclic structure is unreachable. The copying collectors, starting from the same set of pointers, simply fail to copy the objects involved in the cycle.

Copying collectors compact memory as a natural part of the process. The collector can either update all the stored pointers, or it can require use of an indirection table for each object access. A precise mark-sweep collector can compact memory, too. The collector would move objects from one end of memory into free space at the other end. Again, the collector can either rewrite the existing pointers or mandate use of an indirection table.

In general, a good implementor can make both mark sweep and copying work well enough that they are acceptable for most applications. In

applications that cannot tolerate unpredictable overhead, such as real-time controllers, the runtime system must incrementalize the process, as the amortized reference-counting scheme does. Such collectors are called *real-time collectors*.

6.7 SUMMARY AND PERSPECTIVE

The primary rationale for moving beyond assembly language is to provide a more abstract programming model and, thus, raise both programmer productivity and the understandability of programs. Each abstraction that a programming language supports needs a translation to the target machine's instruction set. This chapter has explored the techniques commonly used to translate some of these abstractions.

Procedural programming was invented early in the history of programming. Some of the first procedures were debugging routines written for early computers; the availability of these prewritten routines allowed programmers to understand the runtime state of an errant program. Without such routines, tasks that we now take for granted, such as examining the contents of a variable or asking for a trace of the call stack, required the programmer to enter long machine-language sequences without error.

The introduction of lexical scoping in languages like Algol 60 influenced language design for decades. Most modern programming languages carry forward some of Algol's philosophy toward naming and addressability. Techniques developed to support lexical scoping, such as access links and displays, reduced the runtime cost of this abstraction. These techniques are still used today.

Object-oriented languages take the scoping concepts of ALLs and reorient them in data-directed ways. The compiler for an object-oriented language uses both compile-time and runtime structures invented for lexical scoping to implement the naming discipline imposed by the inheritance hierarchy of a specific program.

Modern languages have added some new twists. By making procedures first-class objects, languages like Scheme have created new control-flow paradigms. These require variations on traditional implementation techniques—for example, heap allocation of activation records. Similarly, the growing acceptance of implicit deallocation requires occasional conservative treatment of a pointer. If the compiler can exercise a little more care and free the programmer from ever deallocating storage again, that appears to be a good tradeoff. (Generations of experience suggest that programmers

are not effective at freeing all the storage that they allocate. They also free objects to which they retain pointers.)

As new programming paradigms emerge, they will introduce new abstractions that require careful thought and implementation. By studying the successful techniques of the past and understanding the constraints and costs involved in real implementations, compiler writers will develop strategies that decrease the runtime penalty for using higher levels of abstraction.

■ CHAPTER NOTES

Much of the material in this chapter comes from the accumulated experience of the compiler-construction community. The best way to learn more about the name-space structures of various languages is to consult the language definitions themselves. These documents are a necessary part of a compiler writer's library.

Procedures appeared in the earliest high-level languages—that is, languages that were more abstract than assembly language. FORTRAN [27] and Algol 60 [273] both had procedures with most of the features found in modern languages. Object-oriented languages appeared in the late 1960s with SIMULA 67 [278] followed by Smalltalk 72 [233].

Lexical scoping was introduced in Algol 60 and has persisted to the present day. The early Algol compilers introduced most of the support mechanisms described in this chapter, including activation records, access links, and parameter-passing techniques. Much of the material from Sections 6.3 through 6.5 was present in these early systems [293]. Optimizations quickly appeared, like folding storage for a block-level scope into the containing procedure's activation record. The IBM 370 linkage conventions recognized the difference between leaf procedures and others; they avoided allocating a register save area for leaf routines. Murtagh took a more complete and systematic approach to coalescing activation records [272].

The classic reference on memory allocation schemes is Knuth's *Art of Computer Programming* [231, § 2.5]. Modern multipool allocators appeared in the early 1980s. Reference counting dates to the early 1960s and has been used in many systems [95, 125]. Cohen and later Wilson, provide broad surveys of the literature on garbage collection [92, 350]. Conservative collectors were introduced by Boehm and Weiser [44, 46, 120]. Copying collectors appeared in response to virtual memory systems [79, 144]; they led, somewhat naturally, to the generational collectors in widespread use today [247, 337]. Hanson introduced the notion of arena-based allocation [179].

■ EXERCISES

Section 6.2

1. Show the call tree and execution history for the following C program:

```
int Sub(int i, int j) {
   return i - j;
}
int Mul(int i, int j) {
   return i * j;
}
int Delta(int a, int b, int c) {
   return Sub(Mul(b,b), Mul(Mul(4,a),c));
}
void main() {
   int a, b, c, delta;
   scanf("%d %d %d", &a, &b, &c);
   delta = Delta(a, b, c);
   if (delta == 0)
     puts("Two equal roots");
   else if (delta > 0)
     puts("Two different roots");
   else
     puts("No root");
}
```

2. Show the call tree and execution history for the following C program:

```
void Output(int n, int x) {
   printf("The value of %d! is %s.\n", n, x);
}
int Fat(int n) {
   int x;
     if (n > 1)
   x = n * Fat(n - 1);
   else
     x = 1;
   Output(n, x);
   return x;
}
void main() {
   Fat(4);
}
```

3. Consider the following Pascal program, in which only procedure calls and variable declarations are shown: Section 6.3

```
1   program Main(input, output);
2     var a, b, c : integer;
3     procedure P4; forward;
4     procedure P1;
5       procedure P2;
6         begin
7         end;
8       var b, d, f : integer;
9       procedure P3;
10        var a, b : integer;
11        begin
12          P2;
13          end;
14      begin
15        P2;
16        P4;
17        P3;
18        end;
19    var d, e : integer;
20    procedure P4;
21      var a, c, g : integer;
22      procedure P5;
23        var c, d : integer;
24        begin
25          P1;
26          end;
27      var d : integer;
28      begin
29        P1;
30        P5;
31        end;
32    begin
33      P1;
34      P4;
35      end.
```

a. Construct a static coordinate table, similar to the one in Figure 6.3.
b. Construct a graph to show the nesting relationships in the program.
c. Construct a graph to show the calling relationships in the program.

4. Some programming languages allow the programmer to use functions in the initialization of local variables but not in the initialization of global variables.
   a. Is there an implementation rationale to explain this seeming quirk of the language definition?
   b. What mechanisms would be needed to allow initialization of a global variable with the result of a function call?

5. The compiler writer can optimize the allocation of ARs in several ways. For example, the compiler might:
   a. Allocate ARs for leaf procedures statically.
   b. Combine the ARs for procedures that are always called together. (When α is called, it always calls β.)
   c. Use an arena-style allocator in place of heap allocation of ARs.

   For each scheme, consider the following questions:
   a. What fraction of the calls might benefit? In the best case? In the worst case?
   b. What is the impact on runtime space utilization?

6. Draw the structures that the compiler would need to create to support an object of type `Dumbo`, defined as follows:

```
class Elephant {
   private int Length;
   private int Weight;
   static int type;

   public int GetLen();
   public int GetTyp();
}

class Dumbo extends Elephant {
   private int EarSize;
   private boolean Fly;

   public boolean CanFly();
}
```

7. In a programming language with an open class structure, the number of method invocations that need runtime name resolution, or dynamic dispatch, can be large. A method cache, as described in Section 6.3.4, can reduce the runtime cost of these lookups by short-circuiting them. As an alternative to a global method cache, the implementation might maintain a single entry method cache at each call site—an inline

```
1   procedure main;
2     var a : array[1...3] of int;
3         i : int;
4     procedure p2(e : int);
5       begin
6         e := e + 3;
7         a[i] := 5;
8         i := 2;
9         e := e + 4;
10        end;
11    begin
12      a := [1, 10, 77];
13      i := 1;
14      p2(a[i]);
15      for i := 1 to 3 do
16        print(a[i]);
17      end.
```

■ **FIGURE 6.13** Program for Problem 8.

method cache that records record the address of the method most recently dispatched from that site, along with its class. Develop pseudocode to use and maintain such an inline method cache. Explain the initialization of the inline method caches and any modifications to the general method lookup routine required to support inline method caches.

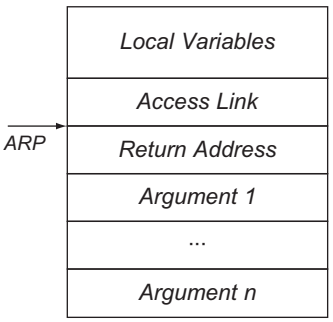
8. Consider the program written in Pascal-like pseudo code shown in Figure 6.13. Simulate its execution under call-by-value, call-by-reference, call-by-name, and call-by-value-result parameter binding rules. Show the results of the print statements in each case. Section 6.4

9. The possibility that two distinct variables refer to the same object (memory area) is considered undesirable in programming languages. Consider the following Pascal procedure, with parameters passed by reference:

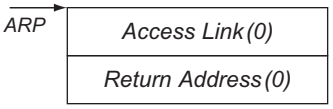
```
procedure mystery(var x, y : integer);
  begin
    x := x + y;
    y := x - y;
    x := x - y;
  end;
```

```
1   program main(input, output);
2     procedure P1( function g(b: integer): integer);
3       var a: integer;
4       begin
5         a := 3;
6         writeln(g(2))
7         end;
8     procedure P2;
9       var a: integer;
10      function F1(b: integer): integer;
11        begin
12          F1 := a + b
13          end;
14      procedure P3;
15        var a:integer;
16        begin
17          a := 7;
18          P1(F1)
19          end;
20      begin
21        a := 0;
22        P3
23        end;
24    begin
25      P2
26      end.
```

(a) Example Pascal Program



(b) Activation Record Structure



(c) Initial Activation Record

■ **FIGURE 6.14** Program for Problem 10.

If no overflow or underflow occurs during the arithmetic operations:

a. What result does `mystery` produce when it is called with two distinct variables, `a` and `b`?

b. What would be the expected result if `mystery` is invoked with a single variable `a` passed to both parameters? What is the actual result in this case?

Section 6.5

10. Consider the Pascal program shown in Figure 6.14a. Suppose that the implementation uses ARs as shown in Figure 6.14b. (Some fields have been omitted for simplicity.) The implementation stack allocates the ARs, with the stack growing toward the top of the page. The ARP is

the only pointer to the AR, so access links are previous values of the ARP. Finally, Figure 6.14c shows the initial AR for a computation. For the example program in Figure 6.14a, draw the set of its ARs just prior to the return from function `F1`. Include all entries in the ARs. Use line numbers for return addresses. Draw directed arcs for access links. Label the values of local variables and parameters. Label each AR with its procedure name.

11. Assume that the compiler is capable of analyzing the code to determine facts such as "*from this point on, variable* `v` *is not used again in this procedure*" or "*variable* `v` *has its next use in line 11 of this procedure*," and that the compiler keeps all local variables in registers for the following three procedures:

```
procedure main
   integer a, b, c
   b = a + c;
   c = f1(a,b);
   call print(c);
   end;
procedure f1(integer x, y)
   integer v;
   v = x * y;
   call print(v);
   call f2(v);
   return -x;
   end;
procedure f2(integer q)
   integer k, r;
   ...
   k = q / r;
   end;
```

a. Variable `x` in procedure `f1` is live across two procedure calls. For the fastest execution of the compiled code, should the compiler keep it in a caller-saves or callee-saves register? Justify your answer.

b. Consider variables `a` and `c` in procedure `main`. Should the compiler keep them in caller-saves or callee-saves registers, again assuming that the compiler is trying to maximize the speed of the compiled code? Justify your answer.

12. Consider the following Pascal program. Assume that the ARs follow the same layout as in problem 10,with the same initial condition, *except* that the implementation uses a global display rather than access links.

```
1   program main(input, output);
2     var x : integer;
3         a : float;
4     procedure p1();
5       var g:character;
6       begin
7         ...
8         end;
9     procedure p2();
10      var h:character;
11      procedure p3();
12        var h,i:integer;
13        begin
14          p1();
15          end;
16      begin
17        p3();
18        end;
19    begin
20      p2();
21      end
```

Draw the set of ARs that are on the runtime stack when the program reaches line 7 in procedure `p1`.