Stack Allocation of Space

Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack. Each time a procedure[1] is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack. As we shall see, this arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allows us to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls.

1. Activation Trees

Stack allocation would not be feasible if procedure calls, or *activations* of pro-cedures, did not nest in time. The following example illustrates nesting of procedure calls.

E x a m p l e 7 . 1 : Figure 7.2 contains a sketch of a program that reads nine inte-gers into an array *a* and sorts them using the recursive quicksort algorithm.

The main function has three tasks. It calls *readArray,* sets the sentinels, and then calls *quicksort* on the entire data array. Figure 7.3 suggests a sequence of calls that might result from an execution of the program. In this execution, the call to *partition(l,9)* returns 4, so *a[l]* through a[3] hold elements less than its chosen separator value *v,* while the larger elements are in a [5] through a [9]. •

In this example, as is true in general, procedure activations are nested in time. If an activation of procedure *p* calls procedure *q,* then that activation of *q* must end before the activation of *p* can end. There are three common cases:

1. The activation of *q* terminates normally. Then in essentially any language, control resumes just after the point of *p* at which the call to *q* was made.

The activation of *q,* or some procedure *q* called, either directly or indi-rectly, aborts; i.e., it becomes impossible for execution to continue. In that case, *p* ends simultaneously with *q.*

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

3. The activation of q terminates because of an exception that *q* cannot han-dle. Procedure p may handle the exception, in which case the activation of *q* has terminated while the activation of *p* continues, although not nec-essarily from the point at which the call to *q* was made. If *p* cannot handle the exception, then this activation of *p* terminates at the same time as the activation of *q,* and presumably the exception will be handled by some other open activation of a procedure.

We therefore can represent the activations of procedures during the running of an entire program by a tree, called an activation tree. Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program. At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p. We show these activations in the order that they are called, from left to right. Notice that one child must finish before the activation to its right can begin.

A Version of Quicksort

The sketch of a quicksort program in Fig. 7.2 uses two auxiliary functions *readArray* and *partition.* The function *readArray* is used only to load the data into the array *a.* The first and last elements of *a* are not used for data, but rather for "sentinels" set in the main function. We assume *a[0]* is set to a value lower than any possible data value, and a[10] is set to a value higher than any data value.

The function *partition* divides a portion of the array, delimited by the arguments *m* and n, so the low elements of *a[m]* through *a[n]* are at the beginning, and the high elements are at the end, although neither group is necessarily in sorted order. We shall not go into the way *partition* works, except that it may rely on the existence of the sentinels. One possible algorithm for *partition* is suggested by the more detailed code in Fig. 9.1.

Recursive procedure *quicksort* first decides if it needs to sort more than one element of the array. Note that one element is always "sorted," so quicksort has nothing to do in that case. If there are elements to sort, quicksort first calls partition, which returns an index i to separate the low and high elements. These two groups of elements are then sorted by two recursive calls to quicksort.

E x a m p l e 7 . 2 : One possible activation tree that completes the sequence of calls and returns suggested in Fig. 7.3 is shown in Fig. 7.4. Functions are represented by the first letters of their names. Remember that this tree is only one possibility, since the arguments of subsequent calls, and also the number of calls along any branch is influenced by the values returned by *partition.* •

The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:

The sequence of procedure calls corresponds to a preorder traversal of the activation tree.

The sequence of returns corresponds to a postorder traversal of the acti-vation tree.

Suppose that control lies within a particular activation of some procedure, corresponding to a node *N* of the activation tree. Then the activations that are currently open *(live)* are those that correspond to node *N* and its ancestors. The order in which these activations were called is the order in which they appear along the path to *N,* starting at the root, and they will return in the reverse of that order.

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

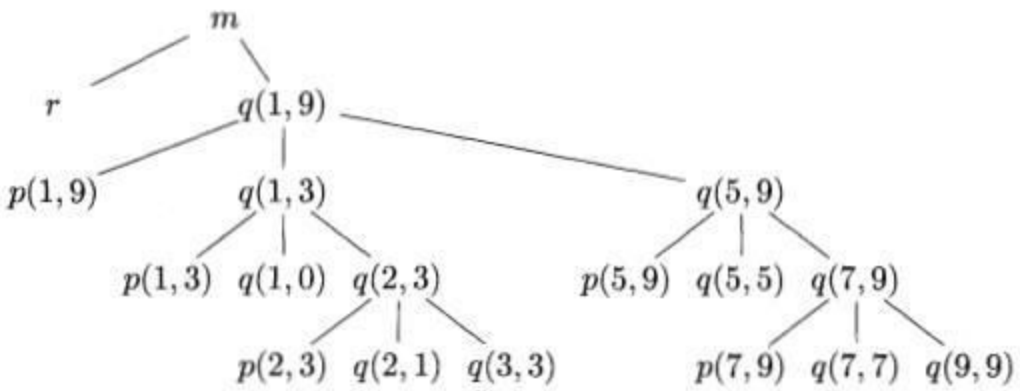

Figure 7.4: Activation tree representing calls during an execution of *quicksort*

2. Activation Records

Procedure calls and returns are usually managed by a run-time stack called the *control stack.* Each live activation has an *activation record* (sometimes called a *frame)* on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides. The latter activation has its record at the top of the stack.

Example **7 . 3 :** If control is currently in the activation 0(2,3) of the tree of Fig. 7.4, then the activation record for *q(2,*3) is at the top of the control stack. Just below is the activation record

for 0(1,3), the parent of 0(2,3) in the tree. Below that is the activation record 0(1,9), and at the bottom is the activation record for m, the main function and root of the activation tree.

We shall conventionally draw control stacks with the bottom of the stack higher than the top, so the elements in an activation record that appear lowest on the page are actually closest to the top of the stack.

The contents of activation records vary with the language being imple-mented. Here is a list of the kinds of data that might appear in an activation record (see Fig. 7.5 for a summary and possible order for these elements):

Actual parameters

Returned values

Control link

Access link

Saved machine status

Local data

Temporaries

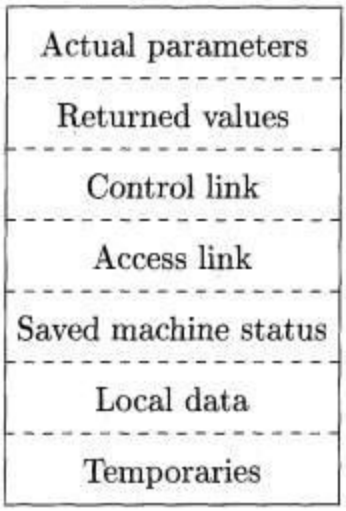

Figure 7.5: A general activation record

Temporary values, such as those arising from the evaluation of expres-sions, in cases where those temporaries cannot be held in registers.

Local data belonging to the procedure whose activation record this is.

A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

An "access link" may be needed to locate data needed by the called proce-dure but found elsewhere, e.g., in another activation record. Access links are discussed in Section 7.3.5.

5. A *control link,* pointing to the activation record of the caller.

Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

Example 7.4 : Figure 7.6 shows snapshots of the run-time stack as control flows through the activation tree of Fig. 7.4. Dashed lines in the partial trees go to activations that have ended. Since array a is global, space is allocated for it before execution begins with an activation of procedure main, as shown in Fig. 7.6(a).

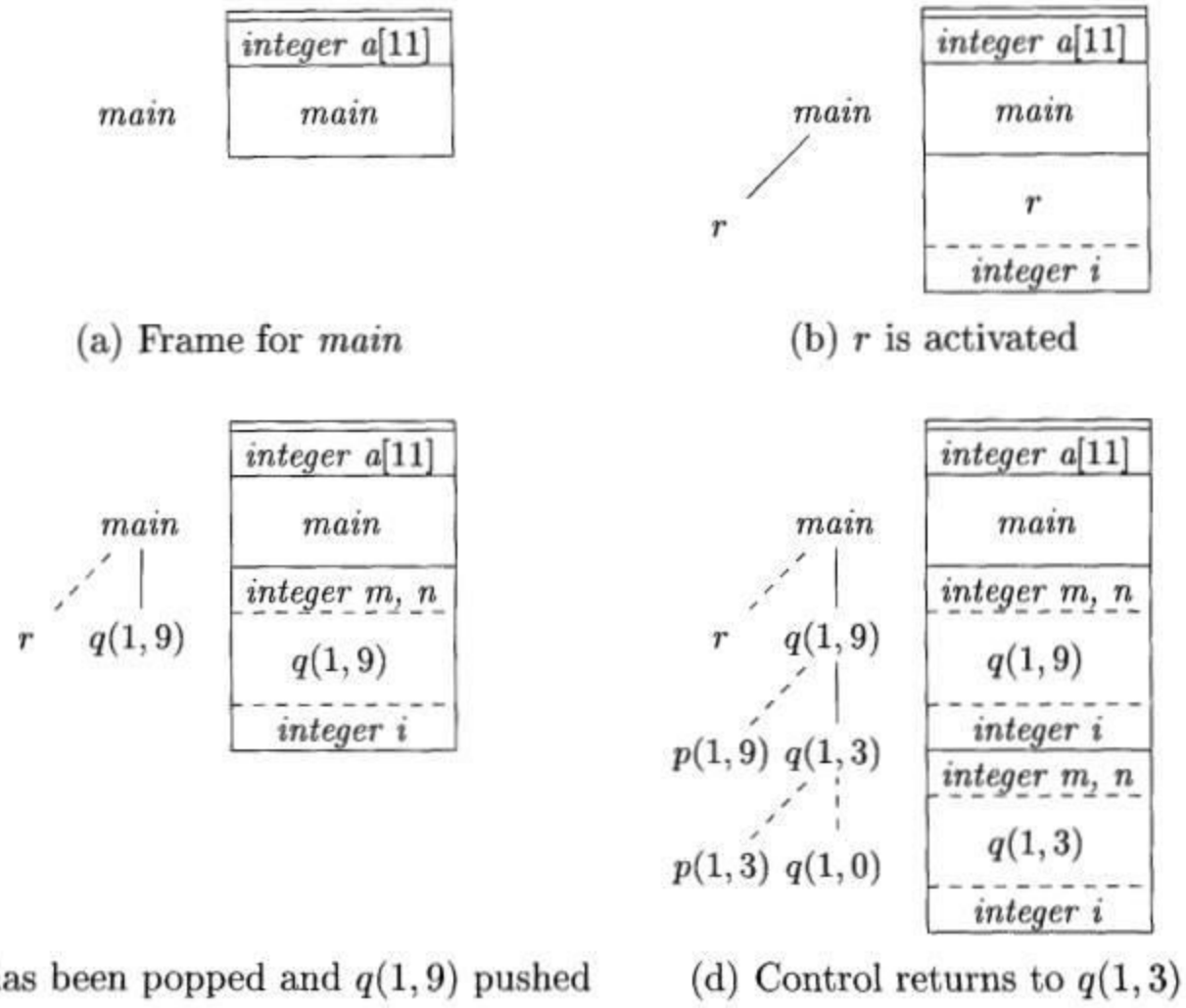

Figure 7.6: Downward-growing stack of activation records

When control reaches the first call in the body of *main,* procedure r is activated, and its activation record is pushed onto the stack (Fig. 7.6(b)). The activation record for r contains space for local variable *i.* Recall that the top of stack is at the bottom of diagrams. When control returns from this activation, its record is popped, leaving just the record for *main* on the stack.

Control then reaches the call to *q* (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack, as in Fig. 7.6(c). The activation record for *q* contains space for the parameters *m* and *n* and the local variable *i,* following the general layout in Fig. 7.5. Notice that space once used by the call of r is reused on the stack. No trace of data local to r will be available to *q(l,* 9). When *q(l,* 9) returns, the stack again has only the activation record for *main.*

Several activations occur between the last two snapshots in Fig. 7.6. A recursive call to g(l,3) was made. Activations p (l , 3) and q(l,0) have begun and ended during the lifetime of q(l, 3), leaving the activation record for q(l, 3) on top (Fig. 7.6(d)). Notice that when a procedure is recursive, it is normal to have several of its activation records on the stack at the same time. •

3. Calling Sequences

Procedure calls are implemented by what are known as *calling sequences,* which consists of code that allocates an activation record on the stack and enters information into its fields. A *return sequence* is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.

Calling sequences and the layout of activation records may differ greatly, even among implementations of the same language. The code in a calling se-quence is often divided between the calling procedure (the "caller") and the procedure it calls (the "callee"). There is no exact division of run-time tasks between caller and callee; the source language, the target machine, and the op-erating system impose requirements that may favor one solution over another. In general, if a procedure is called from *n* different points, then the portion of the calling sequence assigned to the caller is generated n times. However, the portion assigned to the callee is generated only once. Hence, it is desirable to put as much of the calling sequence into the callee as possible — whatever the callee can be relied upon to know. We shall see, however, that the callee cannot know everything.

When designing calling sequences and the layout of activation records, the following principles are helpful:

1. Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record. The motivation is that the caller can compute the values of the actual parameters of the call and place them on top of its own activation record, without having to create the entire activation record of the callee, or even to know the layout of that record.

2. Moreover, it allows for the use of procedures that do not always take the same number or type of arguments, such as C's p r i n t f function. The callee knows where to place the return value, relative to its own activation record, while however many arguments are present will appear sequentially below that place on the stack.

3. Fixed-length items are generally placed in the middle. From Fig. 7.5, such items typically include the control link, the access link, and the machine status fields. If exactly the same components of the machine status are saved for each call, then the same code can do the

saving and restoring for each. Moreover, if we standardize the machine's status information, then programs such as debuggers will have an easier time deciphering the stack contents if an error occurs.

Items whose size may not be known early enough are placed at the end of the activation record. Most local variables have a fixed length, which can be determined by the compiler by examining the type of the variable. However, some local variables have a size that cannot be determined until the program executes; the most common example is a dynamically sized array, where the value of one of the callee's parameters determines the length of the array. Moreover, the amount of space needed for tempo-raries usually depends on how successful the code-generation phase is in keeping temporaries in registers. Thus, while the space needed for tem-poraries is eventually known to the compiler, it may not be known when the intermediate code is first generated.

4. We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of the fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer. A consequence of this approach is that variable-length fields in the activation records are actually "above" the top-of-stack. Their offsets need to be calculated at run time, but they too can be accessed from the top-of-stack pointer, by using a positive offset.

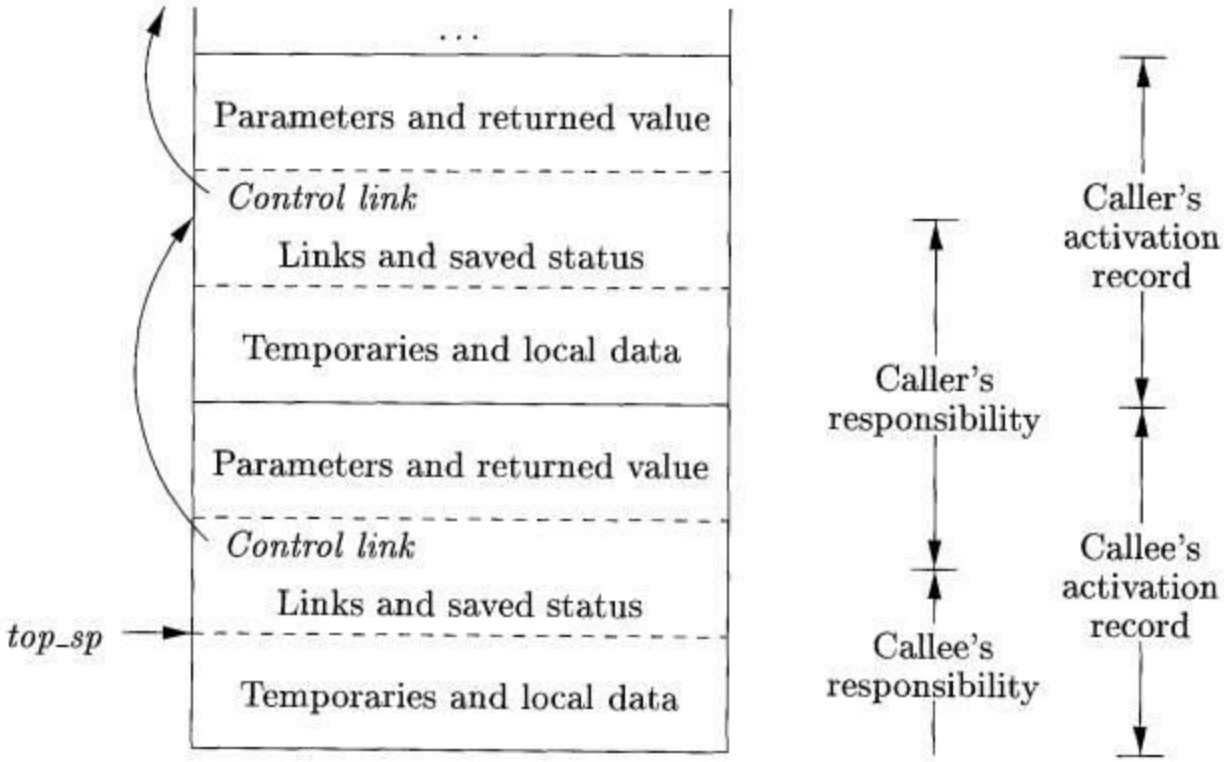

Figure 7.7: Division of tasks between caller and callee

An example of how caller and callee might cooperate in managing the stack is suggested by Fig. 7.7. A register topsp points to the end of the machine-status field in the current top activation record. This position within the callee's activation record is known to the caller, so the caller can be made responsible for setting topsp before control is passed to the callee. The calling sequence and its division between caller and callee is as follows:

1. The caller evaluates the actual parameters.

The caller stores a return address and the old value of *topsp* into the callee's activation record. The caller then increments *topsp* to the po-sition shown in Fig. 7.7. That is, *topsp* is moved past the caller's local data and temporaries and the callee's parameters and status fields.

The callee saves the register values and other status information.

The callee initializes its local data and begins execution.

A suitable, corresponding return sequence is:

1. The callee places the return value next to the parameters, as in Fig. 7.5.

2. Using information in the machine-status field, the callee restores *topsp* and other registers, and then branches to the return address that the caller placed in the status field.

3. Although *topsp* has been decremented, the caller knows where the return value is, relative to the current value of *topsp;* the caller therefore may use that value.

The above calling and return sequences allow the number of arguments of the called procedure to vary from call to call (e.g., as in C's p r i n t f function). Note that at compile time, the target code of the caller knows the number and types of arguments it is supplying to the callee. Hence the caller knows the size of the parameter area. The target code of the callee, however, must be prepared to handle other calls as well, so it waits until it is called and then examines the parameter field. Using the organization of Fig. 7.7, information describing the parameters must be placed next to the status field, so the callee can find it. For example, in the p r i n t f function of C, the first argument describes the remaining arguments, so once the first argument has been located, the caller can find whatever other arguments there are.

4. **Variable-Length Data on the Stack**

The run-time memory-management system must deal frequently with the allo-cation of space for objects the sizes of which are not known at compile time, but which are local to a procedure and thus may be allocated on the stack. In modern languages, objects whose size cannot be determined at compile time are allocated space in the heap, the storage structure that we discuss in Section 7.4. However, it is also possible to allocate objects, arrays, or other structures of unknown size on the stack, and we discuss here how to do so. The reason to prefer placing objects on the stack if possible is that we avoid the expense of garbage collecting their space. Note that the stack can be used only for an object if it is local to a procedure and becomes inaccessible when the procedure returns.

A common strategy for allocating variable-length arrays (i.e., arrays whose size depends on the value of one or more parameters of the called procedure) is shown in Fig. 7.8. The same scheme works for objects of any type if they are

local to the procedure called and have a size that depends on the parameters of the call.

In Fig. 7.8, procedure *p* has three local arrays, whose sizes we suppose cannot be determined at compile time. The storage for these arrays is not part of the activation record for p, although it does appear on the stack. Only a pointer to the beginning of each array appears in the activation record itself. Thus, when *p* is executing, these pointers are at known offsets from the top-of-stack pointer, so the target code can access array elements through these pointers.

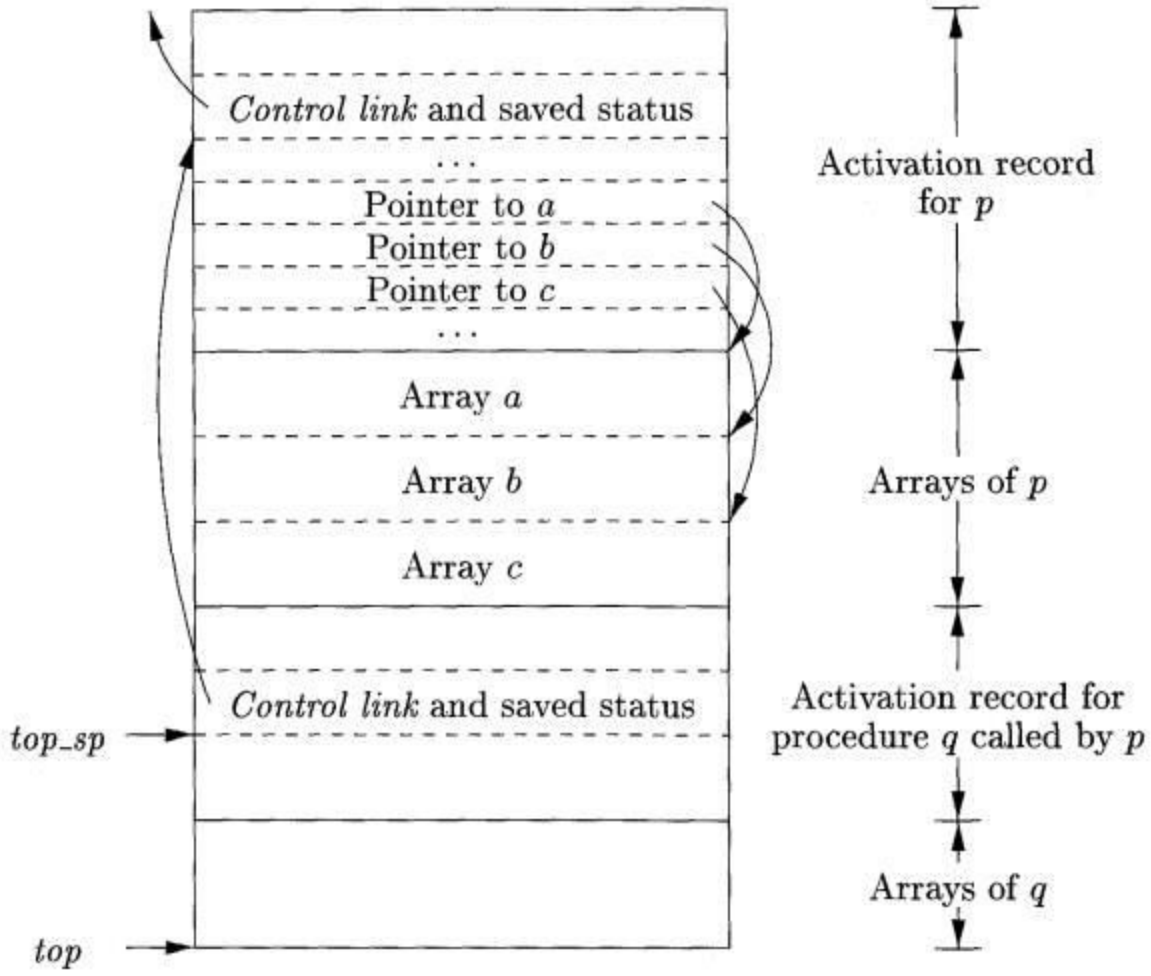

Figure 7.8: Access to dynamically allocated arrays

Also shown in Fig. 7.8 is the activation record for a procedure q, called by p. The activation record for q begins after the arrays of p, and any variable-length arrays of q are located beyond that . Access to the data on the stack is through two pointers, top and topsp.

Here, top marks the actual top of stack; it points to the position at which the next activation record will begin. The second, topsp is used to find local, fixed-length fields of the top activation record. For consistency with Fig. 7.7, we shall suppose that topsp points to the end of the machine-status field. In Fig. 7.8, topsp points to the end of this field in the activation record for q. From there, we can find the control-link field for q, which leads us to the place in the activation record for p where topsp pointed when p was on top. The code to reposition top and topsp can be generated at compile time,

in terms of sizes that will become known at run time. When q returns, topsp can be restored from the saved control link in the activation record for *q*. The new value of *top* is (the old unrestored value of) *topsp* minus the length of the machine-status, control and access link, return-value, and parameter fields (as in Fig. 7.5) in q's activation record. This length is known at compile time to the caller, although it may depend on the caller, if the number of parameters can vary across calls to *q*.

5. Exercises for Section 7.2

Exercise 7 . 2 . 1 : Suppose that the program of Fig. 7.2 uses a *partition* function that always picks *a[m]* as the separator *v*. Also, when the array *a[m],... ,a[n]* is reordered, assume that the order is preserved as much as possible. That is, first come all the elements less than *v,* in their original order, then all elements equal to *v,* and finally all elements greater than *v,* in their original order.

Draw the activation tree when the numbers 9 , 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1 are sorted.

What is the largest number of activation records that ever appear together on the stack?

Exercise 7 . 2 . 2: Repeat Exercise 7.2.1 when the initial order of the numbers

is 1,3,5,7,9,2,4,6,8.

Exercise 7 . 2 . 3 : In Fig. 7.9 is C code to compute Fibonacci numbers recur-sively. Suppose that the activation record for / includes the following elements in order: (return value, argument n, local *s,* local *t);* there will normally be other elements in the activation record as well. The questions below assume that the initial call is / (5) .

Show the complete activation tree.

What does the stack and its activation records look like the first time / (1) is about to return?

c) What does the stack and its activation records look like the fifth time / (1) is about to return?

Exercise 7 . 2 . 4: Here is a sketch of two C functions / and *g:*

int f(int x) { int i; ••• return i+1; ••• }

int g(int y) { int j; ••• f(j+D ••• 1

That is, function *g* calls /. Draw the top of the stack, starting with the acti-vation record for *g,* after *g* calls /, and / is about to return. You can consider only return values, parameters, control links, and space for local variables; you do not have to consider stored state or temporary or local values not shown in the code sketch. However, you should indicate:

```
int f(int n) {
    int t, s;
    if (n < 2) return 1;
    s = f(n-1);
    t = f(n-2);
    return s+t;
}
```

Figure 7.9: Fibonacci program for Exercise 7.2.3

Which function creates the space on the stack for each element?

Which function writes the value of each element?

To which activation record does the element belong?

Exercise 7.2.5 : In a language that passes parameters by reference, there is a that does the function f(x,y) following:

```
x = x + 1; y = y + 2; return x+y;
```

If *a* is assigned the value 3, and then f (a , a) is called, what is returned?

Exercise 7 . 2 . 6 : The C function f is defined by:

```
int f(int x, *py, **ppz) {
    **ppz += 1; *py += 2; x += 3; return x+y+z;
}
```

Variable a is a pointer to 6; variable 6 is a pointer to c, and c is an integer currently with value 4. If we call f (c , 6, a), what is returned?

Access to Nonlocal Data on the Stack

In this section, we consider how procedures access their data. Especially im-portant is the mechanism for finding data used within a procedure *p* but that does not belong to *p*. Access becomes more complicated in languages where procedures can be declared inside other procedures. We therefore begin with the simple case of C functions, and then introduce a language, ML, that permits both nested function declarations and functions as "first-class objects;" that is, functions can take functions as arguments and return functions as values. This capability can be supported by modifying the implementation of the run-time stack, and we shall consider several options for modifying the stack frames of Section 7.2.

1. Data Access Without Nested Procedures

In the C family of languages, all variables are defined either within a single function or outside any function ("globally"). Most importantly, it is impossible to declare one procedure whose scope is entirely within another procedure. Rather, a global variable *v* has a scope consisting of all the functions that follow the declaration of *v,* except where there is a local definition of the

identifier *v*. Variables declared within a function have a scope consisting of that function only, or part of it, if the function has nested blocks, as discussed in Section 1.6.3.

For languages that do not allow nested procedure declarations, allocation of storage for variables and access to those variables is simple:

Global variables are allocated static storage. The locations of these vari-ables remain fixed and are known at compile time. So to access any variable that is not local to the currently executing procedure, we simply use the statically determined address.

2. Any other name must be local to the activation at the top of the stack.

We may access these variables through the topsp pointer of the stack.

An important benefit of static allocation for globals is that declared proce-dures may be passed as parameters or returned as results (in C, a pointer to the function is passed), with no substantial change in the data-access strategy. With the C static-scoping rule, and without nested procedures, any name non-local to one procedure is nonlocal to all procedures, regardless of how they are activated. Similarly, if a procedure is returned as a result, then any nonlocal name refers to the storage statically allocated for it.

2. Issues With Nested Procedures

Access becomes far more complicated when a language allows procedure dec-larations to be nested and also uses the normal static scoping rule; that is, a procedure can access variables of the procedures whose declarations surround its own declaration, following the nested scoping rule described for blocks in Section 1.6.3. The reason is that knowing at compile time that the declaration of *p* is immediately nested within *q* does not tell us the relative positions of their activation records at run time. In fact, since either *p* or *q* or both may be recursive, there may be several activation records of *p* and/or *q* on the stack.

Finding the declaration that applies to a nonlocal name *x in a* nested pro-cedure *p* is a static decision; it can be done by an extension of the static-scope rule for blocks. Suppose *x* is declared in the enclosing procedure *q*. Finding the relevant activation of *q* from an activation of *p* is a dynamic decision; it re-quires additional run-time information about activations. One possible solution to this problem is to use "access links," which we introduce in Section 7.3.5.

3. A Language With Nested Procedure Declarations

The C family of languages, and many other familiar languages do not support nested procedures, so we introduce one that does. The history of nested pro-cedures in languages is long. Algol 60, an ancestor of C, had this capability, as did its descendant Pascal, a once-popular teaching language. Of the later languages with nested procedures, one of the most influential is ML, and it is this language whose syntax and semantics we shall borrow (see the box on "More about ML" for some of the interesting features of ML):

ML is a *functional language,* meaning that variables, once declared and initialized, are not changed. There are only a few exceptions, such as the

array, whose elements can be changed by special function calls.

• Variables are defined, and have their unchangeable values initialized, by a statement of the form:

v a l (name) = (expression)

• Functions are defined using the syntax:

fun (name) ((arguments)) = (body)

• For function bodies we shall use let-statements of the form:

let (list of definitions) in (statements) end The definitions are normally v a l or fun statements. The scope of each such definition consists of all following definitions, up to the in, and all the statements up to the end. Most importantly, function definitions can be nested. For example, the body of a function p can contain a let-statement that includes the definition of another (nested) function q. Similarly, q can have function definitions within its own body, leading to arbitrarily deep nesting of function

4. Nesting Depth

Let us give *nesting depth* 1 to procedures that are not nested within any other procedure. For example, all C functions are at nesting depth 1. However, if a procedure *p* is defined immediately within a procedure at nesting depth *i,* then give *p* the nesting depth *i* +

E x a m p l e **7 . 5:** Figure 7.10 contains a sketch in ML of our running quicksort example. The only function at nesting depth 1 is the outermost function, *sort,* which reads an array a of 9 integers and sorts them using the quicksort algo-rithm. Defined within *sort,* at line (2), is the array *a* itself. Notice the form of the ML declaration. The first argument of a r r a y says we want the array to have 11 elements; all ML arrays are indexed by integers starting with 0, so this array is quite similar to the C array *a* from Fig. 7.2. The second argument of a r r a y says that initially, all elements of the array *a* hold the value 0. This choice of initial value lets the ML compiler deduce that *a* is an integer array, since 0 is an integer, so we never have to declare a type for *a.*

More About ML

In addition to being almost purely functional, ML presents a number of other surprises to the programmer who is used to C and its family.

ML supports *higher-order functions.* That is, a function can take functions as arguments, and can construct and return other func-tions. Those functions, in turn, can take functions as arguments, to any level.

ML has essentially no iteration, as in C's for- and while-statements, for instance. Rather, the effect of iteration is achieved by recur sion. This approach is essential in a functional language, since we cannot change the value of an iteration variable like i in " f o r (i = 0 ; i<10; i++)" of C. Instead, ML would make i a function argument, and the function would call itself with progressively higher values of i until the limit was reached.

• ML supports lists and labeled tree structures as primitive data types.

ML does not require declaration of variable types. Rather, it deduces types at compile time, and treats it as an error if it cannot. For example, v a l x = 1 evidently makes x have integer type, and if we also see v a l y = 2*x, then we know y is also an integer.

Also declared within *sort* are several functions: *readArray, exchange,* and *quicksort.* On lines (4) and (6) we suggest that *readArray* and *exchange* each access the array *a.* Note that in ML, array accesses can violate the functional nature of the language, and both these functions actually change values of a's elements, as in the C version of quicksort. Since each of these three

functions is defined immediately within a function at nesting depth 1, their nesting depths are all 2.

Lines (7) through (11) show some of the detail of quicksort. Local value v, the pivot for the partition, is declared at line (8). Function partition is defined at line (9). In line (10) we suggest that partition accesses both the array a and the pivot value v, and also calls the function exchange. Since partition is defined immediately within a function at nesting depth 2, it is at depth 3. Line

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ··· ;
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

(11) suggests that quicksort accesses variables a and v, the function partition, and itself recursively.

Line (12) suggests that the outer function sort accesses a and calls the two procedures readArray and quicksort. •

5. Access Links

A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the *access link* to each activation record. If procedure *p* is nested

immediately within procedure *q* in the source code, then the access link in any activation of *p* points to the most recent activation of *q*. Note that the nesting depth of *q* must be exactly one less than the nesting depth of p. Access links form a chain from the activation record at the top of the stack to a sequence of activations at progressively lower nesting depths. Along this chain are all the activations whose data and procedures are accessible to the currently executing procedure.

Suppose that the procedure *p* at the top of the stack is at nesting depth n_p, and *p* needs to access *x,* which is an element defined within some procedure *q* that surrounds p and has nesting depth nq. Note that nq =< np, with equality only if p and q are the same procedure. To find x, we start at the activation record for p at the top of the stack and follow the access link np — nq times, from activation record to activation record. Finally, we wind up at an activation record for q, and it will always be the most recent (highest) activation record for *q* that currently appears on the stack. This activation record contains the element *x* that we want. Since the compiler knows the layout of activation records, *x* will be found at some fixed offset from the position in g's activation record that we can reach by following the last access link.

E x a m p l e 7 . 6 : Figure 7.11 shows a sequence of stacks that might result from execution of the function *sort* of Fig. 7.10. As before, we represent function names by their first letters, and we show some of the data that might appear in the various activation records, as well as the access link for each activation. In Fig. 7.11(a), we see the situation after *sort* has called *readArray* to load input into the array *a* and then called *quicksort(l,* 9) to sort the array. The access link from *quicksort(l,* 9) points to the activation record for *sort,* not because *sort* called *quicksort* but because *sort* is the most closely nested function surrounding *quicksort* in the program of Fig. 7.10.

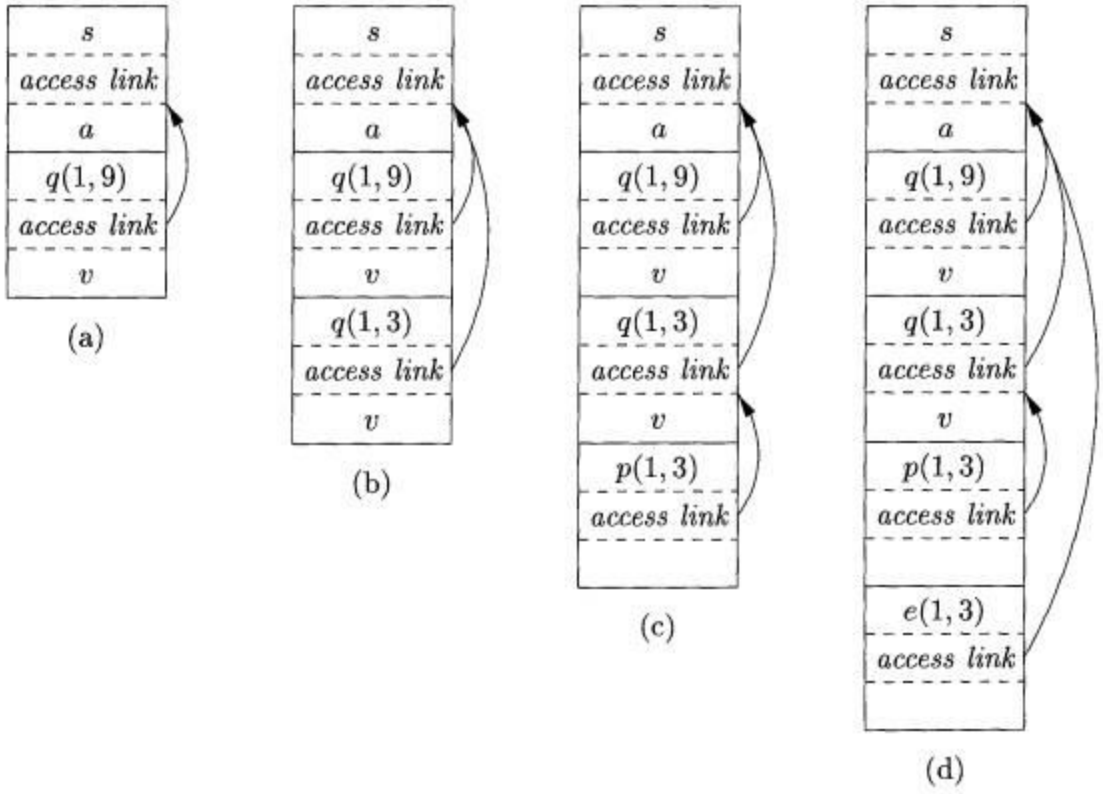

Figure 7.11: Access links for finding nonlocal data

In successive steps of Fig. 7.11 we see a recursive call to *quicksort(l,* 3), followed by a call to *partition,* which calls *exchange.* Notice that *quicksort(l,* 3)'s access link points to *sort,* for the same reason that *quicksort(l,* 9)'s does.

In Fig. 7.11(d), the access link for *exchange* bypasses the activation records for *quicksort* and *partition,* since *exchange* is nested immediately within *sort.* That arrangement is fine, since *exchange* needs to access only the array *a,* and the two elements it must swap are indicated by its own parameters *i* and *j.*

6. Manipulating Access Links

How are access links determined? The simple case occurs when a procedure call is to a particular procedure whose name is given explicitly in the procedure call. The harder case is when the call is to a procedure-parameter; in that case, the particular procedure being called is not known until run time, and the nesting depth of the called procedure may differ in different executions of the call. Thus, let us first consider what should happen when a procedure *q* calls procedure *p,* explicitly. There are three cases:

1. Procedure *p* is at a higher nesting depth than *q*. Then p must be defined immediately within *q,* or the call by *q* would not be at a position that is within the scope of the procedure name *p*. Thus, the nesting depth of *p* is exactly one greater than that of *q,* and the access link from *p* must lead to *q*. It is a simple matter for the calling sequence to include a step that places in the access link for *p* a pointer to the activation record of *q*. Examples include the call of *quicksort* by *sort* to set up Fig. 7.11(a), and the call of *partition* by *quicksort* to create Fig. 7.11(c).

The call is recursive, that is, *p = q2* Then the access link for the new acti-vation record is the same as that of the activation record below it. An ex-

ample is the call of *quicksort(l,* 3) by *quicksort(l,* 9) to set up Fig. 7.11(b).

3. The nesting depth np of p is less than the nesting depth nq of q. In order for the call within q to be in the scope of name p, procedure q must be nested within some procedure r, while p is a procedure defined immediately within r. The top activation record for r can therefore be found by following the chain of access links, starting in the activation record for q, for nq — np + 1 hops. Then, the access link for p must go to this activation of r.

Example 7 . 7: For an example of case (3), notice how we-go from Fig. 7.11(c) to Fig. 7.11(d). The nesting depth 2 of the called function exchange is one less than the depth 3 of the calling function partition. Thus, we start at the activation record for partition and follow 3 - 2 + 1 = 2 access links, which takes us from partition's activation record to that of quicksort(l,S) to that of sort.

The access link for exchange therefore goes to the activation record for sort, as we see in Fig. 7.11(d).

An equivalent way to discover this access link is simply to follow access links for $n_q - n_p$ hops, and copy the access link found in that record. In our example, we would go one hop to the activation record for *quicksort(l,* 3) and copy its access link to *sort.* Notice that this access link is correct for *exchange,* even though *exchange* is not in the scope of *quicksort,* these being sibling functions nested within *sort.*

7. Access Links for Procedure Parameters

When a procedure *p* is passed to another procedure *q* as a parameter, and *q* then calls its parameter (and therefore calls *p* in this activation of *q),* it is possible that *q* does not know the context in which *p* appears in the program. If so, it is impossible for *q* to know how to set the access link for *p*. The solution to this problem is as follows: when procedures are used as

parameters, the caller needs to pass, along with the name of the procedure-parameter, the proper access link for that parameter.

The caller always knows the link, since if *p* is passed by procedure *r* as an actual parameter, then *p* must be a name accessible to r, and therefore, r can determine the access link for *p* exactly as if *p* were being called by *r* directly. That is, we use the rules for constructing access links given in Section 7.3.6.

E x a m p l e **7 . 8:** In Fig. 7.12 we see a sketch of an ML function *a* that has functions *b* and c nested within it. Function *b* has a function-valued parameter f, which it calls. Function c defines within it a function *d,* and c then calls *b* with actual parameter *d.*

```
fun a(x) =
    let
        fun b(f) =
            ··· f ··· ;
        fun c(y) =
            let
                fun d(z) = ···
            in
                ··· b(d) ···
            end
    in
        ··· c(1) ···
    end;
```

Figure 7.12: Sketch of ML program that uses function-parameters

Let us trace what happens when *a* is executed. First, *a* calls c, so we place an activation record for c above that for *a* on the stack. The access link for c points to the record for a, since c is defined immediately within *a.* Then c calls *b(d).* The calling sequence sets up an activation record for *b,* as shown in Fig. 7.13(a).

Within this activation record is the actual parameter *d* and its access link, which together form the value of formal parameter f in the activation record for b. Notice that c knows about d, since d is defined within c, and therefore c passes a pointer to its own activation record as the access link. No matter where d was defined, if c is in the scope of that definition, then one of the three rules of Section 7.3.6 must apply, and c can provide the link.

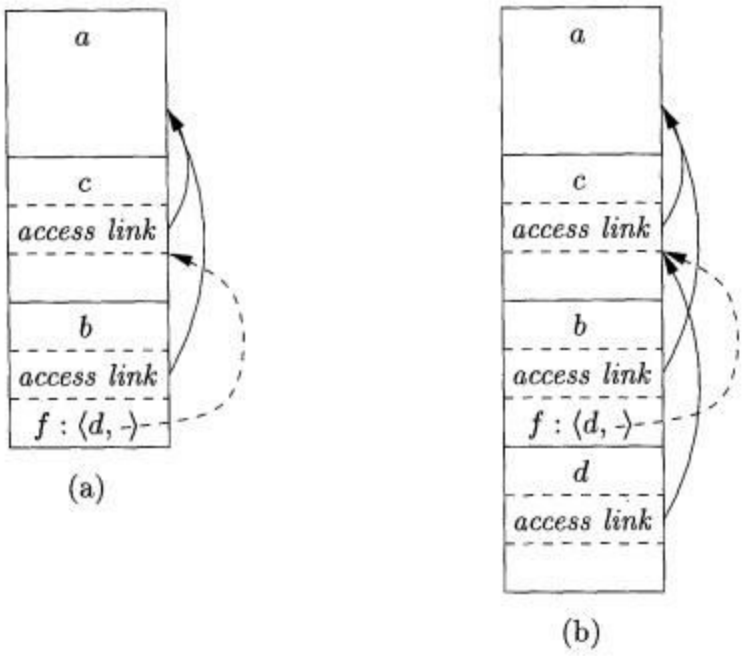

Figure 7.13: Actual parameters carry their access link with them

Now, let us look at what *b* does. We know that at some point, it uses its parameter f, which has the effect of calling *d.* An activation record for *d* appears on the stack, as shown in Fig. 7.13(b). The proper access link to place in this activation record is found in the value for parameter /; the link is to the activation record for c, since c immediately surrounds the definition of *d.* Notice that *b* is capable of setting up the proper link, even though *b* is not in the scope of c's definition. •

8. Displays

One problem with the access-link approach to nonlocal data is that if the nesting depth gets large, we may have to follow long chains of links to reach the data we need. A more efficient implementation uses an auxiliary array *d,* called the *display,* which consists of one pointer for each nesting depth. We arrange that, at all times, *d[i]* is a pointer to the highest activation record on the stack for any procedure at nesting depth *i.* Examples of a display are shown in Fig. 7.14.

For instance, in Fig. 7.14(d), we see the display d, with d[l] holding a pointer to the activation record for sort , the highest (and only) activation record for a function at nesting depth 1. Also, d[2] holds a pointer to the activation record for exchange, the highest record at depth 2, and d[3] points to partition, the highest record at depth 3.

The advantage of using a display is that if procedure *p* is executing, and it needs to access element *x* belonging to some procedure *q,* we need to look only in *d[i],* where *i* is the nesting depth of *q;* we follow the pointer *d[i]* to the activation record for *q,* wherein *x* is found at a known offset. The compiler knows what *i* is, so it can generate code to access *x* using *d[i]* and the offset of

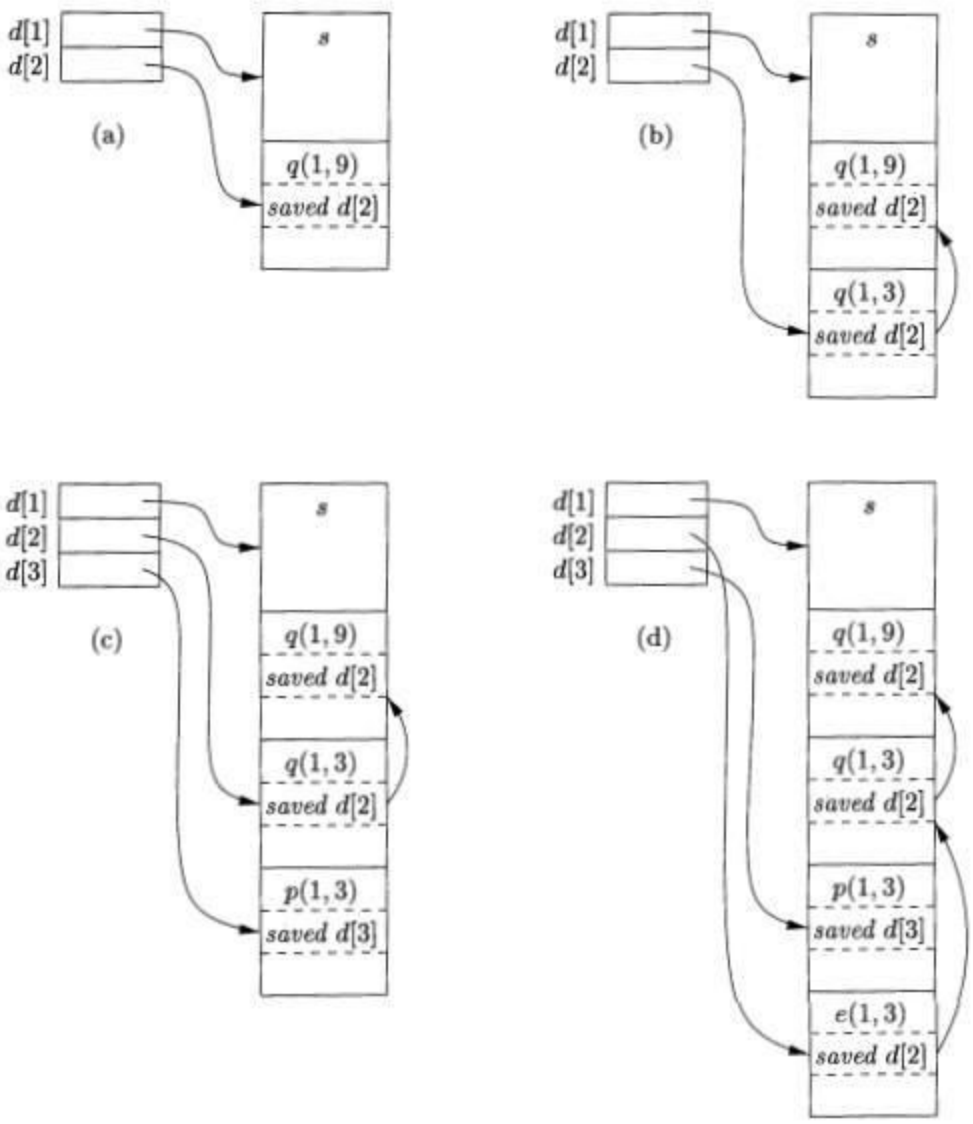

Figure 7.14: Maintaining the display

x from the top of the activation record for *q*. Thus, the code never needs to follow a long chain of access links.

In order to maintain the display correctly, we need to save previous values of display entries in new activation records. If procedure *p* at depth n_p is called, and its activation record is not the first on the stack for a procedure at depth n_p, then the activation record for *p* needs to hold the previous value of $d[n_p]$, while $d[n_p]$ itself is set to point to this activation of *p*. When *p* returns, and its activation record is removed from the stack, we restore $d[n_p]$ to have its value prior to the call of p.

Example 7.9 : Several steps of manipulating the display are illustrated in Fig. 7.14. In Fig. 7.14(a), sort at depth 1 has called quicksort(l, 9) at depth 2.

The activation record for *quicksort* has a place to store the old value of *d[2],* indicated as *saved d[2],* although in this case since there was no prior activation record at depth 2, this pointer is null.

In Fig. 7.14(b), *quicksort(l,* 9) calls *quicksort(l,* 3). Since the activation records for both calls are at depth 2, we must store the pointer to *quicksort(l,* 9), which was in *d[2],* in the record for *quicksort(l,* 3). Then, *d[2]* is made to point to *quicksort(l,* 3).

Next, *partition* is called. This function is at depth 3, so we use the slot *d[3]* in the display for the first time, and make it point to the activation record for *partition.* The record for *partition* has a slot for a former value of *d[3],* but in this case there is none, so the pointer remains null. The display and stack at this time are shown in Fig. 7.14(c).

Then, partition calls exchange. That function is at depth 2, so its activation record stores the old pointer d[2], which goes to the activation record for quicksort(l, 3). Notice that the display pointers "cross"; that is, d[3] points further down the stack than d[2] does. However, that is a proper situation; exchange can only access its own data and that of sort, via d[l].

9. Exercises for Section 7.3

Exercise 7.3.1 : In Fig. 7.15 is a ML function main that computes Fibonacci numbers in a nonstandard way. Function f ibO will compute the nth Fibonacci number for any n > 0. Nested within in is f i b l , which computes the nth Fibonacci number on the assumption n > 2, and nested within f i b l is f ib2, which assumes n > 4. Note that neither f i b l nor f ib2 need to check for the basis cases. Show the stack of activation records that result from a call to main, up until the time that the first call (to f i b O (l)) is about to return. Show the access link in each of the activation records on the stack.

Exercise 7.3.2 : Suppose that we implement the functions of Fig. 7.15 using a display. Show the display at the moment the first call to f i b O (l) is about to return. Also, indicate the saved display entry in each of the activation records on the stack at that time.

```
fun main () {
    let
        fun fib0(n) =
            let
                fun fib1(n) =
                    let
                        fun fib2(n) = fib1(n-1) + fib1(n-2)
                    in
                        if n >= 4 then fib2(n)
                        else fib0(n-1) + fib0(n-2)
                    end
            in
                if n >= 2 then fib1(n)
                else 1
            end
    in
        fib0(4)
    end;
```

Figure 7.15: Nested functions computing Fibonacci numbers

Heap Management

1 The Memory Manager

2 The Memory Hierarchy of a Computer

3 Locality in Programs

4 Reducing Fragmentation

5 Manual Deallocation Requests

6 Exercises for Section 7.4

The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it. While local variables typically become inaccessible when their procedures end, many languages enable us to create objects or other data whose existence is not tied to the procedure activation that creates them. For example, both C + + and Java give the programmer new to create objects that may be passed — or pointers to them may be passed — from procedure to procedure, so they continue to exist long after the procedure that created them is gone. Such objects are stored on a heap.

In this section, we discuss the *memory manager,* the subsystem that allo-cates and deallocates space within the heap; it serves as an interface between application programs and the operating system. For languages like C or C + + that deallocate chunks of storage *manually* (i.e., by explicit statements of the program, such as f r e e or delete) , the memory manager is also responsible for implementing deallocation.

In Section 7.5, we discuss *garbage collection,* which is the process of finding spaces within the heap that are no longer used by the program and can therefore be reallocated to house other data items. For languages like Java, it is the garbage collector that deallocates memory. When it is required, the garbage collector is an important subsystem of the memory manager.

1. The Memory Manager

The memory manager keeps track of all the free space in heap storage at all times. It performs two basic functions:

• *Allocation.* When a program requests memory for a variable or object,[3] the memory manager produces a chunk of contiguous heap memory of the requested size. If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.

• *Deallocation.* The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests. Memory managers typically do not return memory to the operating sys-tem, even if the program's heap usage drops.

Memory management would be simpler if (a) all allocation requests were for chunks of the same size, and (b) storage were released predictably, say, first-allocated first-deallocated. There are some languages, such as Lisp, for which condition (a) holds; pure Lisp uses only one data element — a two- pointer cell — from which all data structures are built. Condition (b) also holds in some situations, the most common being data that can be allocated on the run-time stack. However, in most languages, neither (a) nor (b) holds in general. Rather, data elements of different sizes are allocated, and there is no good way to predict the lifetimes of all allocated objects.

Thus, the memory manager must be prepared to service, in any order, allo-cation and deallocation requests of any size, ranging from one byte to as large as the program's entire address space.

Here are the properties we desire of memory managers:

• Space Efficiency. A memory manager should minimize the total heap space needed by a program. Doing so allows larger programs to run in a fixed virtual address space. Space efficiency is achieved by minimizing "fragmentation," discussed in Section 7.4.4.

• *Program Efficiency.* A memory manager should make good use of the memory subsystem to allow programs to run faster. As we shall see in Section 7.4.2, the time taken to execute an instruction can vary widely depending on where objects are placed in memory. Fortunately, programs tend to exhibit "locality," a phenomenon discussed in Section 7.4.3, which refers to the nonrandom clustered way in which typical programs access memory. By attention to the placement of objects in memory, the memory manager can make better use of space and, hopefully, make the program run faster.

• *Low Overhead.* Because memory allocations and deallocations are fre-quent operations in many programs, it is important that these operations be as efficient as possible. That is, we wish to minimize the *overhead* — the fraction of execution time spent performing allocation and dealloca-tion. Notice that the cost of allocations is dominated by small requests; the overhead of managing large objects is less important, because it usu-ally can be amortized over a larger amount of computation.

2. **The Memory Hierarchy of a Computer**

Memory management and compiler optimization must be done with an aware-ness of how memory behaves. Modern machines are designed so that program-mers can write correct programs without concerning themselves with the details of the memory subsystem. However, the efficiency of a program is determined not just by the number of instructions executed, but also by how long it takes to execute each of these instructions. The time taken to execute an instruction can vary significantly, since the time taken to access different parts of memory can vary from nanoseconds to milliseconds. Data-intensive programs can there-fore benefit significantly from optimizations that make good use of the memory subsystem. As we shall see in Section 7.4.3, they can take advantage of the phenomenon of "locality" — the nonrandom behavior of typical programs.

The large variance in memory access times is due to the fundamental limitation in hardware technology; we can build small and fast storage, or large and slow storage, but not storage that is both large and fast. It is simply impossible today to build gigabytes of storage with nanosecond access times, which is how fast high-performance processors run. Therefore, practically all

modern computers arrange their storage as a memory hierarchy. A memory hierarchy, as shown in Fig. 7.16, consists of a series of storage elements, with the smaller faster ones "closer" to the processor, and the larger slower ones further away.

Typically, a processor has a small number of registers, whose contents are under software control. Next, it has one or more levels of cache, usually made out of static RAM, that are kilobytes to several megabytes in size. The next level of the hierarchy is the physical (main) memory, made out of hundreds of megabytes or gigabytes of dynamic RAM. The physical memory is then backed up by virtual memory, which is implemented by gigabytes of disks. Upon a memory access, the machine first looks for the data in the closest (lowest-level) storage and, if the data is not there, looks in the next higher level, and so on.

Registers are scarce, so register usage is tailored for the specific applications and managed by the code that a compiler generates. All the other levels of the hierarchy are managed automatically; in this way, not only is the programming task simplified, but the same program can work effectively across machines with different memory configurations. With each memory access, the machine searches each level of the memory in succession, starting with the lowest level, until it locates the data. Caches are managed exclusively in hardware, in order to keep up with the relatively fast RAM access times. Because disks are rela

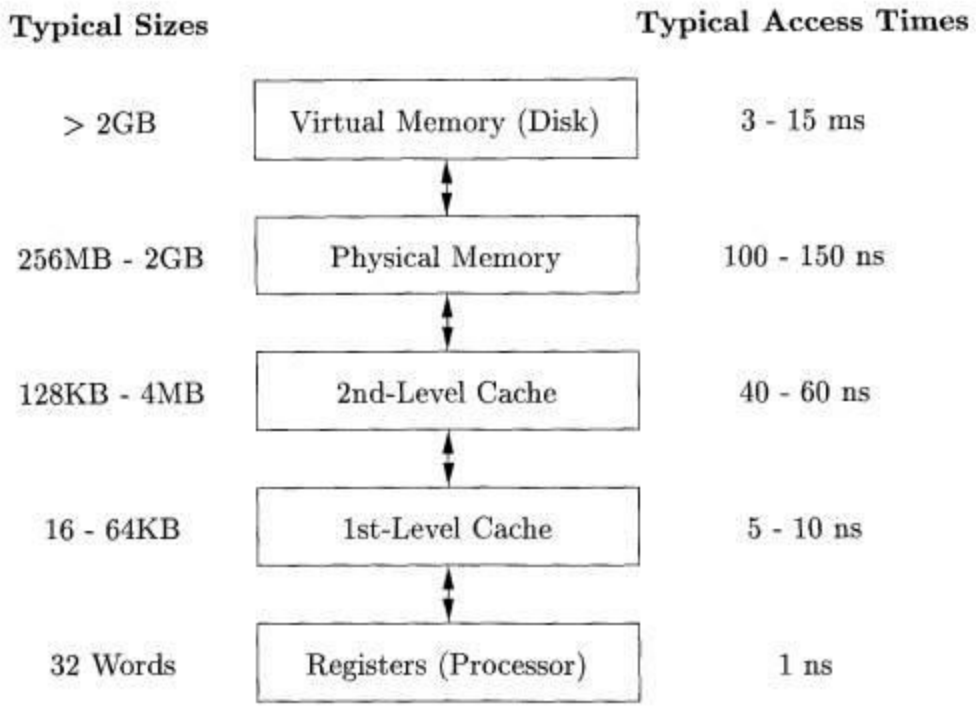

Figure 7.16: Typical Memory Hierarchy Configurations

tively slow, the virtual memory is managed by the operating system, with the assistance of a hardware structure known as the "translation lookaside buffer."

Data is transferred as blocks of contiguous storage. To amortize the cost of access, larger blocks are used with the slower levels of the hierarchy. Be-tween main memory and cache, data is transferred in blocks known as *cache lines,* which are typically from 32 to 256 bytes long. Between virtual memory (disk) and main memory, data is transferred in blocks known as *pages,* typically between 4K and 64K bytes in size.

3. Locality in Programs

Most programs exhibit a high degree of locality; that is, they spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data. We say that a program has temporal locality if the memory locations it accesses are likely to be accessed again within a short period of time. We say that a program has *spatial locality* if memory locations close to the location accessed are likely also to be accessed within a short period of time.

The conventional wisdom is that programs spend 90% of their time executing 10% of the code. Here is why:

Programs often contain many instructions that are never executed. Pro-grams built with components and libraries use only a small fraction of the provided functionality. Also as requirements change and programs evolve, legacy systems often contain many instructions that are no longer used.

Static and Dynamic RAM

Most random-access memory is *dynamic,* which means that it is built of very simple electronic circuits that lose their charge (and thus "forget" the bit they were storing) in a short time. These circuits need to be refreshed — that is, their bits read and rewritten — periodically. On the other hand, static RAM is designed with a more complex circuit for each bit, and consequently the bit stored can stay indefinitely, until it is changed. Evidently, a chip can store more bits if it uses dynamic-RAM circuits than if it uses static-RAM circuits, so we tend to see large main memories of the dynamic variety, while smaller memories, like caches, are made from static circuits.

• Only a small fraction of the code that could be invoked is actually executed in a typical run of the program. For example, instructions to handle illegal inputs and exceptional cases, though critical to the correctness of the program, are seldom invoked on any particular run.

The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.

Locality allows us to take advantage of the memory hierarchy of a modern computer, as shown in Fig. 7.16. By placing the most common instructions and data in the fast-but-small storage, while leaving the rest in the slow-but-large storage, we can lower the average memory-access time of a program significantly.

It has been found that many programs exhibit both temporal and spatial locality in how they access both instructions and data. Data-access patterns, however, generally show a greater variance than instruction-access patterns. Policies such as keeping the most recently used data in the fastest hierarchy work well for common programs but may not work well for some data-intensive programs — ones that cycle through very large arrays, for example.

We often cannot tell, just from looking at the code, which sections of the code will be heavily used, especially for a particular input. Even if we know which instructions are executed heavily, the fastest cache often is not large enough to hold all of them at the same time. We must therefore adjust the contents of the fastest storage dynamically and use it to hold instructions that are likely to be used heavily in the near future.

Optimization Using the Memory Hierarchy

The policy of keeping the most recently used instructions in the cache tends to work well; in other words, the past is generally a good predictor of future memory usage. When a new instruction is executed, there is a high probability that the next instruction also will be executed. This phenomenon is an example of spatial locality. One effective technique to improve the spatial lo-cality of instructions is to have the compiler place basic blocks (sequences of instructions that are always executed sequentially) that are likely to follow each other contiguously — on the same page, or even the same cache line, if possi-ble. Instructions belonging to the same loop or same function also have a high probability of being executed together.[4]

We can also improve the temporal and spatial locality of data accesses in a program by changing the data layout or the order of the computation. For example, programs that visit large amounts of data repeatedly, each time per-forming a small amount of computation, do not perform well. It

is better if we can bring some data from a slow level of the memory hierarchy to a faster level (e.g., disk to main memory) once, and perform all the necessary computations on this data while it resides at the faster level. This concept can be applied recursively to reuse data in physical memory, in the caches and in the registers.

4. Reducing Fragmentation

At the beginning of program execution, the heap is one contiguous unit of free space. As the program allocates and deallocates memory, this space is broken up into free and used chunks of memory, and the free chunks need not reside in a contiguous area of the heap. We refer to the free chunks of memory as *holes.* With each allocation request, the memory manager must *place* the requested chunk of memory into a large-enough hole. Unless a hole of exactly the right size is found, we need to *split* some hole, creating a yet smaller hole.

With each deallocation request, the freed chunks of memory are added back to the pool of free space. We coalesce contiguous holes into larger holes, as the holes can only get smaller otherwise. If we are not careful, the memory may end up getting fragmented, consisting of large numbers of small, noncontiguous holes. It is then possible that no hole is large enough to satisfy a future request, even though there may be sufficient aggregate free space.

Best - Fit and Next - Fit Object Placement

We reduce fragmentation by controlling how the memory manager places new objects in the heap. It has been found empirically that a good strategy for mini-mizing fragmentation for real-life programs is to allocate the requested memory in the smallest available hole that is large enough. This *best-fit* algorithm tends to spare the large holes to satisfy subsequent, larger requests. An alternative, called *first-fit,* where an object is placed in the first (lowest-address) hole in which it fits, takes less time to place objects, but has been found inferior to best-fit in overall performance.

To implement best-fit placement more efficiently, we can separate free space into *bins,* according to their sizes. One practical idea is to have many more bins for the smaller sizes, because there are usually many more small objects. For example, the Lea memory manager, used in the GNU C compiler gcc, aligns all chunks to 8-byte boundaries. There is a bin for every multiple of 8-byte chunks from 16 bytes to 512 bytes. Larger-sized bins are logarithmically spaced (i.e., the minimum size for each bin is twice that of the previous bin), and within each of these bins the chunks are ordered by their size. There is always a chunk of free space that can be extended by requesting more pages from the operating system. Called

the *wilderness chunk,* this chunk is treated by Lea as the largest-sized bin because of its extensibility.

Binning makes it easy to find the best-fit chunk.

If, as for small sizes requested from the Lea memory manager, there is a bin for chunks of that size only, we may take any chunk from that bin.

For sizes that do not have a private bin, we find the one bin that is allowed to include chunks of the desired size. Within that bin, we can use

either a first-fit or a best-fit strategy; i.e., we either look for and select the first chunk that is sufficiently large or, we spend more time and find the smallest chunk that is sufficiently large. Note that when the fit is not exact, the remainder of the chunk will generally need to be placed in a bin with smaller sizes.

• However, it may be that the target bin is empty, or all chunks in that bin are too small to satisfy the request for space. In that case, we simply repeat the search, using the bin for the next larger size(s). Eventually, we either find a chunk we can use, or we reach the "wilderness" chunk, from which we can surely obtain the needed space, possibly by going to the operating system and getting additional pages for the heap.

While best-fit placement tends to improve space utilization, it may not be the best in terms of spatial locality. Chunks allocated at about the same time by a program tend to have similar reference patterns and to have similar lifetimes. Placing them close together thus improves the program's spatial locality. One useful adaptation of the best-fit algorithm is to modify the placement in the case when a chunk of the exact requested size cannot be found. In this case, we use a *next-fit* strategy, trying to allocate the object in the chunk that has last been split, whenever enough space for the new object remains in that chunk. Next-fit also tends to improve the speed of the allocation operation.

M a n a g i n g and Coalescing Free Space

When an object is deallocated manually, the memory manager must make its chunk free, so it can be allocated again. In some circumstances, it may also be possible to combine *(coalesce)* that chunk with adjacent chunks of the heap, to form a larger chunk. There is an advantage to doing so, since we can always use a large chunk to do the work of small chunks of equal total size, but many small chunks cannot hold one large object, as the combined chunk could.

If we keep a bin for chunks of one fixed size, as Lea does for small sizes, then we may prefer not to coalesce adjacent blocks of that size into a chunk of double the size. It is simpler to keep all the chunks of one size in as many pages as we need, and never coalesce them. Then, a simple allocation/deallocation scheme is to keep a bitmap, with one bit for each chunk in the bin. A 1 indicates the chunk is occupied; 0 indicates it is free. When a chunk is deallocated, we change its 1 to a 0. When we need to allocate a chunk, we find any chunk with a 0 bit, change that bit to a 1, and use the corresponding chunk. If there are no free chunks, we get a new page, divide it into chunks of the appropriate size, and extend the bit vector.

Matters are more complex when the heap is managed as a whole, without binning, or if we are willing to coalesce adjacent chunks and move the resulting chunk to a different bin if necessary. There are two data structures that are useful to support coalescing of adjacent free blocks:

• Boundary Tags. At both the low and high ends of each chunk, whether free or allocated, we keep vital information. At both ends, we keep a free/used bit that tells whether or not the block is currently allocated (used) or available (free). Adjacent to each free/used bit is a count of the total number of bytes in the chunk.

• A Doubly Linked, Embedded Free List. The free chunks (but not the allocated chunks) are also linked in a doubly linked list. The pointers for this list are within the blocks themselves, say adjacent to the boundary tags at either end. Thus, no additional space is needed for the free list, although its existence does place a lower bound on how small chunks can get; they must accommodate two boundary tags and two pointers, even if the object is a single byte. The order of chunks on the free list is left unspecified. For example, the list could be sorted by size, thus facilitating best-fit placement.

Example 7 . 1 0 : Figure 7.17 shows part of a heap with three adjacent chunks, A, B, and C. Chunk B: of size 100, has just been deallocated and returned to the free list. Since we know the beginning (left end) of 5, we also know the end of the chunk that happens to be immediately to *B's* left, namely *A* in this example. The free/used bit at the right end of *A* is currently 0, so *A* too is free. We may therefore coalesce *A* and *B* into one chunk of 300 bytes.

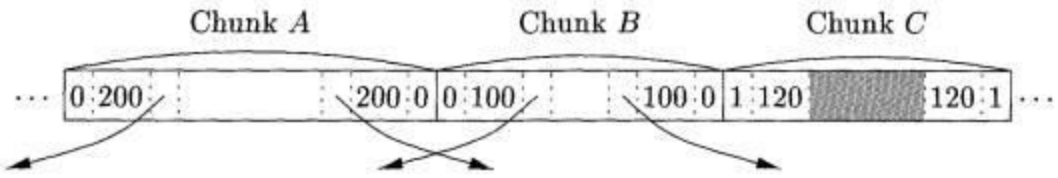

Figure 7.17: Part of a heap and a doubly linked free list

It might be the case that chunk C, the chunk immediately to B's right, is also free, in which case we can combine all of A, B, and C. Note that if we always coalesce chunks when we can,

then there can never be two adjacent free chunks, so we never have to look further than the two chunks adjacent to the one being deallocated. In the current case, we find the beginning of *C* by starting at the left end of *B,* which we know, and finding the total number of bytes in *B,* which is found in the left boundary tag of *B* and is 100 bytes. With this information, we find the right end of *B* and the beginning of the chunk to its right. At that point, we examine the free/used bit of *C* and find that it is 1 for used; hence, *C* is not available for coalescing.

Since we must coalesce *A* and *B,* we need to remove one of them from the free list. The doubly linked free-list structure lets us find the chunks before and after each of *A* and *B*. Notice that it should not be assumed that physical neighbors *A* and *B* are also adjacent on the free list. Knowing the chunks preceding and following *A* and *B* on the free list, it is straightforward to manipulate pointers on the list to replace *A* and *B* by one coalesced chunk. •

Automatic garbage collection can eliminate fragmentation altogether if it moves all the allocated objects to contiguous storage. The interaction between garbage collection and memory management is discussed in more detail in Sec-tion 7.6.4.

5. Manual Deallocation Requests

We close this section with manual memory management, where the programmer must explicitly arrange for the deallocation of data, as in C and C + + . Ideally, any storage that will no longer be accessed should be deleted. Conversely, any storage that may be referenced must not be deleted. Unfortunately, it is hard to enforce either of these properties. In addition to considering the difficulties with manual deallocation, we shall describe some of the techniques programmers use to help with the difficulties.

Problems with Manual Deallocation

Manual memory management is error-prone. The common mistakes take two forms: failing ever to delete data that cannot be referenced is called a *memory-leak* error, and referencing deleted data is a *dangling-pointer-dereference* error.

It is hard for programmers to tell if a program *will never* refer to some stor-age in the future, so the first common mistake is not deleting storage that will never be referenced. Note that although memory leaks may slow down the exe-cution of a program due to increased memory usage, they do not affect program correctness, as long as the machine does not run out of memory. Many

pro-grams can tolerate memory leaks, especially if the leakage is slow. However, for long-running programs, and especially nonstop programs like operating systems or server code, it is critical that they not have leaks.

Automatic garbage collection gets rid of memory leaks by deallocating all the garbage. Even with automatic garbage collection, a program may still use more memory than necessary. A programmer may know that an object will never be referenced, even though references to that object exist somewhere. In that case, the programmer must deliberately remove references to objects that will never be referenced, so the objects can be deallocated automatically.

Being overly zealous about deleting objects can lead to even worse problems than memory leaks. The second common mistake is to delete some storage and then try to refer to the data in the deallocated storage. Pointers to storage that has been deallocated are known as dangling pointers. Once the freed storage has been reallocated to a new variable, any read, write, or deallocation via the dangling pointer can produce seemingly random effects. We refer to any operation, such as read, write, or deallocate, that follows a pointer and tries to use the object it points to, as *dereferencing* the pointer.

Notice that reading through a dangling pointer may return an arbitrary value. Writing through a dangling pointer arbitrarily changes the value of the new variable. Deallocating a dangling pointer's storage means that the storage of the new variable may be allocated to yet another variable, and actions on the old and new variables may conflict with each other.

Unlike memory leaks, dereferencing a dangling pointer after the freed storage is reallocated almost always creates a program error that is hard to debug. As a result, programmers are more inclined not to deallocate a variable if they are not certain it is unreferencable.

A related form of programming error is to access an illegal address. Common examples of such errors include dereferencing null pointers and accessing an out-of-bounds array element. It is better for such errors to be detected than to have the program silently corrupt the results. In fact, many security violations exploit programming errors of this type, where certain program inputs allow unintended access to data, leading to a "hacker" taking control of the program and machine. One antidote is to have the compiler insert checks with every access, to make sure it is within bounds. The compiler's optimizer can discover and remove those checks that are not really necessary because the optimizer can deduce that the access must be within bounds.

An Example: Purify

Rational's Purify is one of the most popular commercial tools that helps programmers find memory access errors and memory leaks in programs. Purify instruments binary code by adding additional instructions to check for errors as the program executes. It keeps a map of memory to indicate where all the freed and used spaces are. Each allocated object is bracketed with extra space; accesses to unallocated locations or to spaces between objects are flagged as errors. This approach finds some dangling pointer references, but not when the memory has been reallocated and a valid object is sitting in its place. This approach also finds some out-of-bound array accesses, if they happen to land in the space inserted at the end of the objects.

Purify also finds memory leaks at the end of a program execution. It searches the contents of all the allocated objects for possible pointer values. Any object without a pointer to it is a leaked chunk of memory. Purify reports the amount of memory leaked and the locations of the leaked objects. We may compare Purify to a "conservative garbage collector," which will be discussed in Section 7.8.3. and machine. One antidote is to have the compiler insert checks with every access, to make sure it is within bounds. The compiler's optimizer can discover and remove those checks that are not really necessary because the optimizer can deduce that the access must be within bounds.

Programming Conventions and Tools

We now present a few of the most popular conventions and tools that have been developed to help programmers cope with the complexity in managing memory:

• *Object ownership* is useful when an object's lifetime can be statically rea-soned about. The idea is to associate an *owner* with each object at all times. The owner is a pointer to that object, presumably belonging to some function invocation. The owner (i.e., its function) is responsible for either deleting the object or for passing the object to another owner. It is possible to have other, nonowning pointers to the same object; these pointers can be overwritten any time, and no deletes should ever be ap-plied through them. This convention eliminates memory leaks, as well as attempts to delete the same object twice. However, it does not help solve the dangling-pointer-reference problem, because it is possible to follow a nonowning pointer to an object that has been deleted.

Reference counting is useful when an object's lifetime needs to be deter-mined dynamically. The idea is to associate a count with each dynamically allocated object. Whenever a reference to the object is created, we incre-ment the reference count; whenever a reference is removed, we decrement the reference count. When the count goes to zero, the object can no longer be referenced and can therefore be deleted. This technique, however, does not catch useless, circular data structures, where a collection of objects cannot be accessed, but their reference counts are not zero, since they refer to each other. For an illustration of this problem, see Example 7.11. Reference counting does eradicate all dangling-pointer references, since there are no outstanding

references to any deleted objects. Reference counting is expensive because it imposes an overhead on every operation that stores a pointer.

• *Region-based allocation* is useful for collections of objects whose lifetimes are tied to specific phases in a computation.When objects are created to be used only within some step of a computation, we can allocate all such objects in the same region. We then delete the entire region once that computation step completes. This *region-based allocation* technique has limited applicability. However, it is very efficient whenever it can be used; instead of deallocating objects one at a time, it deletes all objects in the region in a wholesale fashion.

6. Exercises for Section 7.4

Exercise 7 **. 4** . 1 : Suppose the heap consists of seven chunks, starting at address 0. The sizes of the chunks, in order, are 80, 30, 60, 50, 70, 20, 40 bytes. When we place an object in a chunk, we put it at the high end if there is enough space remaining to form a smaller chunk (so that the smaller chunk can easily remain on the linked list of free space). However, we cannot tolerate chunks of fewer that 8 bytes, so if an object is almost as large as the selected chunk, we give it the entire chunk and place the object at the low end of the chunk.

If we request space for objects of the following sizes: 32, 64, 48, 16, in that order, what does the free space list look like after satisfying the requests, if the method of selecting chunks is

First fit.

Best fit.