Chapter 5
Memory Management

An important component of an abstract machine's interpreter is the one dealing with memory management. If this component can be extremely simple in a physical machine, memory management in an abstract machine for a high-level language is fairly complicated and can employ a range of techniques. We will see both static and dynamic management and will examine activation records, the system stack and the heap. One section in particular is dedicated to the data structures and mechanisms used to implement scope rules.

Conceptually, garbage-collection techniques, techniques for the automatic recovery of memory allocated in a heap, are included in memory management. However, to make the presentation more coherent, these techniques will be explained in Sect. 8.12, after having dealt with data types and pointers.

5.1 Techniques for Memory Management

As we said in Chap. 1, memory management is one of the functions of the interpreter associated with an abstract machine. This functionality manages the allocation of memory for programs and for data, that is determines how they must be arranged in memory, how much time they may remain and which auxiliary structures are required to fetch information from memory.

In the case of a low-level abstract machine, the hardware, for example, memory management is very simple and can be entirely *static*. Before execution of the program begins, machine language program and its associated data is loaded into an appropriate area of memory, where it remains until its execution ends.

In the case of a high-level language, matters are, for various reasons, more complicated. First of all, if the language permits recursion, static allocation is insufficient.[1] In fact, while we can statically establish the maximum number of active procedures at any point during execution in the case of languages without recursion,

[1] We will see below an exception to this general principle. This is the case of so-called tail recursion.

DOI 10.1007/978-1-84882-914-5_5, © Springer-Verlag London Limited 2010

when we have recursive procedures this is no longer true because the number of simultaneously active procedure calls can depend on the parameters of the procedures or, generally, on information available only at runtime.

Example 5.1 Consider the following fragment:

```
int fib (int n) {
   if (n == 0) return 1;
   else if (n == 1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

which, if called with an argument n, computes (in a very inefficient way) the value of the nth Fibonacci number. Let us recall that Fibonacci numbers are the terms of the sequence[2] defined inductively as follows: $Fib(0) = Fib(1) = 1$; $Fib(n) = Fib(n-1) + Fib(n-2)$, for $n > 1$. It is clear that the number of active calls to `Fib` depends, other than on the point of execution, on the value of the argument, n. Using a simple recurrence relation, it can be verified that the number, $C(n)$, of calls to `Fib` necessary to calculate the value of the term $Fib(n)$ (and, therefore, the simultaneously active calls) is exactly equal to this value. From a simple inspection of the code, indeed, it can be seen that $C(n) = 1$ for $n = 0$ and $n = 1$, while $C(n) = C(n-1) + C(n-2)$ for $n > 1$. It is known that the Fibonacci numbers grow exponentially, so the number of calls to `fib` is of the order of $O(2^n)$.

Given that every procedure call requires its own memory space to store parameters, intermediate results, return addresses, and so on, in the presence of recursive procedures, static allocation of memory is no longer sufficient and we have to allow *dynamic* memory allocation and deallocation operations, which are performed during the execution of the program. Such dynamic memory processing can be implemented in a natural fashion using a *stack* for procedure (or in-line block) activations since they follow a LIFO (Last In First Out) policy—the last procedure called (or the last block entered) will be the first to be exited.

There are, however, other cases which require dynamic memory management for which a stack is not enough. These are cases in which the language allows explicit memory allocation and deallocation operations, as happens, for example, in C with the `malloc` and `free` commands. In these cases, given that the allocation operation (`malloc`) and the one for deallocation (`free`) can be alternated in any order whatsoever, it is not possible to use a stack to manage the memory and, as we will better see as the chapter unfolds, a particular memory structure called a *heap* is used.

[2] The sequence takes the name of the Pisan mathematician of the same name, known also as Leonardo da Pisa (ca., 1175–1250), who seems to have encountered the sequence by studying the increase in a population of rabbits. For information on inductive definition, see the box on page 153.

5.2 Static Memory Management

Static memory management is performed by the complier before execution starts. Statically allocated memory objects reside in a fixed zone of memory (which is determined by the compiler) and they remain there for the entire duration of the program's execution. Typical elements for which it is possible statically to allocate memory are *global variables*. These indeed can be stored in a memory area that is fixed before execution begins because they are visible throughout the program. The *object code instructions* produced by the compiler can be considered another kind of static object, given that normally they do not change during the execution of the program, so in this case also memory will be allocated by the compiler. *Constants* are other elements that can be handled statically (in the case in which their values do not depend on other values which are unknown at compile time). Finally, various *compiler-generated tables*, necessary for the runtime support of the language (for example, for handling names, for type checking, for garbage collection) are stored in reserved areas allocated by the compiler.

In the case in which the language does not support recursion, as we have anticipated, it is possible statically to handle the memory for other components of the language. Substantially, this is done by statically associating an area of memory in which is stored the information local to the procedure with each procedure (or subroutine)[3] itself. This information is composed of local variables, possible parameters of the procedure (containing both arguments and results), the return address (or the address to which control must pass when the procedure terminates), possible temporary values used in complex calculations and various pieces of "bookkeeping" information (saved register values, information for debugging and so on).

The situation of a language with only static memory allocation is shown in Fig. 5.1.

It will be noted that successive calls to the same procedure share the same memory areas. This is correct because, in the absence of recursion, there cannot be two different calls to the same procedure that are active at the same time.

5.3 Dynamic Memory Management Using Stacks

Most modern programming languages allow block structuring of programs.[4]

Blocks, whether in-line or associated with procedures, are entered and left using the LIFO scheme. When a block A is entered, and then a block B is entered, before leaving A, it is necessary to leave B. It is therefore natural to manage the memory

[3]It would be more correct to speak of subroutines because this was the term used in languages that used static memory allocation, such as, for example, the first versions of FORTRAN from the 1960s and 70s.

[4]We will see below that important languages, such as C, though, do not offer the full potential of this mechanism, in that they do not permit the declaration of local procedures and functions in nested blocks.

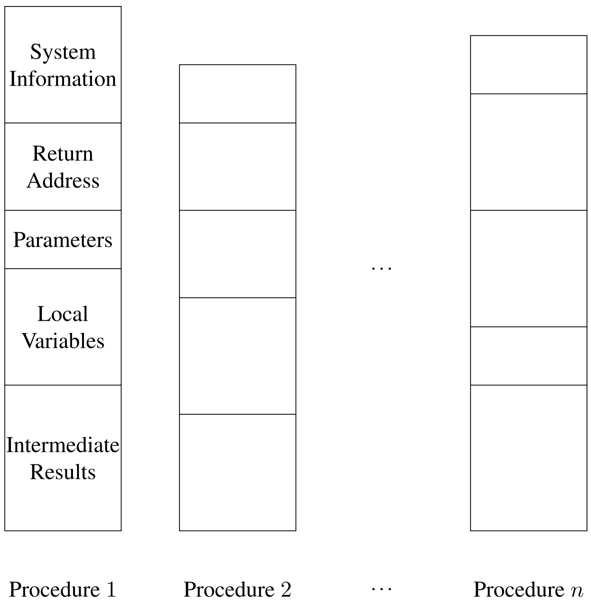


Fig. 5.1 Static memory management

space required to store the information local to each block using a stack. We will see an example.

Example 5.2 Let us consider the following program:

```
A:{int a = 1;
   int b = 0;

   B:{int c = 3;
      int b = 3;
     }
   b=a+1;
  }
```

At runtime, when block A is entered, a push operation allocates a space large enough to hold the variables `a` and `b`, as shown in Fig. 5.2. When block B is entered, we have to allocate a new space on the stack for the variables `c` and `b` (recall that the inner variable `b` is different from the outer one) and therefore the situation, after this second allocation, is that shown in Fig. 5.3. When block B exits, on the other hand, it is necessary to perform a pop operation to deallocate the space that had been reserved for the block from the stack. The situation after such a deallocation and after the assignment is shown in Fig. 5.4. Analogously, when block A exits, it will be necessary to perform another pop to deallocate the space for A as well.

The case of procedures is analogous and we consider it in Sect. 5.3.2.

The memory space, allocated on the stack, dedicated to an in-line block or to an activation of a procedure is called the *activation record*, or *frame*. Note that an acti-

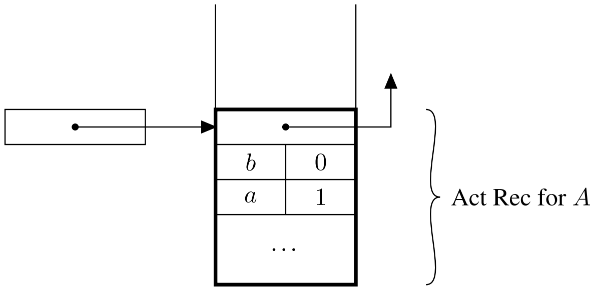


Fig. 5.2 Allocation of an activation record for block A in Example 5.2

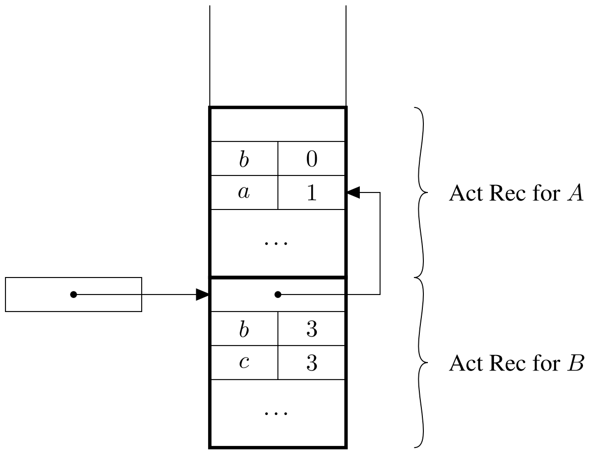


Fig. 5.3 Allocation of activation records for blocks A and B in Example 5.2

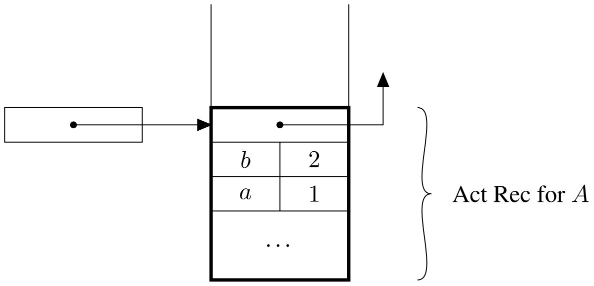


Fig. 5.4 Organisation after the execution of the assignment in Example 5.2

vation record is associated with a specific activation of a procedure (one is created when the procedure is called) and not with the declaration of a procedure. The values that must be stored in an activation record (local variables, temporary variables, etc.) are indeed different for the different calls on the same procedure.

The stack on which activation records are stored is called the *runtime* (*or system*) *stack*.

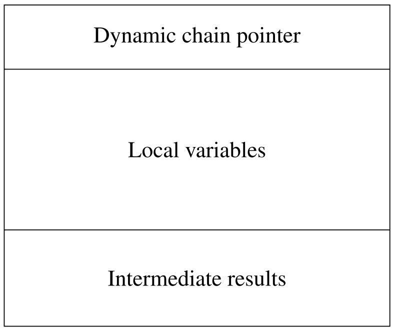


Fig. 5.5 Organisation of an activation record for an in-line block

It should finally be noted, that to improve the use of runtime memory, dynamic memory management is sometimes also used to implement languages that do not support recursion. If the average number of simultaneously active calls to the same procedure is less than the number of procedures declared in the program, using a stack will save space, for there will be no need to allocate a memory area for each declared procedure, as must be done in the case of entirely static management.

5.3.1 Activation Records for In-line Blocks

The structure of a generic activation record for an in-line block is shown in Fig. 5.5. The various sectors of the activation record contain the following information:

Intermediate results When calculations must be performed, it can be necessary to store intermediate results, even if the programmer does not assign an explicit name to them. For example, the activation record for the block:

```
{int a =3;
 b= (a+x)/ (x+y);}
```

could take the form shown in Fig. 5.6, where the intermediate results `(a+x)` and `(x+y)` are explicitly stored before the division is performed. The need to store intermediate results on the stack depends on the compiler being used and on the architecture to which one is compiling. On many architectures they can be stored in registers.

Local variables Local variables which are declared inside blocks, must be stored in a memory space whose size will depend on the number and type of the variables. This information in general is recorded by the compiler which therefore will be able to determine the size of this part of the activation record. In some cases, however, there can be declarations which depend on values recorded only at runtime (this is, for example, the case for dynamic arrays, which are present in some languages, whose dimensions depend on variables which are only instantiated at execution time). In these cases, the activation record also contains a variable-length

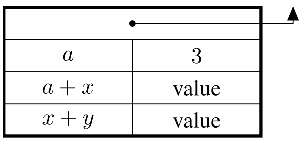


Fig. 5.6 An activation record with space for intermediate results

part which is defined at runtime. We will examine this in detail in Chap. 8 when discussing arrays.

Dynamic chain pointer This field stores a pointer to the previous activation record on the stack (or to the last activation record created). This information is necessary because, in general, activation records have different sizes. Some authors call this pointer the *dynamic link* or *control link*. The set of links implemented by these pointers is called the *dynamic chain*.

5.3.2 Activation Records for Procedures

The case of procedures and functions[5] is analogous to that of in-line blocks but with some additional complications due to the fact that, when a procedure is activated, it is necessary to store a greater amount of information to manage correctly the control flow. The structure of a generic activation record for a procedure is shown in Fig. 5.7. Recall that a function, unlike a procedure, returns a value to the caller when it terminates its execution. Activation records for the two cases are therefore identical with the exception that, for functions, the activation record must also keep tabs on the memory location in which the function stores its return value.

Let us now look in detail at the various fields of an activation record:

Intermediate results, local variables, dynamic chain pointer The same as for in-line blocks.

Static chain pointer This stores the information needed to implement the static scope rules described in Sect. 5.5.1.

Return address Contains the address of the first instruction to execute after the call to the current procedure/function has terminated execution.

Returned result Present only in functions. Contains the address of the memory location where the subprogram stores the value to be returned by the function when it terminates. This memory location is inside the caller's activation record.

Parameters The values of actual parameters used to call the procedure or function are stored here.

The organisation of the different fields of the activation record varies from implementation to implementation. The dynamic chain pointer and, in general, every

[5]Here and below, we will almost always use the terms "function" and "procedure" as synonyms. Although there are is no agreement between authors, the term "procedure" should denote a subprogram which does not directly return a value, while a function is a subprogram that returns one.

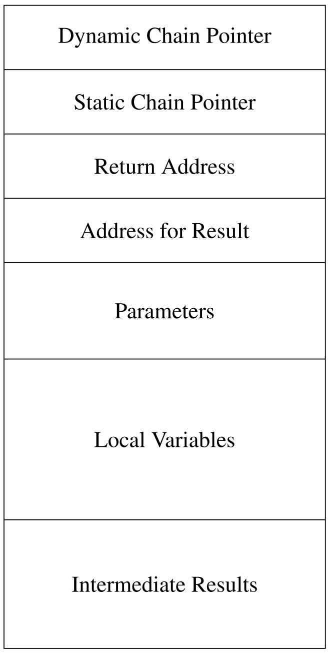


Fig. 5.7 Structure of the activation record for a procedure

pointer to an activation record, points to a fixed (usually central) area of the activation record. The addresses of the different fields are obtained, therefore, by adding a negative or positive offset to the value of the pointer.

Variable names are not normally stored in activation records and the compiler substitutes references to local variables for addresses relative to a fixed position in (i.e., an offset into) the activation record for the block in which the variables are declared. This is possible because the position of a declaration inside a block is fixed statically and the compiler can therefore associate every local variable with an exact position inside the activation record.

In the case of references to non-local variables, also, as we will see when we discuss scope rules, it is possible to use mechanisms that avoid storing names and therefore avoid having to perform a runtime name-based search through the activation-record stack in order to resolve a reference.

Finally, modern compilers often optimise the code they produce and save some information in registers instead of in the activation record. For simplicity, in this book, we will not consider these optimisations. In any case for greater clarity, in the examples, we will assume that variable names are stored in activation records.

To conclude, let us note that all the observations that we have made about variable names, their accessibility and storage in activation records, can be extended to other kinds of denotable object.

5.3.3 Stack Management

Figure 5.8 shows the structure of a system stack which we assume growing downwards (the direction of stack growth varies according to the implementation). As shown in the figure, an external pointer to the stack points to the last activation record on the stack (pointing to a predetermined area of the activation record which is used as a base for calculating the offsets used to access local names). This pointer, which we call the activation record pointer, is also called the frame or current environment pointer (because environments are implemented using activation records). In the figure, we have also indicated where the first free location is. This second pointer, used in some implementations, can, in principle, also be omitted if the activation-record pointer always points to a position that is at a pre-defined distance from the start of the free area on the stack.

Activation records are stored on and removed from the stack at runtime. When a block is entered or a procedure is called, the associated activation record is pushed onto the stack; it is later removed from the stack when the block is exited or when the procedure terminates.

The runtime management of the system stack is implemented by code fragments which the compiler (or interpreter) inserts immediately before and after the call to a procedure or before the start and after the end of a block.

Let us examine in detail what happens in the case of procedures, given that the case of in-line blocks is only a simplification.

First of all, let us clarify the terminology that we are using. We use "caller" and "callee" to indicate, respectively, the program or procedure that performs a call (of a procedure) and the procedure that has been called.

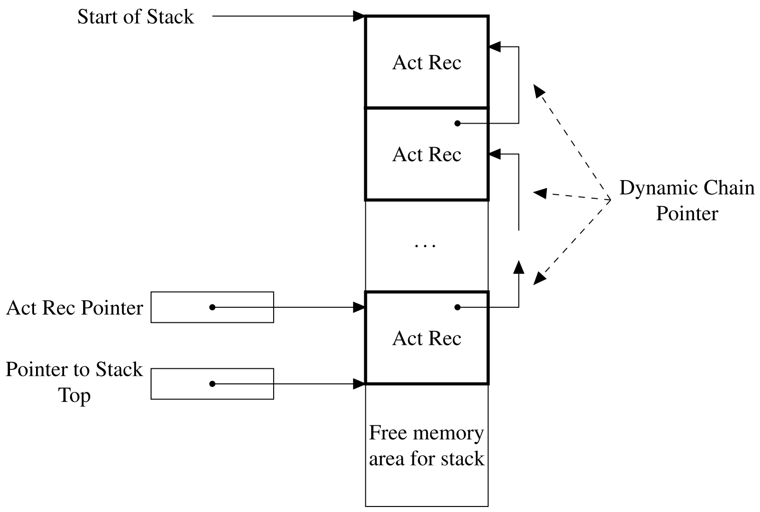


Fig. 5.8 The stack of activation records

Stack management is performed both by the caller and by the callee. To do this, as well as handling other control information, a piece of code called the *calling sequence* is inserted into the caller to be executed, in part, immediately before the procedure call. The remainder of this code is executed immediately after the call. In addition, in the callee two pieces of code are added: a *prologue*, to be executed immediately after the call, and an *epilogue*, which is executed when the procedure ends execution. These three code fragments manage the different operations needed to handle activation records and correctly implement a procedure call. The exact division of what the caller and callee do depends, as usual, on the compiler and on the specific implementation under consideration. Moreover, to optimise the size of code produced, it is preferable that the larger part of the activity is given to the callee, since the code is added only once (to the code associated with the declaration of the call) instead of many times (to the code associated with different calls). Without therefore further specifying the division of activities, they consist of the following tasks:

Modification of program counter This is clearly necessary to pass control to the called procedure. The old value (incremented) must be saved to maintain the return address.

Allocation of stack space The space for the new activation record must be pre-allocated and therefore the pointer to the first free location on the stack must be updated as a consequence.

Modification of activation record pointer The pointer must point to the new activation record for the called procedure; the activation record will have been pushed onto the stack.

Parameter passing This activity is usually performed by the caller, given that different calls of the same procedure can have different parameters.

Register save Values for control processing, typically stored in registers, must be saved. This is the case, for example, with the old activation record pointer which is saved as a pointer in the dynamic chain.

Execution of initialisation code Some languages require explicit constructs to initialise some items stored in the new activation record.

When control *returns to the calling program*, i.e. when the called procedure terminates, the *epilogue* (in the called routine) and the *calling sequence* (in the caller) must perform the following operations:

Update of program counter This is necessary to return control to the caller.

Value return The values of parameters which pass information from the caller to the called procedure, or the value calculated by the function, must be stored in appropriate locations usually present in the caller's activation record and accessible to the activation record of the called procedure.

Return of registers The value of previously saved registers must be restored. In particular, the old value of the activation record pointer must be restored.

Execution of finalisation code Some languages require the execution of appropriate finalisation code before any local objects can be destroyed.

Deallocation of stack space The activation record of the procedure which has terminated must be removed from the stack. The pointer to (the first free position on) the stack must be modified as a result.

It should be noted that in the above description, we have omitted the handling of the data structures necessary for the implementation of scope rules. This will be considered in detail in Sect. 5.5 of this chapter.

5.4 Dynamic Management Using a Heap

In the case in which the language includes explicit commands for memory allocation, as for example do C and Pascal, management using just the stack is insufficient. Consider for example the following C fragment:

```
int *p, *q; /* p,q  NULL pointers to integers */
p =  malloc (sizeof (int));
        /* allocates the memory pointed to by p */
q =  malloc (sizeof (int));
        /* allocates the memory pointed to by  q */
*p = 0;     /* dereferences and assigns */
*q = 1;     /* dereferences and assigns */
free(p);    /* deallocates the memory pointed to by p */
free(q);    /* deallocates the memory pointed to by q */
```

Given that the memory deallocation operations are performed in the same order as allocations (first p, then q), the memory cannot be allocated in LIFO order.

To manage explicit memory allocations, which can happen at any time, a particular area of memory, called a *heap*, is used. Note that this term is used in computing also to mean a particular type of data structure which is representable using a binary tree or a vector, used to implement efficiently priorities (and used also in the "heap sort" sorting algorithm, where the term "heap" was originally introduced). The definition of heap that we use here has nothing to do with this data structure. In the programming language jargon, a heap is simply an area of memory in which blocks of memory can be allocated and deallocated relatively freely.

Heap management methods fall into two main categories according to whether the memory blocks are considered to be of *fixed* or *variable* length.

5.4.1 Fixed-Length Blocks

In this case, the heap is divided into a certain number of elements, or blocks, of fairly small fixed length, linked into a list structure called the *free list*, as shown in Fig. 5.9. At runtime, when an operation requires the allocation of a memory block from the heap (for example using the malloc command), the first element of the free list is removed from the list, the pointer to this element is returned to the operation that

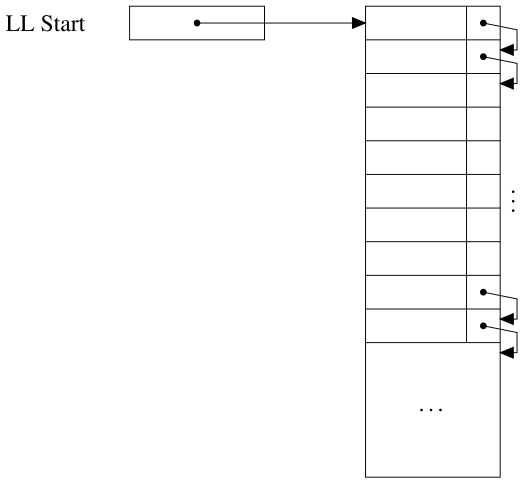


Fig. 5.9 Free list in a heap with fixed-size blocks

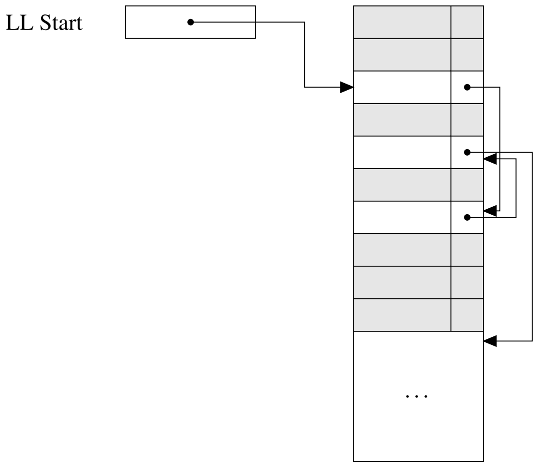


Fig. 5.10 Free list for heap of fixed-size blocks after allocation of some memory. *Grey blocks* are allocated (in use)

requested the memory and the pointer to the free list is updated so that it points to the next element.

When memory is, on the other hand, freed or deallocated (for example using `free`), the freed block is linked again to the head of the free list. The situation after some memory allocations is shown in Fig. 5.10. Conceptually, therefore, management of a heap with fixed-size blocks is simple, provided that it is known how to identify and reclaim the memory that must be returned to the free list easily. These operations of identification and recovery are not obvious, as we will see below.

5.4.2 *Variable-Length Blocks*

In the case in which the language allows the runtime allocation of variable-length memory spaces, for example to store an array of variable dimension, fixed-length blocks are no longer adequate. In fact the memory to be allocated can have a size greater than the fixed block size, and the storage of an array requires a contiguous region of memory that cannot be allocated as a series of blocks. In such cases, a heap-based management scheme using variable-length blocks is used.

This second type of management uses different techniques, mainly defined with the aim of increasing memory occupation and execution speed for heap management operations (recall that they are performed at runtime and therefore impact on the execution time of the program). As usual, these two characteristics are difficult to reconcile and good implementations tend towards a rational compromise.

In particular, as far as memory occupancy is concerned, it is a goal to avoid the phenomenon of memory *fragmentation*. So-called *internal fragmentation* occurs when a block of size strictly larger than the requested by the program is allocated. The portion of unused memory internal to the block clearly will be wasted until the block is returned to the free list. But this is not the most serious problem. Indeed, so-called *external fragmentation* is worse. This occurs when the free list is composed of blocks of a relatively small size and for which, even if the sum of the total available free memory is enough, the free memory cannot be effectively used. Figure 5.11 shows an example of this problem. If we have blocks of size x and y (words or some other unit—it has no relevance here) on the free list and we request the allocation of a block of greater size, our request cannot be satisfied despite the fact that the total amount of free memory is greater than the amount of memory that has been requested. The memory allocation techniques tend therefore to

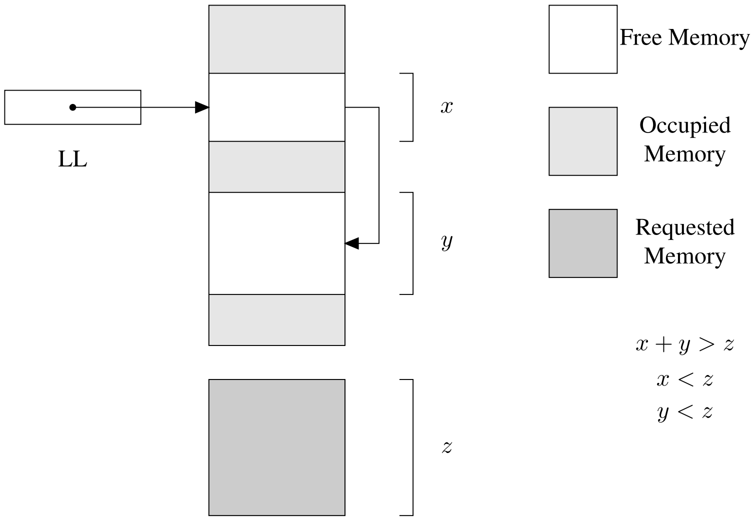


Fig. 5.11 External fragmentation

"compact" free memory, merging contiguous free blocks in such a way as to avoid external fragmentation. To achieve this objective, merging operations can be called which increase the load imposed by the management methods and therefore reduce efficiency.

Single free list The first technique we examine deals with a single free list, initially composed of a single memory block containing the entire heap. It is indeed convenient to seek to maintain blocks of largest possible size. It makes no sense, therefore, initially to divide the heap into many small blocks as, on the other hand, we did in the case of fixed-size blocks. When the allocation of a block of n words of memory is requested, the first n words are allocated and the pointer to the start of the head is incremented by n. Successive requests are handled in a similar way, in which deallocated blocks are collected on a free list. When the end of the heap's memory space is reached, is necessary to reuse deallocated memory and this can be done in the two following ways:

(i) **Direct use of the free list** In this case, a free list of blocks of variable size is used. When the allocation of a memory block n words in length is requested, the free list is searched for a block of size k words, where k is greater than or equal to n. The requested memory is allocated inside this block. The unused part of the block (of size $k - n$) if longer than some predefined threshold, is used to form a new block that is inserted into the free list (internal fragmentation is permitted below this threshold). The search for a block of sufficient size can be performed using one of two methods. Using *first fit*, the search is for the first block of sufficient size, while using *best fit*, the search is for a block whose size is the least of those blocks of sufficient size. The first technique favours processing time, while the second favours memory occupation. For both, however, the cost of allocation is linear with respect to the number of blocks on the free list. If the blocks are held in order of increasing block size, the two schemes are the same, because the list is traversed until a large enough block is found. Moreover, in this case, the cost of insertion of a block into the free list increases (from constant to linear), because is necessary to find the right place to insert it. Finally, when a deallocated block is returned to the free list, in order to reduce external fragmentation, a check is make to determine whether the physically adjacent blocks are free, in which case they are compacted into a single block. This type of compaction is said to be *partial* because it compacts only adjacent blocks.

(ii) **Free memory compaction** In this technique, when the end of the space initially allocated to the heap is reached, all blocks that are still active are moved to the end; they are the blocks that cannot be returned to the free list, leaving all the free memory in a single contiguous block. At this point, the heap pointer is updated so that it points to the start of the single block of free memory and allocation starts all over again. Clearly, for this technique to work, the blocks of allocated memory must be movable, something that is not always guaranteed (consider blocks whose addresses are stored in pointers on the stack). Some compaction techniques will be discussed in Sect. 8.12, when we discuss garbage collection.

Multiple free lists To reduce the block allocation cost, some heap management techniques use different free lists for blocks of different sizes. When a block of size n is requested, the list that contains blocks of size greater than or equal to n is chosen and a block from this list is chosen (with some internal fragmentation if the block has a size greater than n). The size of the blocks in this case, too, can be static or dynamic and, in the case of dynamic sizes, two management methods are commonly used: the *buddy system* and the *Fibonacci heap*. In the first, the size of the blocks in the various free lists are powers of 2. If a block of size n is requested and k is the least integer such that $2^k \geq n$, then a block of size 2^k is sought (in the appropriate free list). If such a free block is found it is allocated, otherwise, a search is performed in the next free list for a block of size 2^{k+1} and it is split into two parts. One of the two blocks (which therefore has size 2^k) is allocated, while the other is inserted into the free list for blocks of size 2^k. When a block resulting from a split is returned to the free list, a search is performed for its "buddy", that is the other half that was produced by the split operation, and it is free, the two blocks are merged to re-form the initial block of size 2^{k+1}. The *Fibonacci heap* method is similar but uses Fibonacci numbers instead of powers of 2 as block sizes. Given that the Fibonacci sequence grows more slowly than the series 2^n, this second method leads to less internal fragmentation.

5.5 Implementation of Scope Rules

The possibility of denoting objects, even complex ones, by names with appropriate visibility rules constitutes one of the most important aspects that differentiate high-level languages from low-level ones. The implementation of environments and scope rules discussed in Chap. 4 requires, therefore, suitable data structures. In this section, we analyse these structures and their management.

Given that the activation record contains the memory space for local names, when a reference to a non-local name is encountered, the activation records that are still active must be examined (that is, the ones present on the stack) in order to find the one that corresponds to the block where the name in question was declared; this will be the block that contains the association for our name. The order in which to examine the activation records varies according to the kind of scope under consideration.

5.5.1 Static Scope: The Static Chain

If the static scope rule is employed, as we anticipated in Chap. 4, the order in which activation records are consulted when resolving non-local references is not the one defined by their position on the stack. In other words, the activation record directly connected by the dynamic chain pointer is not necessarily the first activation record in which to look in order to resolve a non-local reference; the first activation record within which to look is defined by the textual structure of the program. Let us see an example.

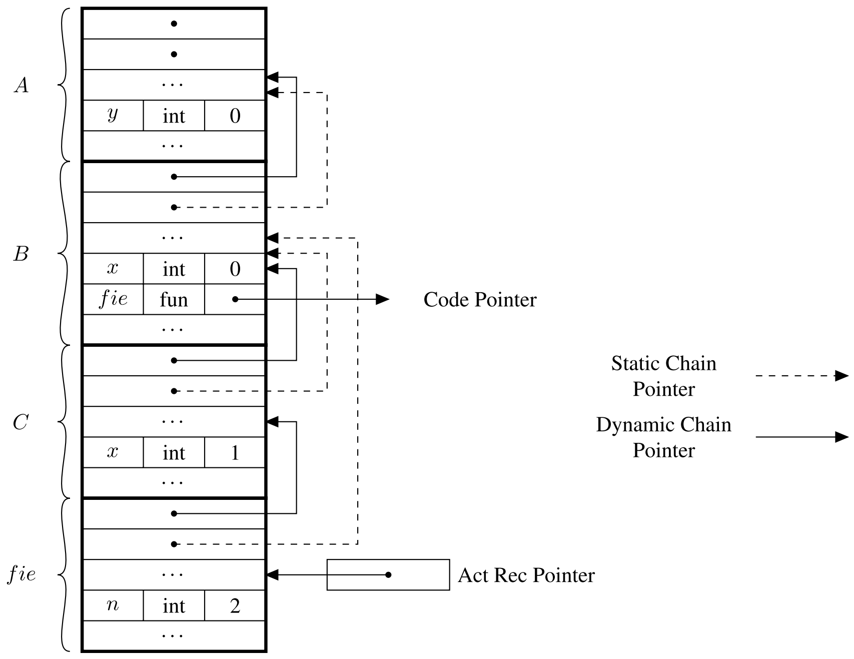


Fig. 5.12 Activation stack with static chain (see Example 5.3)

Example 5.3 Consider the following code (as usual, for ease of reference, we have labelled the blocks):

```
A:{int y=0;
   B:{int x = 0;
      void (int n){
         x = n+1;
         y = n+2;
      }
      C:{int x = 1;
         fie(2);
         write(x);
         }
      }
  write(y);
  }
```

After executing the call `fie(2)`, the situation on the activation-record stack is that shown in Fig. 5.12. The first activation record on the stack (the uppermost one in the figure) is for the outermost block; the second is the one for block `B`; the third is for `C` and finally the fourth is the activation record for the call to the procedure. The non-local variable, `x`, used in procedure `fie`, as we know from the static scope rule is not the one declared in block `C` but the one declared in block `B`. To be able

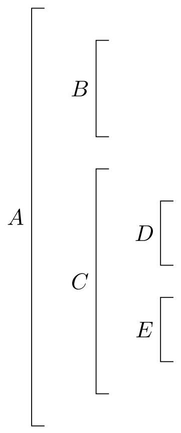


Fig. 5.13 A block structure

to locate this information correctly at runtime, the activation record for the call to the procedure is connected by a pointer, called the *static chain* pointer, to the record for the block containing the declaration of the variable. This record is linked, in its turn, by a static chain pointer to the record for block `A`, because this block, being the first immediately external to `B`, is the first block to be examined when resolving references non-local to `B`. When inside the call to procedure `fie`, the variables `x` and `y` are used, to access the memory area in which they are stored the static chain pointers are followed from `fie`'s activation record until first the record for `B` is encountered (for `x`) and then that for `A` (when searching for `y`).

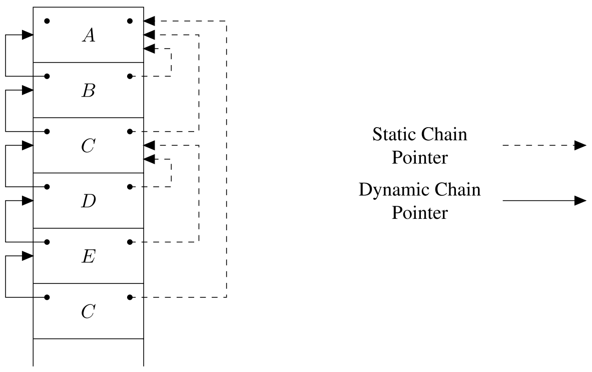
Generalising this example, we can say that, for the runtime management of static scope, the activation of the generic block `B` is linked by the *static chain pointer* to the record for the block immediately enclosing `B` (that is the nearest block that contains `B`). It should be noted that in the case in which `B` is the block for a procedure call, the block immediately enclosing `B` is the one containing the declaration of the procedure itself. Moreover if `B` is active, that is if its activation record is on the stack, then also the blocks enclosing `B` must be active and therefore can be located on the stack.

Hence, in addition to the *dynamic chain*, which is formed from the various records present on the system stack (linked in the order of the stack itself), there must exist a *static chain*, formed from the various static chain pointers used to represent the static nesting structure of the blocks within the program.

As an example, consider Fig. 5.13 which shows a generic structure of blocks which results from nested procedures. Consider now the sequence of calls: A, B, C, D, E, C, where it is intended that each call remains active when the next call is made. The situation on the activation-record stack, with is various static chain pointers, after such a sequence of calls is that shown in Fig. 5.14.

The runtime management of the static chain is one of the functions performed by the calling sequence, prologue and epilogue code, as we saw above. Such a management of the static chain can be performed by the caller and the callee in various ways. According to the most common approach, when a new block is entered, the caller calculates the static chain pointer and then passes it to the called routine. This

Fig. 5.14 Static chain for the previous structure and the sequence of calls A, B, C, D, E, C



computation is fairly simple and can be easily understood by separating the two cases:

The called routine is external to the caller In this case, by the visibility rules defined by static scope, for the called routine to be visible, it must be located in an outer block which includes the caller's block. Therefore, the activation record for such an outer block must already be stored on the stack. Assume that among the caller and the called routines, there are k levels of nesting in the program's block structure; if the caller is located on nesting level n and the called routine is on level m, we can assume therefore that $k = n - m$. This value of k can be determined by the compiler, because it depends only on the static structure of the program and therefore can be associated with the call in question. The caller can then calculate the static chain pointer for the called procedure simply by dereferencing its own static chain pointer k times (that is, it runs k steps along its own static chain).

Called inside calling routine In this case, the visibility rules ensure that the called routine is declared in same the block in which the call occurs and therefore the first block external to the called one is precisely that of the caller. The static chain pointer of the called routine must point to the caller's activation record. The caller can simply pass to the called routine the pointer to its own activation record as a pointer to the static chain.

Once the called routine has received the static chain pointer, it need only store it in the appropriate place in its activation record, an operation that can be performed by the prologue code. When a block exit occurs, the static chain requires no particular management actions.

We have hinted at the fact that the compiler, in order to perform runtime static-chain management, keeps track of the nesting level of procedure calls. This is done using the *symbol table*, a sort of dictionary where, more generally, the compiler stores all the names used in the program and all the information necessary to manage the objects denoted by the names (for example to determine the type) and to implement the visibility rules.

In particular, a number is maintained that depends on the nesting level and indicates the scope that contains the declaration of a name; this allows to associate to each name a number indicating the scope when the declaration for such a name is made. Using this number, it is possible to calculate, at compile time, the distance between the scope of the call and that of the declaration which is necessary at runtime to handle the static chain.

It should be noted that this distance is calculated statically and it also allows the runtime resolution of non-local references without having to perform any name searches in the activation record on the stack. Indeed, if we use a reference to the non-local name, `x`, to find the activation record containing the memory space for `x` it suffices to start at the activation record corresponding to the block that contains the reference and follow the static chain for a number of links equal to the value of the distance. Inside the activation record that is thus found, the memory location for `x` is also fixed by the compiler and, therefore, at runtime, there is no need for a search but only the static offset of `x` with respect to the activation record pointer is needed.

However, it is clear that, in a static model, the compiler cannot completely resolve a reference to a non-local name and it is always necessary to follow the static-chain links at runtime. This is why, in general, it is not possible to know statically what the number of activation records present on the stack is.

As a concrete example of what has just been said, consider the code in Example 5.3. The compiler "knows" that to use variable `y` in procedure `fie`, it is necessary to pass two external blocks (`B` and `A`) to arrive at the one containing the declaration of the variable. It is enough, therefore, to store this value at compilation time so that it can subsequently be known, at runtime, that to resolve the name `y`, it is necessary to follow two pointers in the static chain. It is not necessary to store the name `y` explicitly because its position inside the activation record for the block `A` is fixed by the compiler. Analogously, the type information that we, for clarity, have included in Fig. 5.12, is stored in the symbol table, and after appropriate compile-time checks, can, in a large part, be omitted at runtime.

5.5.2 Static Scope: The Display

The implementation of static scope using the static chain has one inconvenient property: if we have to use a non-local name declared in an enclosing block, k levels of block away from the point at which we currently find ourselves, at runtime we have to perform k memory accesses to follow the static chain to determine the activation block that contains the memory location for the name of interest. This problem is not all that severe, given that in real programs it is rare that more than 3 levels of block and procedure nesting are required. The technique called the *display*, however, allows the reduction of the number of accesses to a constant (2).

This technique uses a vector, called the *display*, containing as many elements as there are levels of block nesting in the program, where the kth element of the vector contains the pointer to the activation record at nesting level k that is currently active. When a reference is made to a non-local object, declared in an block

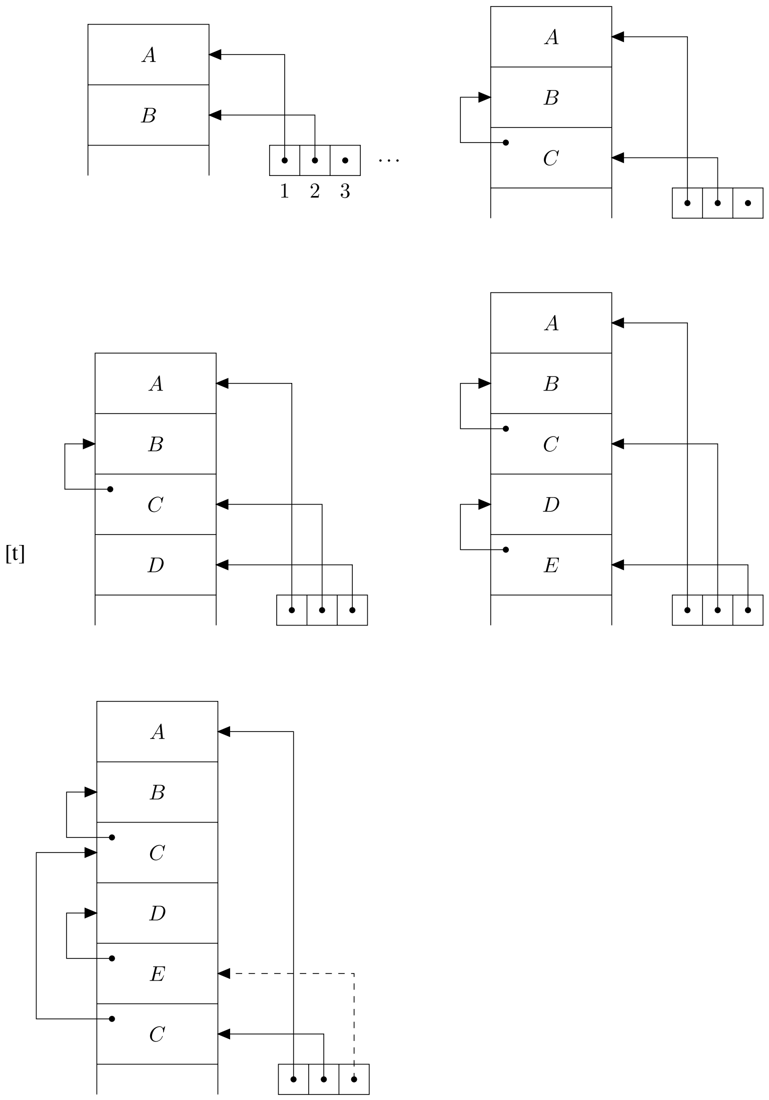


Fig. 5.15 Display for the structure in Fig. 5.13 and the call sequence A, B, C, D, E, C

external to level k, the activation record containing this object can be retrieved simply by accessing the kth position in the vector and following the pointer stored there.

Display processing is very simple, even if it is slightly more costly than static chain handling; when an environment is entered or left, in addition to updating the pointer stored in the vector, it is also necessary to save the old value. More precisely, when a procedure is called (or an in-line block is entered) at level k, position k in the display will have to be updated with the value of the pointer to the activation record for the call, because this has become the new active block at level k. Before this update, however, it is necessary to save the preceding contents of the kth position of the display, normally storing it in the activation record of the call.

The need to save the old display value can be better understood by examining the following 2 possible cases:

The called routine is external to the caller Let use assume that the call is at nesting level n and the called routine is at level m, with $m < n$. The called routine and the caller therefore share the static structure up to level $m - 1$ and also the display up to position $m - 1$. Display element m is updated with the pointer to the activation record of the called routine and until the called routine terminates, the active display is the one formed of the first m elements. The old value contained in position m must be saved because it points to the activation record of the block which will be re-activated when the called routine terminates; thereafter, the display will go back to being the one used before the call.

The called routine is located inside the caller The nesting depth reached this far is incremented. If the caller is located at level n, caller and called routine share the whole current display up to position n and it is necessary to add a new value at position $n + 1$, so that it holds the pointer to the activation record for the caller. When we have the first activation of a block at level $n + 1$, the old value stored in the display is of no interest to us. However, in general, we cannot know if this is the case. Indeed, we could have reached the current call by a series of previous calls that also use level $n + 1$. In this case, as well, it will be necessary therefore to store the old value in the display at position $n + 1$.

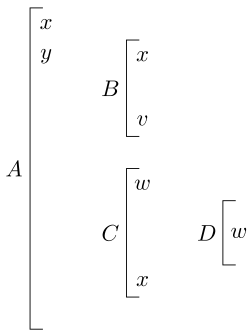
Both display update and the saving of the old value can be performed by the called procedure. Figure 5.15 shows the handling of the display for the call sequence A, B, C, D, E, C, using the block structure described in Fig. 5.13. The pointers on the left of the stack denote storage of the old display value in the activation record of the called routine, while the dotted pointer denotes a display pointer that is not currently active.

5.5.3 *Dynamic Scope: Association Lists and CRT*

Conceptually, the implementation of the dynamic scope rule is much simpler than the one for static scope. Indeed, given that non-local environments are considered in the order in which they are activated at runtime, to resolve a non-local reference to a name `x`, it suffices, at least in principle, to run backwards down the stack, starting with the current activation record until the activation record is found in which the name `x` is declared.

The various associations between names and the objects they denote which constitute the various local environment can be stored directly in the activation record. Let us consider, for example, the block structure shown in Fig. 5.16, where the

Fig. 5.16 A block structure with local declarations



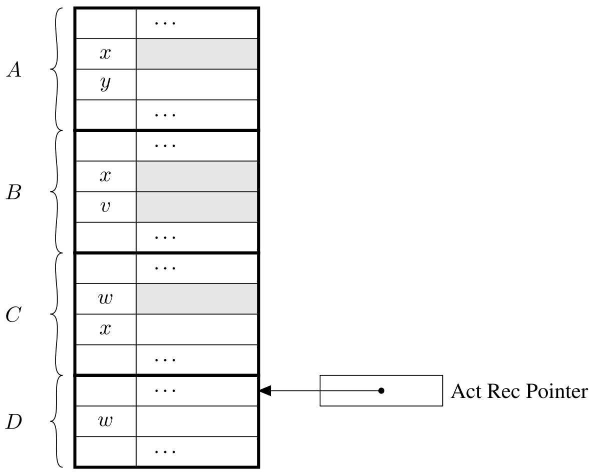


Fig. 5.17 Environment for block D in Fig. 5.16 after the call sequence A, B, C, D, with dynamic scope implemented using stored associations in the activation record. In *grey*: inactive associations

names denote local variable declarations (assuming the usual visibility rules). If we execute the call sequence A, B, C, D (where, as usual, all the calls remain active) when control reaches block D, we obtain the stack shown in Fig. 5.17 (the field on the right of each name contains the information associated with the object denoted by its name). The environment (local or otherwise) of D is formed from all the name-denoted object associations in which the information field is in white in the figure. The association fields that are no longer active are shown in grey. An association is not active either because the corresponding name is no longer visible (this is the case for v) or because it has been redefined in an inner block (this is the case for w and for the occurrences of x in A and B).

Other than direct storage in the activation record, name-object associations can be stored separately in an association list, called an *A-list*, which is managed like a stack. This solution is usually chosen for LISP implementations.

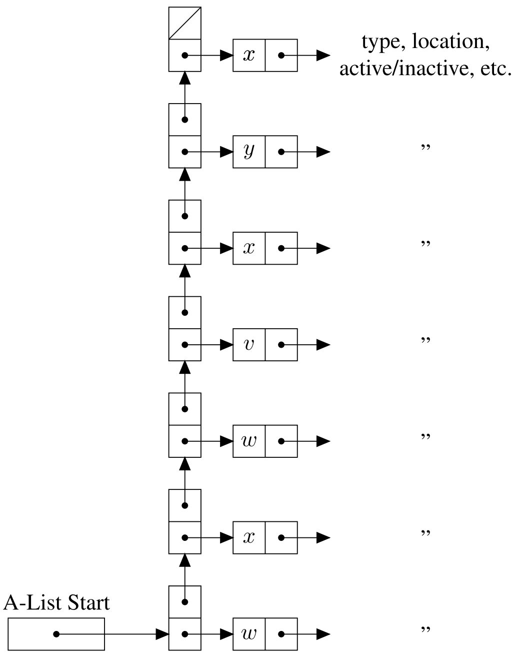


Fig. 5.18 Environment for block D in Fig. 5.16, after the call sequence A, B, C, D, with dynamic scope implemented using an A-list

When the execution of a program enters a new environment, the new local associations are inserted into the A-list. When an environment is left, the local associations are removed from the A-list. The information about the denoted objects will contain the location in memory where the object is actually stored, its type, a flag which indicates whether the association for this object is active (there can also be other information needed to make runtime semantic checks). Figure 5.18 shows how dynamic scope is implemented for the example in Fig. 5.16 using an A-list (the fields in grey are implemented using the flags described above and are omitted from the figure). Both using A-list and using direct storage in the activation record, the implementation of dynamic scope has two disadvantages.

First, names must be stored in structures present at runtime, unlike in the scheme that we saw for static scope. In the case of the A-list, this is clear (it depends on its definition). In the case, on the other hand, in which activation records are used to implement local environments, the need to store names depends on the fact that the same name, if declared in different blocks, can be stored in different positions in different activation records. Given that we are using the dynamic scope rule, we cannot statically determine which is the block (and therefore the activation record) that can be used to resolve a non-local reference; we cannot know the position in the activation record to access in order to search for the association belonging the name that we are looking for. The only possibility is therefore explicitly to store the name and perform a search (based on the name itself) at runtime.

The second disadvantage is due to the inefficiency of this runtime search. It can often be the case that is necessary to scan almost all of the list (which is either an A-list or a stack of activation records) in the case reference is made to a name declared in one of the first active blocks (as for “global” names).

Central Referencing environment Table (CRT) To restrict the effects of these two disadvantages, at the cost of a greater inefficiency in the block entry and exit operations, we can implement dynamic scope in a different way. This alternative approach is based on the *Central Referencing environment Table* (CRT).

Using the CRT-based technique, environments are defined by arranging for all the blocks in the program to refer to an single central table (the CRT). All the names used in the program are stored in this table. For each name, there is a flag indicating whether the association for the name is active or not, together with a value composed of a pointer to information about the object associated with the name (memory location, type, etc.). If we assume that all the identifiers used in the program are not known at compile time, each name can be given a fixed position in the table. At runtime, access to the table can, therefore, take place in constant time by adding the memory address of the start of the table to an offset from the position of the name of interest. When, on the other hand, all names are not known at compile time, the search for a name’s position in the table can be make use of runtime hashing for efficiency. The block entry and exit operations now are, however, more complicated. When entering block B from block A, the central table must be modified to describe B’s new local environment, and, moreover, deactivated associations must be saved so that they can be restored when block B exits and control returns to block A. Usually a stack is the best data structure for storing such associations.

It should be observed that the associations for a block are not necessarily stored in contiguous locations within the CRT. To perform the operations required of the CRT on block entry and exit, it is necessary, therefore,to consider the individual elements of the table. This can be done in a convenient fashion by associating with each entry in the table (i.e., with every name present) a dedicated stack that contains the valid associations at the top and, in successive locations, the associations for this name that have been deactivated. This solution is shown in Fig. 5.19 (the second column contains the flags).

Alternatively, we can use a single hidden stack separate from the central table to store the deactivated associations for all names. In this case, for every name, the second column of the table contains a flag which indicates whether the association for this name is active or not, while the third column contains the reference to the object denoted by the name in question. When an association is deactivated, it is stored in the hidden stack to be removed when it becomes active again. Considering the structure in Fig. 5.16 and the call sequence A, B, C, D, the development of the CRT is shown in the upper part of Fig. 5.20; the lower portion of the Figure depicts the evolution of the hidden stack.

Using the CRT, with or without the hidden stack. access to an association in the environment requires one access to the table (either direct or by means of a hash function) and one access to another memory area by means of the pointer stored in the table. Therefore, no runtime search is required.

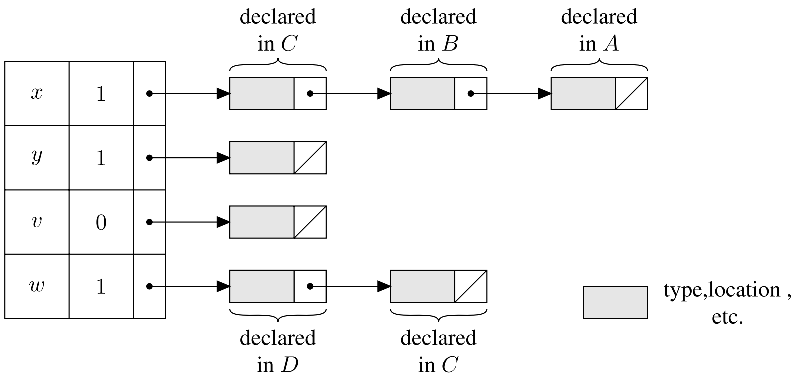


Fig. 5.19 Environment for block D in Fig. 5.16 after the call sequence A, B, C, D, with dynamic scope implemented using a CRT

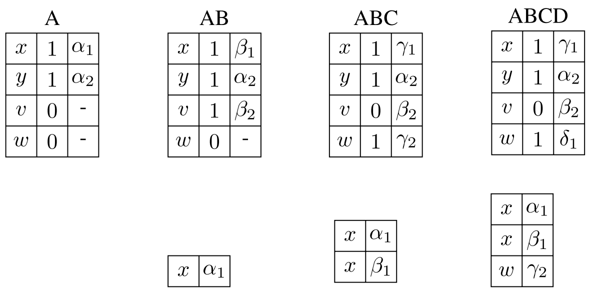


Fig. 5.20 Environment for block D of Fig. 5.16 after the call sequence A, B, C, D, with dynamic scope implemented using a CRT and hidden stack

A

x	1	α_1
y	1	α_2
v	0	-
w	0	-

AB

x	1	β_1
y	1	α_2
v	1	β_2
w	0	-

x	α_1

ABC

x	1	γ_1
y	1	α_2
v	0	β_2
w	1	γ_2

x	α_1
x	β_1

ABCD

x	1	γ_1
y	1	α_2
v	0	β_2
w	1	δ_1

x	α_1
x	β_1
w	γ_2

5.6 Chapter Summary

In this chapter, we have examined the main techniques for both static and dynamic memory management, illustrating the reasons for dynamic memory management using a stack and those that require the use of a heap. It remains to consider the important exception to this: in the presence of a particular type of recursion (called *tail recursion*) memory can be managed in a static fashion (this case will be given detailed consideration in the next chapter).

We have illustrated in detail the following on stack-based management:

- The format of activation records for procedures and in-line blocks.
- How the stack is managed by particular code fragments which are inserted into the code for the caller, as well as in the routine being called, and which act to implement the various operations for activation record allocation, initialisation, control field modification, value passing, return of results, and so on.

In the case of heap-based management, we saw:

- Some of the more common techniques for its handling, both for fixed- and variable-sized blocks.
- The fragmentation problem and some methods which can be used to limit it.

Finally, we discussed the specific data structures and algorithms used to implement the environment and, in particular, to implement scope rules. We examined the following in detail:

- The static chain.
- The display.
- The association list.
- The central referencing table.

This has allowed us better to understand our hint in Chap. 4 that it is more difficult to implement the static scope rules than those for dynamic scope. In the first case, indeed, whether static chain pointers or the display is used, the compiler makes use of appropriate information on the structure of declarations. This information is gathered by the compiler using symbol tables and associated algorithms, such as, for example, LeBlanc-Cook's, whose details fall outside the scope of the current text. In the case of dynamic scope, on the other hand, management can be, in principle, performed entirely at runtime, even if auxiliary structures are often used to optimise performance (for example the Central Referencing Table).

5.7 Bibliographic Notes

Static memory management is usually treated in textbooks on compilers, of which the classic is [1]. Determination of the (static) scopes to associate with the names in a symbol table can be done in a number of ways, among which one of the best known is due to LeBlanc and Cook [2]. Techniques for heap management are discussed in many texts, for example [4].

Stack-based management for procedures and for scope was introduced in ALGOL, whose implementation is described in [3].

For memory management in various programming languages, the reader should refer to texts specific to each language, some which are cited at the end of Chap. 13.

5.8 Exercises

1. Using some pseudo-language, write a fragment of code such that the maximum number of activation records present on the stack at runtime is not statically determinable.
2. In some pseudo-language, write a recursive function such that the maximum number of activation records present at runtime on the stack is statically determinable. Can this example be generalised?
3. Consider the following code fragment:

```
A:{int X= 1;
    ....
    B:{ X  = 3;
        ....
      }

  ....
    }
```

Assume that *B* is nested one level deeper that *A*. To resolve the reference to *X* present in *B*, why is it not enough to consider the activation record which immediate precedes that of *B* on the stack? Provide a counter-example filling the spaces in the fragment with dots with appropriate code.

4. Consider the following program fragment written in a pseudo-language using static scope:

```
void P1 {
   void P2 { body-ofi-P2
           }
   void P3 {
            void P4 { body-of-P4
                    }
            body-of-P3
           }
   body-of-P1
   }
```

Draw the activation record stack region that occurs between the static and dynamic chain pointers when the following sequence of calls, `P1`, `P2`, `P3`,`P4`,`P2` has been made (is it understood that at this time they are all active: none has returned).

5. Given the following code fragment in a pseudo-language with `goto` (see Sect. 6.3.1), static scope and labelled nested blocks (indicated by `A: { ... }`):

```
A: { int x = 5;
     goto C;
        B: {int x = 4;
            goto E;
            }
        C: {int x = 3;
            D: {int x = 2;
            }
                 goto B;
            E: {int x = 1; // (**)
               }
            }
     }
```

The static chain is handled using a display. Draw a diagram showing the display and the stack when execution reaches the point indicated with the comment `(**)`. As far as the activation record is concerned, indicate what the only piece of information required for display handling is.

6. Is it easier to implement the static scope rule or the one for dynamic scope? Give your reasons.
7. Consider the following piece of code written in a pseudo-language using static scope and call by reference (see Sect. 7.1.2):

```
{int x = 0;
 int A(reference int y) {
     int x =2;
     y=y+1;
     return B(y)+x;
 }
 int B(reference int y){
     int C(reference int y){
        int x = 3;
        return A(y)+x+y;
     }
     if (y==1) return C(x)+y;
     else return x+y;
 }
 write (A(x));
}
```

Assume that static scope is implemented using a display. Draw a diagram showing the state of the display and the activation-record stack when control enters the function A for the *second* time. For every activation record, just give value for the field that saves the previous value of the display.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, 1988.
2. R. P. Cook and T. J. LeBlanc. A symbol table abstraction to implement languages with explicit scope control. *IEEE Trans. Softw. Eng.*, 9(1):8–12, 1983.
3. B. Randell and L. J. Russell. *Algol 60 Implementation*. Academic Press, London, 1964.
4. C. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Addison-Wesley, Reading, 1996.