Chapter 3

Run-time Memory Layout

In this chapter we examine the overall layout of large memory blocks, while in the chapters that follow we will describe the detailed representation of individual items within those blocks of memory.

3.1 The Categories of Memory

On a physical level, computer memory is divided into a sequence of words, which are manipulated much like an array, accessing individual locations by their numerical offsets. The lower portion of memory is typically occupied by the operating system. Thus, from the compilers point of view, words are numbered from some platform-specific lower address, to an upper addresses determined by the limits of memory (either physical or virtual). Normally, however, the compiler writer has little concern with the actual location of values in memory. Instead, we can envision memory as an amorphus container of values of various different types. Among the major categories of items found in memory (see Figure 3.1) are the following:

- **Code**. The actual instructions used to implement the actions described by a computer program.
- **Global values**. Values that are unique and have a known size.
- **The Stack**. Space for local variables and parameters.
- **The Heap**. Space for values that are not tied to procedure entry and exit.

Each of these categories will be described in more detail in following sections. As we noted, the compiler is, for the most part, not concerned with the actual locations of these elements. Instead, the compiler deals either with symbolic addresses, registers that hold

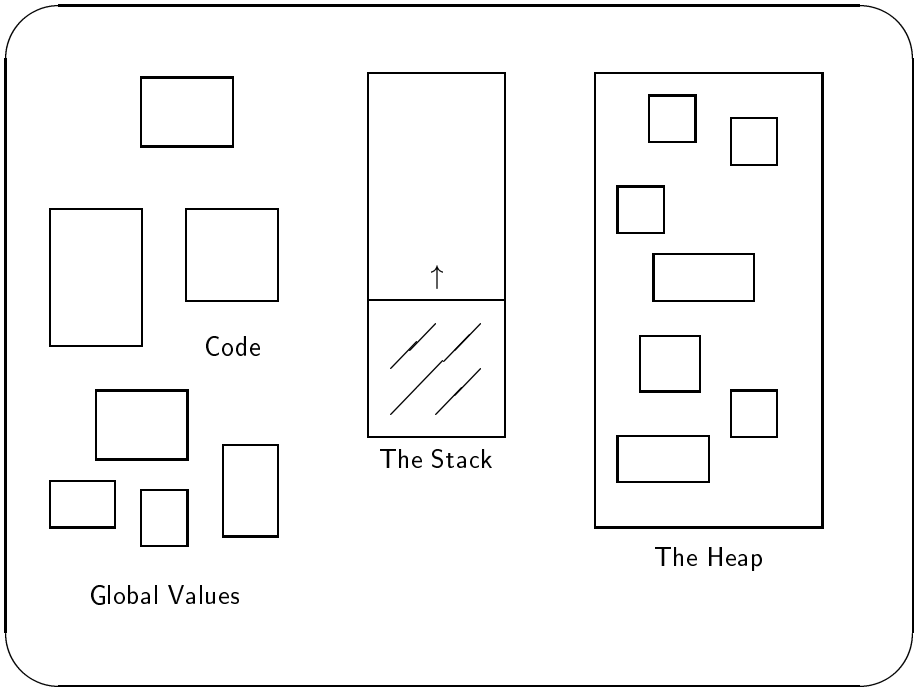
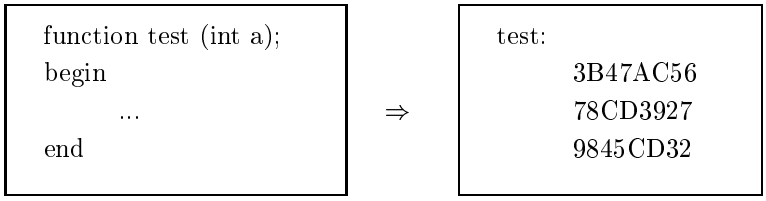


Figure 3.1: Varieties of items found in memory

addresses, or with address computations (for example, addresses computed as a given offset from another address). A separate utility, known as a *linker*, is reponsible for assigning these large blocks of memory to physical locations, and replacing symbolic memory addresses with their actual value. While the task of the linker is vital, the details of its operation are of little concern to the compiler writer. Further information on linkers can be found in many operating systems textbooks.

3.1.1 Code

The majority of this text is concerned with the task of translating computer programs into assembly language instructions, which are then encoded as sequences of binary values. These binary values are placed into memory, and accessed to execute the associated procedure. In essence, therefore, code is nothing more than an initialized section of global data:



As with other global data, we are generally not concerned with the actual location of code in memory, but refer to the location using a symbolic name. For example, a symbolic name will be used to denote the address of the first word in the sequence of assembly language instructions generated by the compiler. Other assembly language instructions will refer to the procedure using this symbolic name. Similarly, since we are using assembly language, we can use symbolic names to refer to locations within a procedure. It is the job of the assembler to replace symbolic label names with offsets relative to the start of the procedure. It is the job of the linker to place the entire sequence of instructions into a specific location in memory, and replace the symbolic names that represent procedures with the actual location at which the data was stored.

Note carefully that we are here discussing the location of the binary instructions generated in association with a function. The data values involved with an execution of the procedure, such as parameters and local variables, are stored in an entirely different location, on the execution stack (see Section 3.1.3).

Mangled Names

Linkers generally like to work with unique names. That is, for every global name, there is one location where the information associated with that name is stored. Programming languages, on the other hand, frequently allow names to be overloaded, if the context surrounding the use of a name makes the meaning clear. A local variable in one procedure can

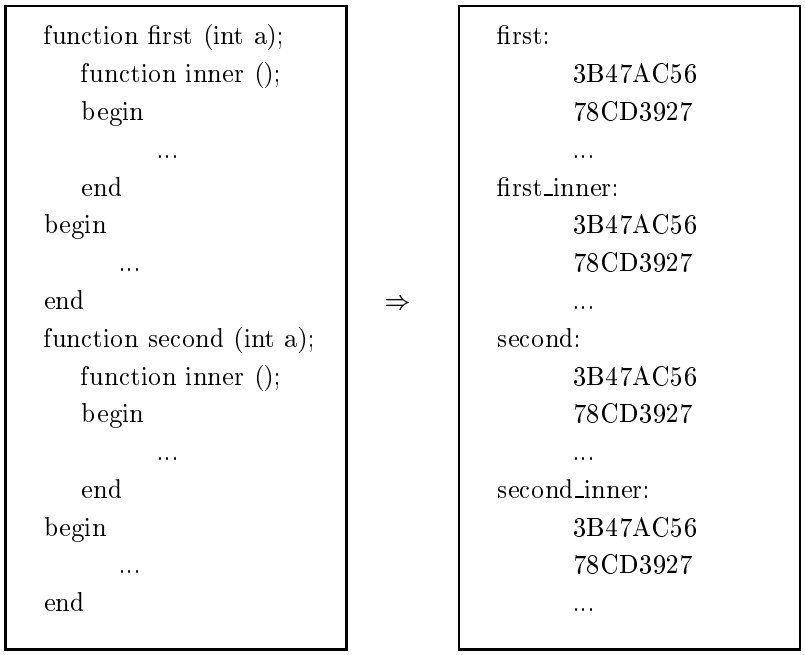


Figure 3.2: Mangled names generated for nested functions

have the same name as a local variable in a different procedure, for example, since for any use of the variable the surrounding procedure will indicate what name is expected.

It will sometimes happen that names associated with global data areas have been overloaded with two or more meanings. A simple example where this can occur is with nested procedures. Consider the situation shown in Figure 3.2. Here there are two procedures named inner, one nested within the function first and one nested within the function second. In order to produce a unique name for the linker, the code generated for the first of these is known intenrally as first_inner, while the code for the second is named second_inner. This combination of name and surrounding context is known as a *mangled name*. Compilers use various different techniques for mangling names, sometimes even incorporating an encoding of the argument types associated with a procedure.

Mangled names will frequently cause difficulties for debuggers, which generally operate using the compiled output, and not the original source program. Mangled names can also appear in error messages generated by the linker (for example, an error caused by the inability to match a function name with a function body). It is therefore useful to know how the particular compiler you are using generates mangled names.

3.1.2 Global Storage

Data elements must have two properties in order to be considered as candidates for storage as a global value. First, the entity must be unique, that is, there must be only one copy of the value. Second, the element must have a size that is known at compile time. A typical global storage element might be an integer variable, declared outside the scope of any procedure, and thus not local to any function.

Note that the term "global" refers to the accessibility of the value in the generated binary code, and not necessarily to the status of a variable in the original source program. While truly global variables in the source program usually map into global storage, many other values can as well. A local variable or data member declared as static in either C++ or Java possesses the two properties mentioned above, despite the fact that such values can only be accessed from within the procedure in which they are declared. However, because they fulfill the requirements stated, they are candidate for storing as a global value. We will subsequently see other values that also fit these requirements.

There are two classifications schemes one can use to further subdivide global storage. Some machines distinguish between elements that will be initialized and never modified, such as code, versus quantities that are expected to change value during the course of execution, such as data variables. The latter can be further subdivided into those elements that are initialized with a starting value, and those that are not. Depending upon the platform being used, all global variables may be stored in adajcent locations, or alternatively the various categories of global values may be stored in different locations.

In any case, the compiler writer is seldom concerned with the actual location of any global value, and instead refers to the address using a symbolic name. As with code, it is then the job of the linker to replace symbolic names with the addresses where the corresponding objects have been placed in memory.

Name manging can sometimes occur with data values, as well as functions. For example, in C++ a static variable named x declared within a procedure first may be known internally as first_x, so as to distinguish the global data area from other data areas associated different variables that also happen to be named x.

3.1.3 The Execution Stack

The execution stack is generally where local variables and parameters are stored. There are two reasons such values are not stored as global variables. First, doing so would be a waste of memory, since the space for local variables is necessary only when the procedure in which they are declared is being executed. Since typically at any one time the vast majority of procedures are not being executed, there is no reason why their local variables should be occuping memory space. Secondly, in the case of recursive procedures, it is necessary to maintain multiple distinct copies of local variables, each separate from the others. Since we cannot predict, during compilation, the number of iterations that a recursive procedure will perform, we cannot determine how much space to set aside for these local values.

Instead, local variables and parameters for all currently executing procedures are stored using a structure called the run-time execution stack, or simply the *stack*. Under some operating systems there is a single execution stack, while under other systems each executing thread has its own execution stack. In either case, the structure of the stack follows a pattern that we will subsequently describe in section 3.2.

Values stored on the execution stack must satisfy two properties:

1. The lifetime of the value must be tied to procedure entry and exit. That is, the value comes into existance when the procedure starts execution, and can be eliminated altogether when a procedure finally exits.

2. The size of the data value is known at compile time.

Examples of such values include some local variables (those, such as integer values, that have a known size) and parameters.

Because the exact memory address of values held in the run-time stack cannot be predicated at compile time, local variables cannot be given a symbolic name, as can globals. This does not mean, however, that there is absolutely no information concerning the location of such values. As we will see in Section 3.2, local variables are addressed by a combination of a register (which will hold a run-time memory address) and a constant offset known at compile time.

3.1.4 The Run-time Heap

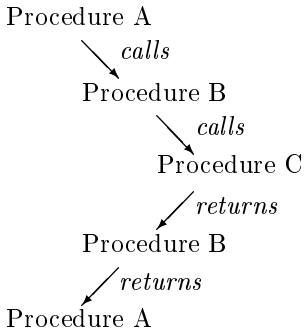
Some programs must manipulate values that fail to satisfy one or both of the constraints necessary for values to be stored on the execution stack. An example might be an array in which the number of elements is not known until run time. Another example is a value that is created within a function, but must be manipulated after the function in which it is created has exited. Still another source of problems are polymorphic variables in object-oriented languages, since the class, and hence the size, of the value referenced by the variable is not known at compile time.

Values that cannot be stored on the stack are instead stored in an area known as the *heap*. Heap memory is managed as a collection of data values. A heap management system keeps track of the utilization of heap memory. Each time space is required for a values that will be stored in the heap, this manager is invoked, and a memory address for the new value is returned. There are various techniques used to recover heap memory. These will be described in more detail in Section 3.3.

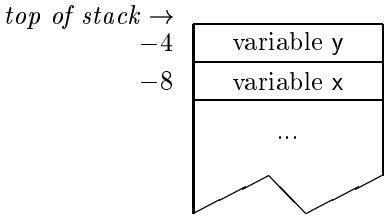
3.2 Activation Records

In almost all computer languages, local variables are created when a procedure begins execution, exist as long as the procedure is running or has called another procedure, and can be

eliminated once the procedure returns from execution. Furthermore, procedures typically obey a stack-like protocol; if procedure A calls procedure B, procedure B must exit in order for procedure A to be restarted, and therefore must exit before procedure A can exit.

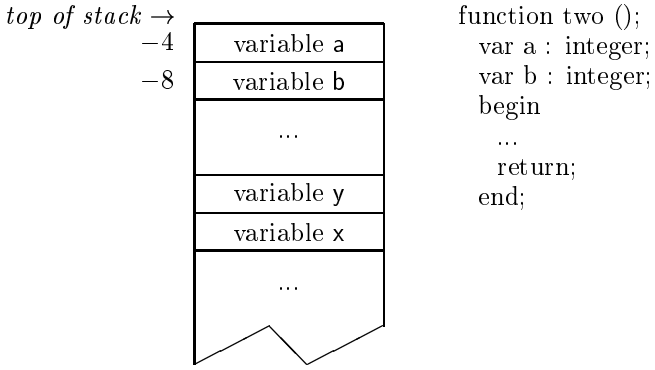


This observation of the stack-like behavior of function calls leads to an elegant mechanism for managing local variables. Space for local variables for all currently executing procedures is managed in a single run-time stack. When a procedure starts execution, the top of the stack is incremented by an amount equal to the space required for the local variables used by the procedure. Because local variables each have a known size, the location of any variable can be determined at compile time as an *offset* relative to the current top of stack. For example, in the picture, the compiler knows that variable x is found at offset -4 from the top of stack, variable y at offset -8, and so on. So, even if the exact location of location variables cannot be determined, a simple calculation using the top of stack can be used to find a given value.

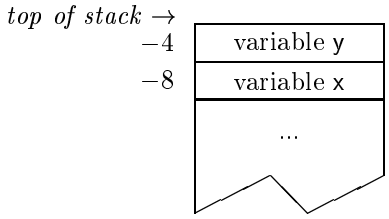



```
function one ();
  var y : integer;
  var x : integer;
  begin
    ...
    two();
  end;
```

When the current procedure calls a new procedure, space sufficient to hold local variables for the new procedure is pushed on the stack. Note that the offsets for variables in the caller procedure are no longer valid; however, this is normally of little concern, since unless the procedures are nested, these variables cannot be accessed while the new procedure is executing. We will examine how access to variables in nested procedures is handled in the next chapter.



Similarly, when a procedure is finished with execution, the procedure knows how much memory was allocated for local variables, and can restore the stack top to its previous value, thereby restoring the the run-time stack to the state it held prior to execution of the procedure. This is necessary to ensure that the calling procedure can continue to access its own local variables.

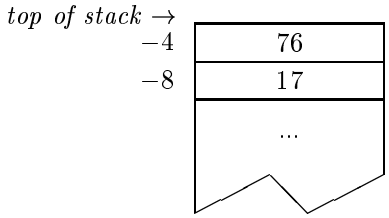


Local variables are not the only category of data values that are tied to function execution. Parameters also have this property. However, parameters are *shared* between a calling procedure and a called procedure. A calling procedure evaluates the argument expressions, and must make the value of these expressions available as the initial value for the parameters used by the called procedure.

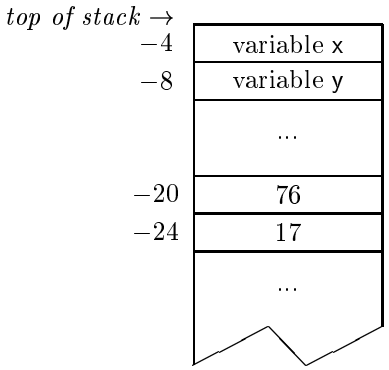
Once more, the stack provides an elegant means of accomplishing this. When the calling procedure is to perform a function call, it evaluates the parameter values, and pushes the results on to the stack. For reasons we explore further in the exercises, argument values are traditionally evaluated right to left, that is last to first. In this fashion, the function call:

```
three (2*38, 5+12)
```

first evaluates the expression $5 + 12$ and pushes the result, the integer value 17, on to the stack. Next the value of $2 * 38$ is evaluated and pushed on to the stack. The result is that the first argument is, in the end, the topmost value on the stack, as shown in the following:



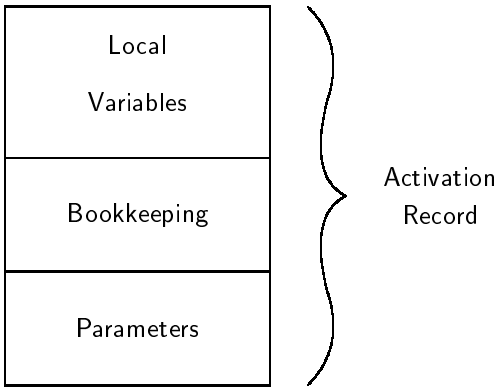
Control is then transferred to the calling procedure, which then allocates space for local values. Just as the offsets for local values can be determined at compile time relative to the top of stack, offsets for parameters are also similarly known.



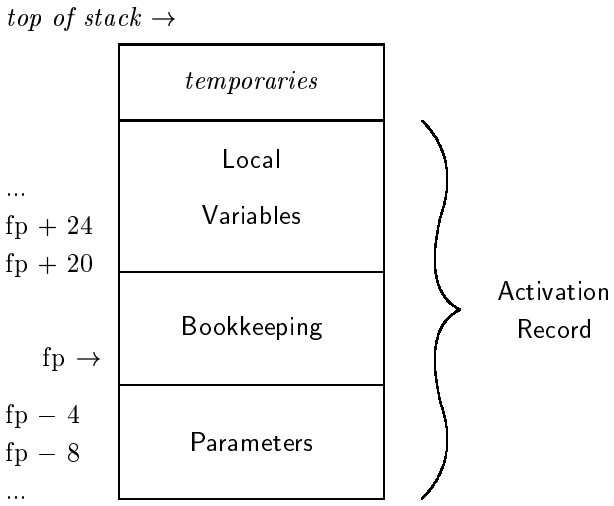

```
function three (p : integer, q : integer);
  var x : integer;
  var y : integer;
  begin
    ...
  end
```

We have not yet completed identifying all the information that is tied to procedure invocation and exit. Consider the actions that must occur when a procedure is finished with execution. Control must then return to the calling procdure. But one procedure can be called many times from many different places–how does the called procedure know where to return? The solution to this problem is to push the return address on the top of the stack, along with parameter values. As we will see in subsequent chapters, there are other "internal" values that can also be usefully stored on the stack. A good example of this is the value of registers, so that the called procedure will not destroy the contents of registers being used by the calling procedure.

The section of the run-time stack devoted to data for one procedure invocation is known as the *activation record* for the procedure. We have identified three parts to the activation record. The parameter section is built by the caller, who also places the return address on the stack. The called procedure may store other internal values, such as the contents of registers, or the previous top of stack pointer. These are here identified as the "bookkeeping" section. Finally, space is allocated for local variables.



It is often convenient to use the stack for yet another purpose, which is as a storage area for temporary computations, such as the intermediate results that can arise during the evaluation of a complex expression. But doing so destroys a property that we have been up to this point relying upon, namely that the offset of values relative to the top of the stack can be determined at compile time. To solve this problem, actual systems usually use a *pair* of pointers to reference values stored on the stack. In addition to the top of stack pointer, the *frame pointer* is a register that points to a known location in the activation record, typically somewhere in the middle of the area we have identified as storing the bookkeeping information. Both parameters and local variables can be accessed as offsets relative to the frame pointer, independent of the value held by the top of stack pointer. These offsets remain valid, regardless of how the top of stack pointer may increase or decrease during the evaluation of expressions.



```
Caller:
    evaluates the arguments, and pushes them on the stack
    pushes its own current address (the return address)
        on to the stack
    transfers control to the called procedure

Called Procedure:
    saves the current frame pointer and stack pointer
    saves other register values
    creates space for local variables

    executes the procedure

    pops the space for temporaries and local variables from the stack
    restores the saved value of registers
    restores the frame pointer and stack pointer
    loads and return address, and returns control to the caller

Caller:
    pops the space for parameters from the stack
    continues execution
```

Figure 3.3: A Typical Procedure Calling Sequence

Part of the bookkeeping information stored with the activation record is the previous value of the frame pointer, the value that pointed to the activation record for the calling procedure. By saving this value, the frame pointer can be restored to its previous value before control is returned to the caller. The stored frame pointer value is sometimes referred to as the *dynamic link*, since it is created dynamically at run-time. In the next chapter we will contrast this with another type of link to activation records, which is used to access data values from surrounding procedures.

To summarize, procedure entry and exit is like a dance performed by the caller and the the called procedure. Each has a specific role to play in the creation of the activation record. In pseudo-code, a typical procedure invocation protocol looks something like sequence of instructions shown in Figure 3.3.

There are several points to note regarding this process. Although we think of the activation record as one unit, it is in fact built in part by the caller, and in part by the called procedure. On many machines the tasks of pushing the current address on to the stack and transferring control to a new location are combined in a single assembly language instruc-

tion, often named **call**. Local variables are found on one side of the frame pointer, while parameters are found on the other. Thus, one typically have positive offsets, while the other have negative offsets. For historical reasons stacks often grown downwards, in which case parameters are positive, and local variables are negative. An important principle illustrated by the process shown in Figure 3.3 is that each party is reponsible for cleaning up any storage it creates–the parameters are both allocated and released by the caller, while local variables are both allocated and released by the called procedure.

3.2.1 Register Parameters

Machines with a large number of resources sometimes use registers to pass the first few parameter values. Often there is a small limit on the arguments that can be passed in this fashion, for example only the first 4 integer or pointer arguments might be so handled. Depending upon the hardware, floating point or structured values may not be passed in this form, and these would then be passed on the stack as described previously.

Passing arguments in registers rather than on the stack saves memory accesses, for both the caller (who would otherwise need to write the argument to memory in the stack) and for the called procedure (who must read the parameter value from the stack). However, this is only true if the called procedure does not need to use the register in which the argument resides. If the register is needed by the called procedure, the value passed by the caller in the register must be saved in the register save area, along with the values for whatever other registers the called procedure will use. The parameter value can thereafter be read, when needed, from this memory location.

It might seem that if the value of an argument is going to be saved in memory anyway (in the register save area), that nothing has been gained–we have simply moved the responsibility for saving the value in memory from the caller to the called procedure. But if the result seems to be no better than before, one can also argue that the result is no worse than before. Furthermore, it will often happen that the called procedure will *not* require the register in question (for example, if it does not itself make any further procedure calls). In such cases, the store to memory can be avoided, and a real savings in both time and space will be realized.

Despite the fact that register parameters are becoming more common in modern machines, we will for the remainder of this text treat this technique as a machine-specific optimization, and continue to assume that all parameters are passed through the activation record stack.

3.2.2 Non Traditional Activation Records

There are some situations where activation records should not be deleted when the associated function is finished with execution. This can happen, for example, in languages that have both nested functions and the ability to manipulate functions as values. By manipulate functions as values, we mean the ability to assign a function to a variable, pass a function

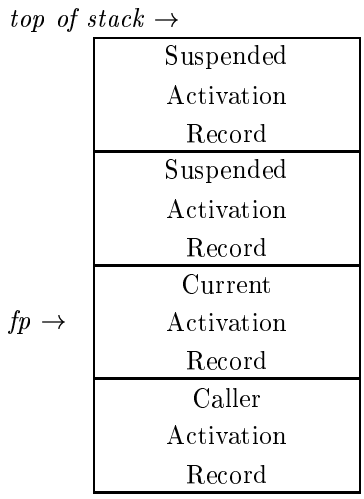
as an argument, or return a function as the result of executing another function. Consider the following example program:

```
function A(); var fun : function();

    function B(); var y : integer;

        function C(); var z : integer;
        begin
            print(y);
        end;

    begin
        y := 7; fun := C;
    end;

begin
    B(); fun();
end;
```

Normally, the activation record for B, which contains the space for the variable y, will be deleted once B returns. But if this occurs, there is no value for C to print when it is invoked through the variable fun. For this reason the activation record for B must be retained at least until C is finished with execution. This problem is sometimes known as the *up-level fun-arg problem*. Up level means that it occurs when functions are used in a higher level than they are declared (the variable fun is a high scoping level than the function C). Fun-arg is short for *functional argument*, because an analogous problem can arise when functions return another function through a by-reference argument value.

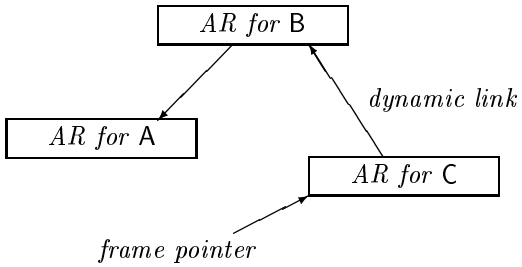
A similar situation occurs in languages such as Icon or Clu, which allow functions to be *suspended* rather than exited. A suspended function can later be resumed. When resumed, the values of local variables held by the function should be the same as at the time of suspension. Yet another example is found in Prolog, where a suspended clause can be resumed should later processing encounter a logically impossible condition.

There are generally two approaches to solving this problem. One technique, used by many Prolog systems, is to separate the tasks of restoring the frame pointer and restoring and decrementing the stack pointer. When an activation record must be saved, the frame pointer is restored on procedure exit, but the stack pointer is not. Thus, suspended activation records continue to be held in the space between the current activation record and the top of stack.

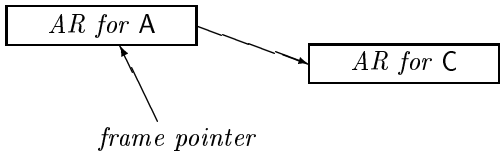


The difficulty with this technique is determining when the suspended activation records can finally be discarded, and the fact that even activation records that may not be needed must also be saved, if they occur in an activation record between the top of stack and the currently active record. A particular feature of the Prolog language makes it easy to determine when a suspended procedure is no longer needed, making this approach attractive.

An alternative is to not store all activation records on the stack, but to keep those activation records that may be needed for a later computation on the heap. Since allocation and recovery of heap-based memory is not tied to procedure entry and exit, activation records on the heap can be saved until they are needed. For example, suppose that procedure A has called procedure B, which in turn has called procedure C. The structure of activation records on the heap is very similar to those stored on the stack. As with the execution stack, the frame pointer references the activation record for the procedure currently being executed. Each activation record contains a pointer to the activation record for the procedure from which it was called. Thus, we have a picture similar to the following:



If, for example, the activation record for C must be saved as part of a function value stored in A, the a variable in the activation record for A will continue to reference C, even after control (indicated by the frame pointer) has returned to A. (In the next chapter we will describe in more detail the internal representation used by a value of type "function").



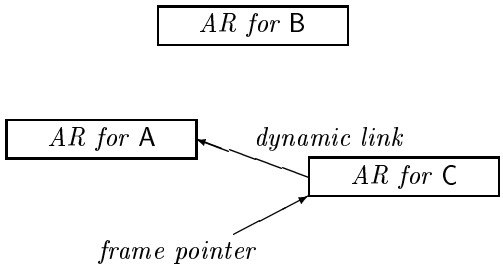
In some languages such activation records are constructed on the heap from the start, while in other languages the activation records are first constructed on the stack, then copied on to the heap when necessary. However it is done, activation records must sometimes be generated on the heap in languages such as Lisp, Smalltalk, and ML.

Tail Recursion Elimination

Although we will not discuss optimizations in detail until a later chapter, there is one important optimization that is easy to understand in the context of languages that use heap storage for activation records, rather than a run-time activation record stack. It is common for a procedure to return a value that is itself being generated by a procedure call, as in the following:

```
function B ( a : integer ) : integer;
begin
    ...
    return C (a + 2); // return result yielded by C
end;
```

In this situation there is no reason to continue to store the activation record for B, since once C is called the activation record for B will never again be accessed. There is only one piece of information contained in the activation record for B that is important to not misplace, and that is the return address to the procedure that called B. By pushing this return address on to the activation record stack for C, rather than a return address in the procedure B, and restoring the frame pointer to the caller of B before transferring control to C, we can eliminate all references to the activation record for B. That is, rather than the picture shown earlier, we have the following situation:



When procedure C terminates, it will now return its value as well as control over execution directly to A, avoiding the return instruction in procedure B altogether. There are now no references at all to the activation record for B, and the memory occupied by this value can be recycled by the heap manager. This important optimization is termed *tail recursion elimination.*

3.3 Heap Memory Management Techniques

The *heap* is a memory area used to store values that cannot be maintained in either global data areas or on the stack. Generally this is because the amount of storage needed for a value is not known at compile time, or because the lifetime for the storage value is not tied to a procedure entry and exit.

Values on the heap are allocated by a memory management system. The user makes a request for a section of memory of a given type, and the heap manager returns a pointer to a block of memory sufficient in size to hold the value of the specified type. In some languages, such as C, the programmer simply specifies the desired number of bytes of memory:

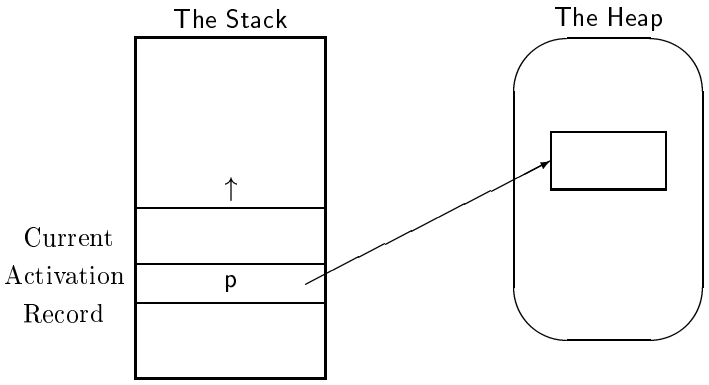
```
double * p;
    // allocate 8 bytes of memory
p = (double *) malloc(8);
```

More strongly typed languages, such as C++ and Java, specify the result using a type, rather than a size. The compiler determines the size of the requested type, so in fact in execution this form looks internally very much like the C version:

```
    // C++, declare a pointer to double
double * p;
    // allocate a new double value
p = new double;
```

The reader should make certain they have a clear picture in their mind where all values now reside. The variable p is held somewhere in an activation record stack. This is permitted

since pointers have a known size, easily determined at compile time. The *value* held by p now references a block of storage somewhere in the heap.



Object-oriented languages often tie the execution of a *constructor* function to the allocation of memory. In this case the constructor is simply treated as an ordinary function, executed using the newly created value as an argument once the memory allocation has taken place.

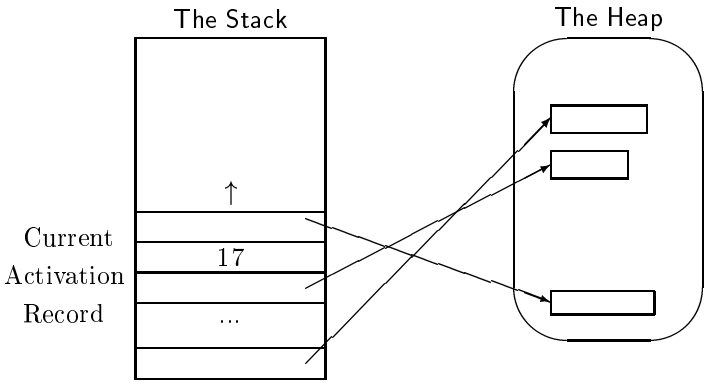
```
    // C++, declare a pointer to a comples
complex * p;
    // create a new complex value
p = new complex (3, 4);
    // constructor will automatically be invoked,
    // once allocation is successful
```

Because the heap manager executes at run-time, it does not demand that the size of memory being allocated be known at compile time. For example, it is possible to create an array in which the number of elements is determined dynamically.

```
    // Java – declare an array of unknown size
double [ ] points;
    // perform some calculation to determine size
int max = height * width;
    // allocate array with max elements
points = new double [ max ];
```

Although Java is described as a language in which there are no pointers, this is only superficially true. While the programmer does not see any pointers, internally all non-primitive values are in fact represented as pointers. An activation record that holds an

object value passed as argument, an integer local value, and two object values as local memory can be envisioned as the following:



There are two broad approaches to the recovery of heap based memory. Languages such as C++ and Pascal leave this task to the programmer, who indicates that dynamic memory is no longer required using an explicit statement, such as the free statement in Pascal:

```
type
    ptr : double;
    ...
begin
    allocate (ptr);
    ...

    free ptr;
end;
```

Memory that is freed is then "recycled", and may be used to satisfy a later dynamic memory request.

The difficulty with this approach is that it is often hard to determine when a value is finished being used, particularly if it is being shared between several functions. If the programmer therefore simply decides not to free any dynamically allocated memory, the heap may eventually become full, and the program will unexpectedly halt due to lack of memory.

An alternative, used by languages such as Lisp, Smalltalk, and Java, is to provide a *garbage collection* system. The garbage collection system monitors the use of dynamically allocated memory, and automatically detects when a value that has been allocated on the heap no longer has any pointers referring to it, and thus can not have any influence on future computations. The garbage collection system then recovers these values, and as

before recycles them to satisfy future requests. The techniques used by garbage collection systems are interesting, but are beyond the scope of the this text. Readers interested in the details are referred to some of the references cited at the end of the chapter.

Further Reading

The run-time memory layout of programs is described in Chapter seven of (Aho 86), and Chapter six of (Appel 97). Techniques for garbage collection are discussed in Chapter 13 of (Appel 97). A detailed description of how Prolog systems save their activation records is given in Chapter 4 of (Aït-Kaci 91). Further information on implementation techniques for Prolog is found in (Campbell 84). The implementation of storage management in Icon is described in Chapter 11 of (Griswold 86).

Exercise 5 describes an alternative solution to the problem described in Section 3.2.2. This alternative is explored in (Hanson xx).

Study Questions

1. From the compilers point of view, what are the four different categories of values stored in memory by a running program?
2. In what way can code be considered to be simply an initialized global value?
3. What is the purpose of a compiler generating an internal managled name for a global value?
4. What two characteristics are required for a quantity to be held as a global value?
5. What types of values are stored on the execution stack?
6. Give two reasons why these values are not held in global locations.
7. Give two reasons why a value might be held in the heap, rather than on the execution stack.
8. What does it mean to say that procedure invocations obey a stack-like protocol?
9. What are the three major sections of an activation record?
10. Explain the need for a frame pointer, kept distinct from the top of stack pointer.
11. Between a caller and the called procedure, who is responsible for creating each of the major sections of an activation record?
12. Why would a compiler want to pass parameter values in registers, rather than on the stack?

13. Under what conditions might a compiler not remove an activation record from the stack following termination of a procedure?

14. What is tail recursion? What is tail recursion elimination?

15. What is a heap? What type of values are stored on a heap?

16. What are the two general approaches to heap memory recovery?

Exercises

1. In many languages designed in the 1950's, parameters and local variables were not stored on the stack, but were instead stored in blocks of memory allocated as global storage. Explain why this precluded the use of recursive functions in these languages.

2. Show the activation record for the following procedure. Assume that integer values and pointers occupy 4 bytes, double precision values require 8 bytes, and that the generated code must save the contents of 3 registers in addition to the stack and frame pointers.

```
function example (x : integer, y : double)
    var z : * integer;
    var w : double;
    begin
        ...
    end
```

3. The reason that arguments are typically pushed on the stack in reverse order is to accomodate procedures that can take a variable number of arguments, such as the writeln procedure in Pascal, or the printf function in C. Assume that the first parameter value is found at location 4 relative to the frame pointer, and that parameter offsets increase. For this question assume that all parameters are integer values, occupying 4 bytes. Give the activation record offsets for the first 4 parameter values.

4. The situation described in the previous question is more complex when parameters are allowed to have various different types and lengths. For example, in C the procedure printf is used to print a variety of different types of values. The argument types are described by the first parameter value, which must be a string. This string contains both literal text, and special formatting characters, incicated by percent signs. For example, the following expression prints the value found in an array at the given index location:

```
printf(" a[ %d ] is %g", i, a[i]);
```

Give an English language description of the execution of the printf function. You only need handle integer arguments, which occupy 4 bytes and are indicated by a %d in the formatting string, and double precision arguments, which occupy 8 bytes and are indicated by a %g.

5. Section 3.2.2 described a problem that can occur when a language allows variables of type "function". One solution to this problem is to disallow such variables. However, another solution is to allow function variables, but eliminate the nesting of functions. Present an argument to show how this limitation guarantees that the problem described in Section 3.2.2 cannot arize.

6. Although not as easy, it is possible in certain circumstances to perform the tail recursion elimination optimization described in Section 3.2.2 using activation records stored on the stack. Assume that, as in the text, procedure A has called B, and B is about to call C, and will return the value yielded by C. Thus, the objective is to eliminate the activation record for B from the stack, and ensure that when C returns it will pass control back directly to A.

   (a) In the activation record for B, what information is stored between the return address location and the bottom of the activation record?

   (b) What information is stored by B in the activation record for C if we do a normal procedure call?

   (c) To implement this optimization, we want to replace the call instruction in B that automatically pushes the return address on to the stack with a simple unconditional transfer of control to the beginning of C. To do so, what must be the relationship between the size and number of parameters that were passed to B, and the size and number of parameters that B will pass to C?

   (d) How will the function invocation code generated in B for the call on C differ from the normal function invocation code?

7. Explain why the tail recursion elimination optimization described in the previous question is trivial in the case of recursive procedures with no arguments. Explain why this is true even if the procedure being optimized does not return any argument value.