CHAPTER SEVEN
7. RUN TIME ENVIRONMENTS

7.1. Introduction

Before considering code generation, we need to relate the static source text of a program to the actions that must occur at run time to implement the program. As execution proceeds, the same name in the source text can denote different data objects in the target machine.

The allocation and deallocation of data objects is managed by the *run-rime support* package, consisting of routines loaded with the generated target code. The design of the run-time support package is influenced by the semantics of procedures.

Each execution of a procedure is referred to as an activation of the procedure. If the procedure is recursive, several of its activations may & alive at the same time. The representation of a data object at run time is determined by its type. Often, elementary data types, such as characters, integers, and reals can be represented by equivalent data objects in the target machine. However, aggregates, such as arrays, strings, and structures, are usually represented by collections of primitive objects.

7.2. Source Language Issues

For specificity, suppose that a program is made up of procedures. This section distinguishes between the source text of a procedure and its activations at run time.

❖ Procedures

A *procedure* definition is a declaration that, in its simplest form, associates an identifier with a statement. The identifier is the *procedure name,* and the statement is the *procedure body*. Procedures that return values are called *function* in many languages; however, it is convenient to refer them as procedures. A complete program will also be treated as a procedure.

When a procedure name appears within an executable statement, we say that the procedure is *called* at that point. The basic idea is that a procedure call executes the procedure body. Note that procedure calls can also occur within expressions.

Some of the identifiers appearing in a procedure definition are special, and are called *formal parameters* (or just formals) of the procedure. (C calls them "formal arguments" and Fortran calls them "dummy arguments)". Arguments, known as actual parameters (or a*ctuals)* may be passed to a called procedure; they are substituted for the formals in the body.

❖ The Scope of a Declaration

A declaration in a language is a syntactic construct that associates information with a name. Declarations may be explicit, as in the Pascal fragment *var i : integer;* or they may be Implicit. For example, any variable name starting with I is assumed to denote an integer in a FORTRAN program, unless otherwise declared.

There may be independent declarations of the same name in different parts of a program. The *scope rules* of a language determine which declaration of a name applies when the name appears in the text of a program.

The portion of the program to which a declaration applies is called the *scope* of that declaration. An occurrence of a name in a procedure is said to be *local* to the procedure if it is in the scope of a declaration within the procedure; otherwise, the occurrence is said to be *nonlocal*. The distinction between local and nonlocal names carries over to any syntactic construct that can have declarations within it.

While scope is a property of the declaration of a name, it is sometimes convenient to use the abbreviation "the scope of name x" for "the scope of the declaration of name x that applies to this occurrence of x".

At compile time, the symbol table can be used to find the declaration that applies to an occurrence of a name. When a declaration is seen, a symbol table entry is created for it. As long as we are in the scope of the declaration, its entry is returned when the name in it is looked up.

❖ **Binding of Names**

Even if each name is declared once in a program, the same name may denote different data objects at run time. The informal term "data object" corresponds to a storage location that can hold values.

In programming language semantics, the term *environment* refers to a function that maps a name to a storage location, and the term *state* refers to a function that maps a storage location to the value held there. Using the terms *l*-value and *r*-value, an environment maps a name to an *l*-value, and a state maps the *l*-value to an *r*-value.

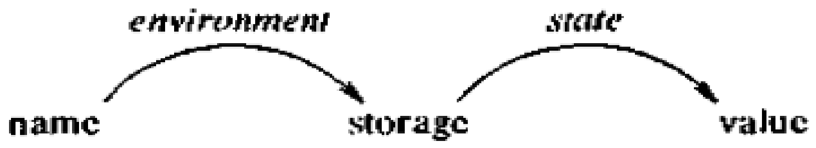
$$\begin{tikzcd}\text{name} \arrow[r, bend left, "environment"] & \text{storage} \arrow[r, bend left, "state"] & \text{value}\end{tikzcd}$$

Figure 7.1: Two-stage mapping from names to values.

Environments and states are different; an assignment changes the state, but not the environment. For example, suppose that storage address 100, associated with variable *pi*, holds 0. After the assignment *pi*:= 3. 14, the same storage address is associated with *pi*, but the value held there is 3.14.

When an environment associates storage location *s* with a name *x*, we say that *x is* bound to *s*; the association itself is referred to as a *binding* of x. A binding is the dynamic counterpart of a declaration. The term storage "location" is to be taken figuratively. If *x* is not of a basic type, the storage *s* for *x* may be a collection of memory words.

7.3. Storage Organization

❖ **Subdivision of Run-Time Memory**

Suppose that the compiler obtains a block of storage from the operating system for the compiled program to run in. This run-time storage might be subdivided to hold:

1. the generated target code,

2. data objects, and
3. a counterpart of the control stack to keep track of procedure activations.

The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area *Code*, usually in the low end of memory. Similarly, the size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called *Static*. One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code. In early versions of FORTRAN, all data objects could be allocated statically.

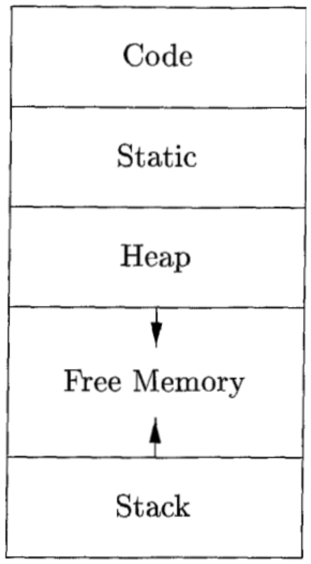

Figure 7.2: Typical subdivision of run-time memory into code and data areas

To maximize the utilization of space at run time, the other two areas, *Stack* and *Heap*, are at the opposite ends of the remainder of the address space. These areas are dynamic; their size can change as the program executes. The stack is used to store data structures called activation records that get generated during procedure calls.

The sizes of the stack and the heap can change as the program executes, so we show these at opposite ends of memory in Fig. 7.2, where they can grow toward each other as needed.

By convention, the heap grows towards higher, and stacks grow down. That is, the "top" of the stack is drawn towards the bottom of the page. Since memory addresses increase as we go down a page, "downwards-growing" means toward higher addresses. If *top* marks the top of the stack, offsets from the top of the stack can be computed by subtracting the offset from *top*. On many machines this computation can be done efficiently by keeping the value of *top* in a register. Stack addresses can then be represented as offsets from *top*.

❖ Activation Records

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an *activation record* or *frame*, consisting of the collection of fields shown in the following figure, Fig. 7.3. Not all languages, nor all compilers use all of these fields; often registers can take the place of one or more of them.

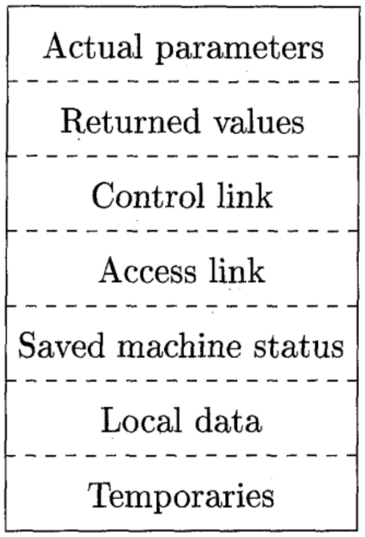

Figure 7.3: A general activation record

The purpose of the fields of an activation record is as follows.

1. Temporary values, such as those arising from the evaluation of expressions, are stored in the field for temporaries.
2. The field for local data holds data that is local to an execution of a procedure.
3. The field for saved machine status holds information about the state of the machine just before the procedure is called. This information includes the values of the program counter and machine registers that have to be restored when control returns from the procedure.
4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record. The optional *access* link is used to refer to nonlocal data held in other activation records.
5. The optional control *link* points to the activation record of the caller.
6. The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.
7. The field for the returned value is used by the called procedure to return a value to the calling procedure.

The sizes of each of these fields can be determined at the time a procedure is called. In fact, the sizes of almost all fields can be determined at compile time. An exception occurs if a procedure may have a local array whose size is determined by the value of an actual parameter, available only when the procedure is called at run time.

❖ **Compile-Time Layout of Local Data**

Suppose run-time storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory. On many machines, a byte is eight bits and some number of bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of the first byte.

The amount of storage needed for a name is determined from its type. An elementary data type, such as a character, integer, or real, can usually be stored in an integral number of bytes. Storage for an aggregate, such as an array or record, must be large enough to hold all its components. For easy access to the components, storage for aggregates is typically allocated in one contiguous block of bytes.

The field for local data is laid out as the declarations in a procedure are examined at compile time, Variable-length data is kept outside this field. We keep a count of the memory locations that have been allocated for previous declarations. From the count we determine a *relative* address of the storage for a local with respect to some position such as the beginning of the activation record. The relative address, or offset, is the difference between the addresses of the position and the data object.

7.4. Storage Allocation Strategies

A different storage-allocation strategy is used in each of the three data areas in the organization of Fig, 7.2.

1. Static allocation lays out storage for all data objects at compile time.
2. Stack allocation manages the run-time storage as a stack.
3. Heap allocation allocates and deallocates storage as needed at run time from a data area known as a heap.

❖ Static Allocation

In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package. Since the bindings do not change at run time, every time a procedure is activated, its names are bound to the same storage locations. This property allows the values of local names to be retained across activations of a procedure. That is, when control returns to a procedure, the values of the locals are the same as they were when control left the last time.

From the type of a name, the compiler determines the amount of storage to set aside for that name. The address of this storage consists of an offset from an end of the activation record for the procedure. The compiler must eventually decide where the activation records go, relative to the target code and to one another. Once this decision is made, the position of each activation record, and hence of the storage for each name in the record is fixed. At compile time we can therefore fill in the addresses at which the target code can find the data it operates on. Similarly, the addresses at which information is to be saved when a procedure call occurs are also known at compile time.

However, some limitations go along with wing static allocation alone.

- ➢ The size of a data object and constraints on its position in memory must be known at compile time.
- ➢ Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.
- ➢ Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

❖ Stack Allocation

Stack allocation is based on the idea of a control stack; storage is organized as a stack, and activation records are pushed and popped as activations begin and end, respectively. Storage for the locals in each call of a procedure is contained in the activation record for that call. Thus locals are bound to fresh storage in each activation, because a new activation record is pushed onto the stack when a call is made. Furthermore, the values of locals are deleted when the activation ends; that is, the values are lost because the storage for locals disappears when the activation record is popped.

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, however, the position of an activation record for a procedure is not known until run time. This position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register. Relative addresses in an activation record can be taken as offsets from any known position in the activation record.

❖ Heap Allocation

The stack allocation strategy discussed above cannot be used if either of the following is possible.

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

In each of the above cases, the deallocation of activation records need not occur in a last-in first out fashion, so storage cannot be organized as a stack. Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be deallocated in any order, so over time the heap will consist of alternate areas that are free and in use.

The question of efficient heap management is a somewhat specialized issue in data-structure theory. There is generally some time and space overhead associated with using a heap manager. For efficiency reasons, it may be helpful to handle small activation records or records of a predictable size as a special case, as follows:

1. For each size of interest, keep a linked list of free blocks of that size.
2. If possible, fill a request for size *s* with a block of size *s'*, where *s'* is the smallest size greater than or equal to *s*. When the block is eventually deallocated, it is returned to the linked list it came from.
3. For large blocks of storage use the heap manager.

This approach results in fast allocation and deallocation of small amounts of storage, since taking and returning a block from a linked list are efficient operations. For large amounts of storage we expect the computation to take some time to use up the storage, so the time taken by the allocator is often negligible compared with the time taken to do the computation.