Chapter 6

Modeling Objects

Overview

This chapter creates a framework for describing the semantics and implementation of objects so that the semantics actually used in any language can be understood and the advantages and drawbacks of the various implementations can be evaluated.

We assume the reader is familiar with the use of objects such as variables, constants, pointers, strings, arrays, and records. When we survey the popular programming languages, we see a great deal of commonality in the semantics of these things in all languages. There are also important differences, sometimes subtle, that cause languages to "feel" different, or require utterly different strategies for use. A program object embodies a real-world object within a program. The program object is stored in a storage object, a collection of contiguous memory cells. Variables are storage objects that store pure values; pointer variables store references.

Initialization and assignment are two processes that place a value in a storage object. Initialization stores a program object in the storage object when the storage object is created. Assignment may be destructive or coherent. Extracting the contents from a storage object is known as dereferencing. Assignment and dereferencing of pointer variables usually yield references to ordinary variables rather than pure values. Managing computer memory involves creating, destroying, and keeping storage objects available. Three strategies are static storage, stack storage, and heap storage.

Exhibit 6.1. Representing objects.

External object: a length of 2" by 4" lumber.
  Program object: a 32-bit floating-point value.
  Storage object: a memory location with four consecutive bytes reserved for this number.

External object: a charge account.
  Program object: a collection of values representing a customer's name, address, account number, billing date, and current balance.
  Storage object: a series of consecutive memory locations totaling 100 bytes.

6.1 Kinds of Objects

A program is a means of modeling processes and objects that are external to the computer. *External objects* might be numbers, insurance policies, alien invaders for a video game, or industrial robots. Each one may be modeled in diverse ways. We set up the model through declarations, allocation commands, the use of names, and the manipulation of pointers. Through these, we create objects in our programs, give them form, and describe their intended meaning. These objects are then manipulated by the functions and operators of a language.

We start by making a distinction between the memory location in which data is stored and the data itself. The ways of getting data into and out of locations are explored.

A *program object* is the embodiment of an object in the program. It may represent an external object, such as a number or a record, in which case it is called a *pure value*. It may also represent part of the computer system itself, such as a memory location, a file, or a printer. During execution, the program manipulates its program objects as a means of simulating meaningful processes on the external objects or controlling its own internal operations. It produces usable information from observed and derived facts about the program objects.

A program commonly deals with many external objects, each being represented by a pure value program object [Exhibit 6.1]. While all the external objects exist at once, their representing program objects can be passed through the computer sequentially and so do not have to be simultaneously present. For example, an accounting program deals with many accounts. Representations of these accounts are put in some sequence on an input medium and become program objects one at a time.

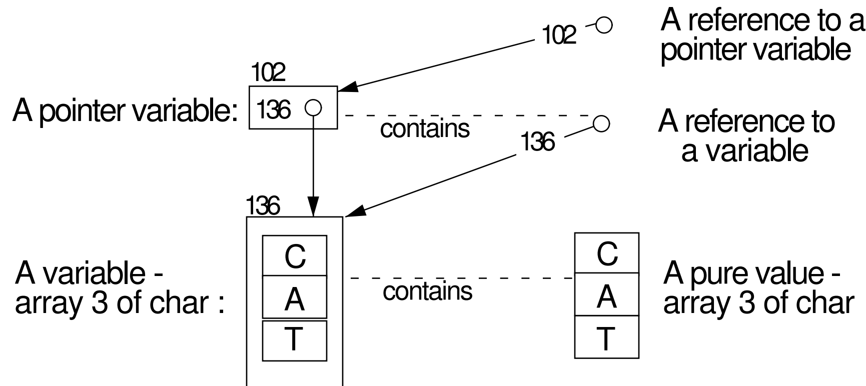
In order to manipulate program objects, the program must generally store all or part of a program object in memory. It uses a storage object for this purpose. A *storage object* is a collection of contiguous memory cells (bits, bytes, etc.) in which a program object, called its *value* or *contents*, can be stored.[1]

A *reference* is the memory address of a storage object and is the "handle" by which the object

[1]A storage object sometimes encompasses more cells than are needed to store the value. These cells, commonly added to achieve word alignment, are called padding.

Exhibit 6.2. Values, variables, and pointers.

The relationship between storage objects and program objects is illustrated. Boxes represent storage objects, letters represent pure values of type "character", small circles (o) from which arrows emerge represent references, and dotted lines represent dereferencing.



is accessed. In older terminology, a pure value is called an r-value or right-hand-value, because it can occur to the right of an assignment operator. A reference is called an l-value or left-hand-value. A reference is created when a storage object is allocated. This reference is itself a program object and may be stored in another storage object for later use [Exhibit 6.2]. A program must possess a reference to a program object in order to use that object.

The *allocation* process sets aside an area of unused computer memory to make a new storage object. The process is essentially the same whether it is being carried out by an interpreter, which does the allocation at run time when the command is interpreted, or by a compiler, which deals with addresses of the storage objects that will be allocated at some future time when the program is executed.

Allocation procedures are usually part of the implementation of a language, not part of the language definition, so the actual allocation process often differs from one translator to the next, as well as from one language to the next. Typically, though, the allocation process will include the following actions:

1. The translator must determine N, the number of bytes of memory that are needed. In some languages the programmer communicates this information by specifying the data type of the new object. Size, in bytes, is calculated by the translator and stored as part of the definition of a type. In lower-level languages the programmer specifies the allocation size explicitly.

2. A segment of free storage is located with length $L \geq N$. A reference to the first location in this segment is saved.

3. The address of the beginning of the free storage area is incremented by N, thus removing N

bytes from free storage.

4. If an initial value was defined, it is stored in the new storage object.
5. The address, or reference, saved in step 2 is returned as the result of the allocation process. It is the means by which the program is able to find the new storage object.

A *variable* is a storage object in which a pure value may be stored. Pure values and the variables in which they are stored have the same size and structure and are considered to be the same data type in many languages. We distinguish between them here because they have very different semantics. Operations you can perform with variables are to allocate and deallocate them and to fetch values from and store values into them. In contrast, pure values can be combined and manipulated with operators and functions, but not allocated and deallocated.

A *pointer variable* is a storage object, or part of a storage object, in which a reference may be stored. (Often this term will be shortened to "pointer".) Pointers are used to create storage structures such as game trees and linked lists and are an important means of modeling external objects.

6.2 Placing a Value in a Storage Object

6.2.1 Static Initialization

A storage object receives a value by one of two processes: initialization or assignment. Until a value is stored in a storage object, it is said to contain *garbage*, or to have an *undefined value*. (When we wish to indicate an undefined value we will write "?".)

Using an undefined value is a commonly made semantic error which generally cannot be detected by a language translator. For this reason some translators initialize all variables to zero, which is the most commonly useful initial value, or to some distinctive bit pattern, so that the semantic error can be more easily detected. It is poor programming practice to depend on such automatic initialization, however. Different translators for the same language may implement different initialization policies, and the program that depends on a particular policy is not portable.

Initialization stores a program object in the storage object when the storage object is created. Many languages permit the programmer to include an initializing clause in an object declaration. Typical declaration forms are shown in Exhibits 6.3 and 6.4. In each exhibit, declarations are given for an integer variable, a character string, and an array of real numbers, and initial values are declared for each.

Initializing compound objects, such as arrays and records, is restricted or not allowed in some languages. Two problems are involved here: how to denote a structured value, and how to implement initialization of dynamically allocated structured objects. The FORTRAN and C examples [Exhibits 6.3 and 6.4] illustrate two approaches to defining the structure of the initializer.

In FORTRAN, the programmer writes an explicit loop or nest of loops which specify the order in which the fields of an array will be initialized and then provides a series of constants that will

Exhibit 6.3. Initial value declarations in FORTRAN.

```
CHARACTER*3 EOFLAG
DIMENSION A (8)
DATA EOFLAG, ISUM / 'NO ', 0/, ( A(I), I=1,8) / 8.2, 2.6, 3.1, 17.0, 4 * 0.0 /
```

Notes:

- In FORTRAN, simple integers and reals may be declared implicitly. Explicit declarations must be given for arrays and strings.
- Initial values are given in separate DATA declarations which must follow the statements that declare the storage objects.
- A single DATA statement can initialize a list of objects. It must contain exactly as many initial values as fields to be initialized. Initial values may be repeated by using a repeat count with a '*'.
- An array may be initialized by giving a loop-controlling expression.

Exhibit 6.4. Initial value declarations in C.

```
static  char end_of_file_flag [ ] = "no ";
        int isum = 0;
static  float a[8] = {8.2, 2.6, 3.1, 17.0};
```

Notes:

- In C an initial value may be given as part of a variable declaration.
- Static arrays can be initialized by listing the correct number of values for the array enclosed in brackets. (The property "static" is explained in Section 6.3.)
- The programmer may omit the array length specifier from the declaration, as in the top line, and the length of the storage object will be deduced from the length of the initial value list.
- If too few initializers are given to fill an array, remaining elements are initialized to zero.

evaluate to the desired initial values. A repetition count can be specified when several fields are to be initialized to the same value. Part or all of an array may be initialized this way. This is a powerful and flexible method, but it does complicate the syntax and semantics of the language.

Contrast this to a C initializer. Its structure is denoted very simply by enclosing the initial values in brackets, which can be nested to denote a type whose fields are themselves structured types. The same simple syntax serves to initialize both records and arrays. Initializers can be constants or constant expressions; that is, expressions that can be evaluated at compile time. In some ways, this is not as flexible a syntax as FORTRAN provides. If any field of a C object is initialized, then all fields will be initialized. If the same nonzero value is to be placed in several fields, it must be written several times. The one shortcut available is that, if the initializer has too few fields, the remaining fields will default to an initial value of zero.

It is likely that the designers of C felt that FORTRAN initializers are too flexible—that they provide unnecessary flexibility, at the cost of unnecessary complication. Applying something akin to the principle of Too Much Flexibility, they chose to include the simpler, but still very useful, form in C.

All data storage in FORTRAN is created and initialized at load time. A translator can evaluate the constant expressions in an initializer and generate store instructions to place the resulting values into storage when the program code is loaded. Modern languages, though, support dynamic allocation of local variables in stack frames. (These are called "automatic" variables in C.) The initialization process for automatic variables is more complex than for static variables.

Suppose a function F contains a declaration and initializations for a local array, V. This array cannot be initialized at load time because it does not yet exist. The translator must evaluate the initializing expressions, store the values somewhere, and generate a series of store instructions to be executed every time F is called. These copy precomputed initial values into the newly allocated area. This process was considered complex enough that the original definition of C simply did not permit initialization of automatic arrays. ANSI C, however, supports this useful facility.

6.2.2 Dynamically Changing the Contents of a Storage Object

Destructive Assignment.

In many languages, one storage object can be used to store different program objects at different times. *Assignment* is an operation that stores a program object into an existing storage object and thus permits the programmer to change the value of a storage object dynamically. This operation is sometimes called *destructive assignment* because the previous contents of the storage object are lost. The storage object now represents a different external object, and we say that its *meaning* has changed.

Functional languages are an important current research topic. The goal of this research is to build a language with a clean, simple semantic model. Destructive assignment is a problem because it causes a change in the meaning of the symbol that names the storage object. It complicates a formal semantic model considerably to have to deal with symbols that mean different things at

Exhibit 6.5. Initializing and copying a compound object in Pascal.

Pascal declarations are given below for a record type named "person" and for two person-variables, a and b. In Pascal, compound objects cannot be initialized coherently, so three assignments are used to store a record-value into b. On the other hand, records *can* be assigned coherently, as shown in the last line, which copies the information from b to a.

```
TYPE person = RECORD age, weight:  integer; sex:  char END;
VAR a, b :  person;
BEGIN
    b.age := 10;
    b.weight := 70;
    b.sex := 'M';
    a := b;
    ...
END;
```

different times.

In a functional language, parameter binding is used in place of destructive assignment to associate names with objects. At the point that a Pascal programmer would store a computed value in a variable, the functional programmer passes that value as an argument to a function. The actions following the assignment in the Pascal program, and depending on it, would form the body of the function. A series of Pascal statements with assignment gets turned "outside in" and becomes a nest of function calls with parameter bindings.[2] This approach produces an attractive, semantically clean language because the parameter name has the same meaning from procedure entry to procedure exit.

Coherent Assignment. An array or a record is a compound object: a whole made up of parts which are objects themselves. Some but not all programming languages permit *coherent assignment* of compound objects. In such languages an entire compound variable is considered to be a single storage object, and the programmer can refer to the compound object as a whole and assign compound values to it [Exhibits 6.5 and 6.7].

In COBOL any kind of object could be copied coherently. It is even possible to use one coherent READ statement to load an entire data table from a file into memory. In most older languages, though, assignment can only be performed on simple (single-word) objects. An array or a record is considered to be a collection of simple objects, not a coherent large object. The abstract process of placing a compound program object into its proper storage object must be accomplished by a series of assignment commands that store its individual simple components.

[2] A deeply nested expression can look like a "rat's nest" of parentheses; deep nesting is avoided by making many short function definitions.

Exhibit 6.6. Initializing and copying a compound object in K&R C.

A record type named "person" is defined, and two person-variables, `a` and `b`, are declared. The variable `b` is initialized by the declaration and copied into `a` by the assignment statements.

The property "static" causes the variable to be allocated in the program environment rather than on the stack, so that it can be initialized at load time. K&R C did not support initialization of dynamically allocated structured objects.

```
typedef struct {int age, weight; char sex;} person;
static person a, b = {10, 70, 'M'};
{   a.age = b.age;
    a.weight = b.weight;
    a.sex = b.sex;
    ...}
```

An example of the lack of coherent assignment can be seen in the original Kernighan and Ritchie definition of C. Coherent assignment was not supported; to copy a record required one assignment statement for each field in the record. Thus three assignments would be required to copy the information from `b` to `a` in Exhibit 6.6. However, coherent initialization of record variables *was* supported, and `b` could be initialized coherently.

Even in languages that support coherent compound assignment, the programmer is generally permitted to assign a value to one part of the compound without changing the others. In such situations, care must always be taken to ensure that a compound storage object is not left containing parts of two different program objects!

Exhibit 6.7. Initializing and copying a compound object in ANSI C.

This example is written in ANSI C, which is newer than both K&R C and Pascal. The difference between this and the clumsier versions in Exhibits 6.5 and 6.6 reflects the growing understanding that coherent representations and operations are important.

The type and object declarations are the same in both versions of C, as are initializations. But compound objects can be assigned coherently in ANSI C, so only one assignment is required to copy the information from `b` to `a`. Further, dynamically allocated (automatic) structs may be initialized in ANSI C.

```
typedef struct {int age, weight; char sex;} person;
person a, b = {10, 70, 'M'};
{ a = b; ...}
```

Exhibit 6.8. Languages where assignment is a statement.

A "yes" in the third column indicates that compound objects (such as arrays and records) may be assigned coherently, as a single action. A "yes" in the fourth column indicates that one ASSIGN statement may be used to store a value in several storage objects.

Language	Assignment Symbol	Compound Assignment?	Multiple Assignment?
COBOL	MOVE	yes	yes
	= (in a COMPUTE statement)	no	yes
	ADD, SUBTRACT, MULTIPLY, DIVIDE	no	yes
FORTRAN	=	no	no
ALGOL	:=	no	no
PL/1	=	yes	yes
FORTH	!	no	no
Pascal	:=	yes	no
Ada	:=	yes	no

Assignment Statements versus Assignment as a Function. Assignment is invoked either by writing an explicit ASSIGN operator or by calling a READ routine. In either case, two objects are involved, a reference and a value. The reference is usually written on the left of the ASSIGN operator or as the parameter to a READ routine, and the value is written on the right of the ASSIGN or is supplied from an input medium.

Assignment is one of a very small number of operations that require a reference as an argument. (Others are binding, dereference, subscript, and selection of a field of a record.) The purpose of an assignment is to modify the information in the computer's memory, not to compute a new value. It is the only operation that modifies the value of existing storage objects. For this reason, ASSIGN and READ occur in many languages as statement types or procedures rather than as functions. Exhibit 6.8 lists the symbols and semantics for the ASSIGN statements in several common programming languages.

In other languages, ASSIGN is a function that returns a result and may, therefore, be included in the middle of an expression. Exhibit 6.9 shows ASSIGN *functions* in common programming languages. LISP returns the reference as the result of an assignment. C returns the value, so that it may be assigned to another storage object in the same expression or may be used further in computing the value of an enclosing expression. Exhibit 6.10 demonstrates how one assignment can be nested within another.

When ASSIGN returns a value, as in C, a single expression may be written which assigns that value to several storage objects. We call this *multiple assignment*. While this facility is not essential, it is often useful, especially when several variables need to be zeroed out at once. The same end is achieved in other languages, such as COBOL, by introducing an additional syntactic rule to allow

Exhibit 6.9. Languages where assignment is a function.

A "yes" in the third column indicates that compound objects (such as arrays and records) may be assigned coherently, as a single action.

Language	Assignment Symbol	Compound Assignment?	Result Returned
LISP	`replaca`, `replacd`	some versions	reference
APL	← (also used for binding)	yes	value
C (1973)	=	no	value
C (ANSI)	=	yes	value

an ASSIGN statement to list references to several storage objects, all of which will receive the single value provided.

6.2.3 Dereferencing

Dereferencing is the act of extracting the contents from a storage object. It is performed by the FETCH operation, which takes a reference to a storage object and returns its value. When a pointer variable is dereferenced, the result is another reference. This could be a reference to a variable, which itself could be dereferenced to get a pure value, or it could be a reference to another pointer, and so forth.

Whereas ASSIGN is always written explicitly in a language, its inverse, FETCH, is often invoked implicitly, simply by using the name of a storage object. Many languages (e.g., FORTRAN, Pascal, C, COBOL, BASIC, LISP) automatically dereference a storage object in any context where a program

Exhibit 6.10. Assignment as a function in C.

An array length is defined as a constant at the top of the program to facilitate modifications. Then the array "ar" is declared to have 100 elements, with subscripts from 0 to 99. Two integers are declared and set to useful numbers: `num_elements` holds the number of elements in the array, and `high_sub` holds the subscript of the last element.

```
#define MAXLENGTH 100
    float ar[ MAXLENGTH ];
    int high_sub, num_elements;
    high_sub = (num_elements = MAXLENGTH) - 1;
```

The last line contains two assignments. The constant `MAXLENGTH` is stored into the variable `num_elements`, and it is also returned as the result of the assignment function. This value is then decremented by one, and the result is stored in `high_sub`.

Exhibit 6.11. Dereferencing by context in Pascal.

We analyze the dereferences triggered by evaluating this expression:

```
xarray[ point_1↑.number ] := eval_function( point_2 ) ;
```

Assume the objects referenced have the following types:

`xarray:`	An array of unspecified type.
`point_1, point_2:`	Pointer to a record with a field called "`number`".
`eval_function:`	A function taking one pointer parameter and returning something of the correct type to be stored in `xarray`.

A variety of dereference contexts occur. Contexts (1), (3), and (4) occur together on the left, as do contexts (2) and (5) on the right.

Reference	Is it dereferenced here?
`xarray`	No, it is on the left of a `:=` operator.
`point_1`	Yes, explicitly, by the ↑ operator. Although this is part of a subscript expression, explicit dereference must be used because pointer variable names are not dereferenced in a pointer expression.
`point_2`	You cannot tell from this amount of context. It will not be dereferenced if the function *definition* specifies that it is a VAR parameter. If VAR is not specified, it will be automatically dereferenced.

object is required. Thus a variable name written in a program sometimes "means" a reference and sometimes a pure value, depending on context. This introduces complexity into a language. You cannot just see a symbol, as in lambda calculus, and know what it means. You must first examine where it is in the program and how it is used. To define the dereferencing rules of a language, contexts must be enumerated and described. The commonly important contexts are:

1. The left-hand side of an assignment operator.
2. The right-hand side of an assignment operator.
3. Part of a subscript expression.
4. A pointer expression.
5. A parameter in a function or procedure call.

Note that these contexts are not mutually exclusive but can occur in a confusing variety of combinations, as shown in Exhibit 6.11. Many other combinations of dereferencing contexts are, of course, possible.

Whether or not a reference is dereferenced in each context varies among languages. In context (1) dereferencing is never done, as a reference is required for an ASSIGN operation. But when a subscript expression (3) occurs in context (1), dereferencing will happen within the subscript part

Exhibit 6.12. Explicit dereferencing in FIG FORTH.

All FORTH expressions are written in postfix form, so you should read and interpret the operators from left to right. The FETCH operator is "@". It is written following a reference and extracts the contents from the corresponding storage object.

On lines 1 and 2, variables named XX and Y are declared and initialized to 13 and 0, respectively. Line 3 dereferences the variable XX and multiplies its value by 2. The result is stored in Y, which is not dereferenced because a reference is needed for assignment.

```
1   13 VARIABLE XX
2   0 VARIABLE Y
3   XX @ 2 * Y ! ( Same as Y = XX * 2 in FORTRAN. )
```

The expression `XX 2 *` would multiply the address, rather than the contents, of the storage object named XX by 2.

---

of the expression (the subscripted variable itself will not be dereferenced). In contexts (2) and (3) most languages will automatically dereference, as long as the situation does not also involve context (4).

In context (4) languages generally do not dereference automatically. They either provide an explicit FETCH operator or combine dereferencing with other functions. Examples of FETCH operators are the Pascal "↑" and C "*". Examples of combined operators are "->" in C, which dereferences a pointer and then returns a reference to a selected part of the resulting record, and "car" and "cdr" in LISP, which select a part of a record and then dereference it.

In context (5), there is no uniformity at all among languages. The particular choices and mechanisms used in various languages are discussed fully in Chapter 8, Section 8.4, and Chapter 9, Section 9.2.

There are also languages in which storage objects are *never* automatically dereferenced, the most common being FORTH. In such languages the dereference command must be written explicitly using a dereference operator ("@" in FORTH) [Exhibit 6.12]. The great benefit of requiring explicit dereference is simplicity. A variable name always means the same thing: a reference. Considering the kind of complexity (demonstrated above) that is inherent in deriving the meaning of a reference from context, it is easy to understand the appeal of FORTH's simple method. The drawback of requiring explicit dereference is that an additional symbol must be written before most uses of a variable name, adding visual clutter to the program and becoming another likely source of error because dereference symbols are easily forgotten.

6.2.4 Pointer Assignment

Pointer assignment is ordinary assignment where the required reference is a reference to a pointer variable and the value is itself a reference, usually to an ordinary variable. Languages that support pointer variables also provide a run-time allocation function that returns a reference to the newly

Exhibit 6.13. Pointer assignments in Pascal.

We assume the initial state of storage shown in Exhibit 6.14.

```
TYPE list = ↑cell;
    cell = RECORD value:char; link:list END;
VAR P1, P2, P3:  list;
```

Code	Comments
P2 := P1;	Dereference P1 and store its value in P2.
P3 := P1 ↑.link;	Dereference P1, select its link field, which is a pointer variable, and dereference it. Store the resulting reference in P3.

`P1, P2`, and `P3` all share storage now. We can refer to the field containing the `%` as `P2↑.link↑.value` or as `P3↑.value`. Note that a pointer must be explicitly dereferenced, using ↑, before accessing a field of the object to which it points.

allocated storage. This reference is then assigned to a pointer variable, which is often part of a compound storage object. Pointer assignment allows a programmer to create and link together simple storage objects into complex, dynamically changing structures of unlimited size.

Multiple pointers may be attached to an object by pointer assignment. The program object of a pointer is a reference to another storage object. When the pointer assignment `P2 := P1` is executed, the program object `P1`, which is a reference to some object, Cell1, is copied into the storage object of `P2`, thus creating an additional pointer to Cell1 and enabling `P2` as well as `P1` to refer to Cell1. Thus two objects now store references to one storage object, and we say they "share" storage dynamically. This is illustrated in Exhibits 6.13 and 6.14.

While such sharing is obviously useful, it creates a complex situation in which the contents of the storage structure attached to a name may change without executing an assignment to that name. This makes pointer programs hard to debug and makes mathematical proofs of correctness very hard to construct. Many programmers find it impossible to construct correct pointer programs

Exhibit 6.14. Pointer structures sharing storage.

Storage, diagrammed before and after the pointer assignments in Exhibit 6.13.

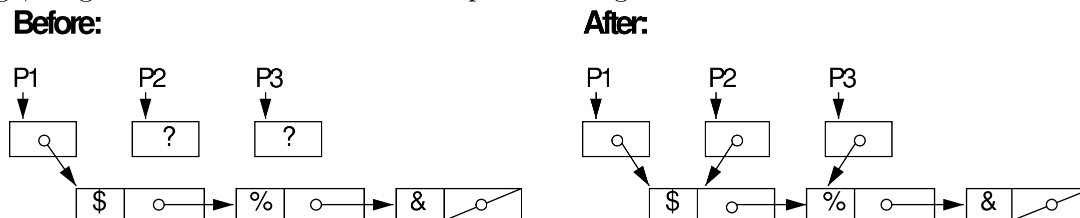


Exhibit 6.15. Pointer assignments with dereference in C.

The right side of the assignment is dereferenced if it evaluates to a structure or a simple object.

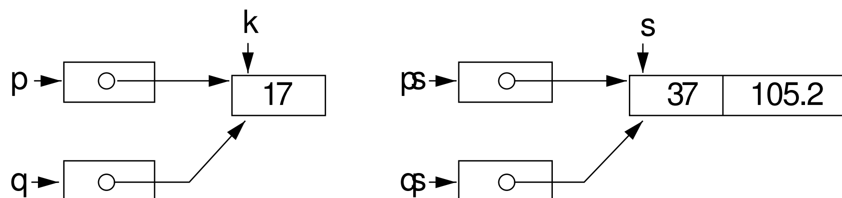
```
typedef struct { int age; float weight; } body;
body s;                          /* A variable of type 'body'.  */
body *ps, *qs;                   /* Two pointers to bodies.  */
int k;                           /* An integer variable.  */
int *p, *q;                      /* Two pointers to integers.  */

p = &k;    /* Store the address of k in p, that is, make p point at k.
              Note:  the & operator prevents automatic dereferencing.  */

ps = &s;   /* Make ps point at s.  */

q = p;     /* Dereference p to get the address stored in p, and store
              that address in q, making q point at the same thing as p.  */

qs = ps;   /* Make qs point at the same thing as ps.  */
```



without making diagrams of their storage objects and pointer variables.

6.2.5 The Semantics of Pointer Assignment

There are two likely ways in which a pointer assignment could be interpreted: with and without automatic dereferencing of the right-hand side. Pascal does dereference, as is shown in Exhibit 6.13. In such a language the statement `Q := P` is legal if `P` and `Q` are both pointers. This makes `Q` point at whatever `P` is pointing at. The assignment `P := K` is *illegal* if `P` is a pointer and `K` is an integer. Exhibit 6.15 shows several pointer assignments in C where the right side is dereferenced.

In a hypothetical language, ":=" *could* be defined such that the assignment "`p := k`" would be legal and would make `p` point at `k`. In this case, pointer assignment is interpreted without dereferencing the right side. In such a language we could create a chain of pointers as follows:

```
k := 5.4;    -- k is type float.
p := k;      -- p must be type pointer to float.
q := p;      -- q must be type pointer to pointer to float.
```

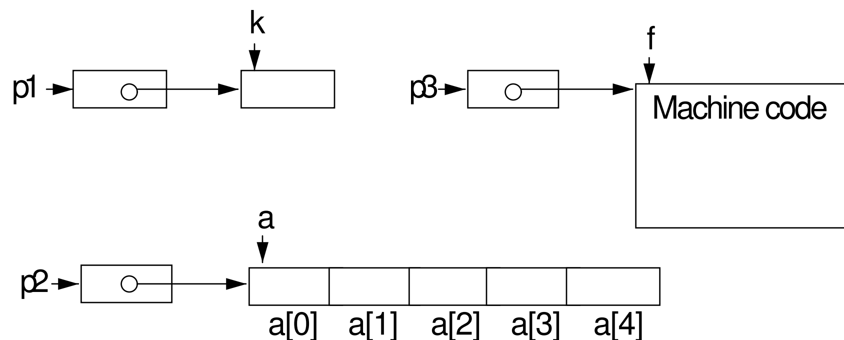
These assignments, taken together, would construct a pointer structure like this:

Exhibit 6.16. Pointer assignment without dereference in C.

We declare an integer variable, `k`; integer pointers, `p1` and `p2`; an array of five integers, `a`; a function that returns an integer, `f`; and a pointer to a function that returns an integer, `p3`.

The right side of a C assignment is not dereferenced if it refers to an array or a function.

```
int k, *p1, *p2, a[5], f(), *p3();
p1 = &k;        /* Make p1 point at k.  */
p2 = a;         /* Make p2 point at the array.  Note absence of "&".*/
p2 = &a[0];     /* Make p2 point at the address of the zeroth element of the
                   array.  This has the same effect as the line above.  */
p3 = f;         /* Store a reference to the function f in pointer p3.
                   Note that f is not dereferenced.*/
```



Note that "`p2 = &a;`" is syntactically incorrect because the name of an array *means* the address of its zeroth element. One must either omit the "`&`" or supply a subscript.

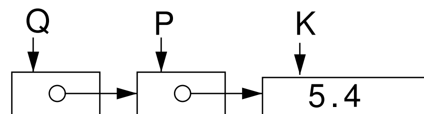


Exhibit 6.16 shows pointer assignments in C which set pointers to an array and a function. In these contexts, in C, the right side will not be dereferenced.

While either interpretation of pointer assignment could make sense, we would expect to see either one or the other used consistently in a language. One of the unusual and confusing facets of C is that the semantics of pointer assignment depends on the type of the expression on the right. If it denotes a simple object (such as an integer or a pointer) or an object defined as a struct, automatic dereferencing is used [Exhibit 6.15]. If the right-hand object is an array or a function, the second meaning, without dereferencing, is implemented [Exhibit 6.16].

6.3 The Storage Model: Managing Storage Objects

The differences among languages are easier to understand when the underlying mechanisms are known. A key part of any translator is managing the computer memory; storage objects must be created, kept available, and destroyed when appropriate. Three storage management strategies are in common use with all three present in some translators, but only one in others. These are static storage and two kinds of dynamic storage: stack storage and heap storage.

6.3.1 The Birth and Death of Storage Objects

A storage object is *born* when it is allocated, and it *dies* when it is no longer available for use by the program. The *lifetime*, or extent, of a storage object is the span of time from its birth to its death. An object that lives until the program is terminated is *immortal*. Most objects, however, die during program execution. It is a semantic error to attempt to reference a storage object after it has died. The run-time system will typically reuse the formerly occupied storage for other purposes, so references to a dead object will yield unpredictable results.

Deallocation is the recycling process by which dead storage objects are destroyed, and the storage locations they occupied are made available for reuse by the allocation process. Deallocation happens sometime, often not immediately, after death.

All live objects must be simultaneously in the computer's virtual memory. Real computers have limited memory, so it is important that the lifetimes of objects correspond to the period of time during which they are actually needed by the program. By having an object die when it is no longer useful, we can recycle the storage it formerly occupied. This enables a program to use a larger number of storage objects than would otherwise fit into memory.

Static Storage Objects

A compiler plans what storage objects will be allocated to a program at load time, and when the object code will be copied into computer memory, linked, and made ready to run. Such objects are allocated before execution begins and are immortal. These are called *static* storage objects because they stay there, unmoved, throughout execution. Static allocation is often accompanied by initialization. The compiler chooses run-time locations for the static objects and can easily put initial values for these locations into the object code.

The number of static storage objects in a program is fixed throughout execution and is equal to the number of static names the programmer has used. Global variables are static in any language. Some languages (for example, COBOL) have only static objects, while others (for example, Pascal) have no static storage except for globals. Still others (ALGOL, C) permit the programmer to declare that a nonglobal object is to be static. In ALGOL, this is done by specifying the attribute "OWN" as part of a variable declaration. In C, the keyword "static" is used for this attribute.

A language with only static storage is limiting. It cannot support recursion, because storage must be allocated and exist simultaneously for the parameters of a dynamically variable number of calls on any recursive function.

A language that limits static storage to global variables is also limiting. Many complex applications can be best modeled by a set of semi-independent functions. Each one of these performs some simple well-defined task such as filling a buffer with data or printing out data eight columns per line. Each routine needs to maintain its own data structures and buffer pointers. Ideally, these are private structures, protected from all other routines. These pointers cannot be ordinary local variables, since the current position on the line must be remembered from one call to the next, and dynamically allocated variables are deallocated between calls. On the other hand, these pointers should not be global, because global storage is subject to accidental tampering by unrelated routines. The best solution is to declare these as static local storage, which simultaneously provides both continuity and protection.

Finally, the unnecessary use of static objects, either global or local, is unwise because they are immortal. Using them limits the amount of storage that can be recycled, thereby increasing the overall storage requirements of a program.

Dynamic Storage Objects

Storage objects that are born during execution are called *dynamic*. The number of dynamic storage objects often depends on the input data, so the storage for them cannot be planned by the compiler in advance but must be allocated at run time. The process of choosing where in memory to allocate storage objects is called *memory management*. A memory manager *must* be sure that two storage objects that are alive at the same time never occupy the same place in memory. It *should* also try to use memory efficiently so that the program will run with as small an amount of physical memory as possible.

Memory management is a very difficult task to do well, and no single scheme is best in all circumstances. The job is considerably simplified if the memory manager knows something in advance about the lifetimes of its storage objects. For this reason, languages typically provide several different kinds of dynamic storage objects which have different lifetime patterns.

The simplest pattern is a totally unrestricted lifetime. Such an object can be born and die at any time under explicit control of the programmer. Nothing can be predicted about the lifetimes of these objects, which are generally stored in an area of memory called the *heap*. Whenever a new one is born, the storage manager tries to find a sufficiently large unused area of heap memory to contain it. Whenever the storage manager learns of the death of a heap object, it takes note of the fact that the memory is no longer in use.

There are many problems in recycling memory. First of all, the blocks in use may be scattered about the heap, leaving many small unused "holes" instead of one large area. If no hole is large enough for a new storage object, then the new object cannot be created, even though the total size of all of the holes is more than adequate. This situation is called *memory fragmentation*.

Second, a memory manager must keep track of the holes so that they can be located when needed. A third problem is that two or more adjacent small holes should be combined into one larger one. Different heap memory managers solve some or all of these problems in different ways. We will talk about some of them later in this chapter.

Because of the difficulty in managing a heap, it is desirable to use simpler, more efficient but restricted memory managers whenever possible. One particularly common pattern of lifetimes is called *nested lifetimes*. In this pattern, any two objects with different lifetimes that exist at the same time have well-nested lifetimes; that is, the lifetime of one is completely contained within the lifetime of the other. This pattern arises from block structure and procedure calls.

Storage for local block variables and procedure parameters only needs to exist while that block or procedure is active. We say that a block is *active* when control resides within it or within some procedure called from it. A storage object belonging to a block can be born when the block begins and die when the block ends, so its lifetime coincides with the time that the block is active. Blocks can be nested, meaning that a block B that starts within a block A finishes before A does. It follows that the lifetimes of any storage objects created by B are contained within the lifetimes of objects created by A.

Dynamic Stack Storage

Storage for objects with nested lifetimes can be managed very simply using a *stack*, frequently called the *run-time stack*. This is an area of memory, like the heap, on which storage objects are allocated and deallocated. Since, in the world of nested lifetime objects, younger objects always die before older ones, objects can always be allocated and deallocated from the top of the stack. For such objects, allocation and deallocation are very simple processes. The storage manager maintains a stack allocation pointer which indicates the first unused location on the stack. When a program block is entered, this pointer is incremented by the number of bytes required for the new storage object(s) during the allocation process. Deallocation is accomplished at block exit time by simply decrementing the stack allocation pointer by the same number of bytes. This returns the newly freed storage to the storage pool, where it will be reused.

In languages that support both heap and stack storage objects, the stack objects should be used wherever possible because their lifetime is tied to the code that uses them, and the birth and death processes are very efficient and automatic. (This is the reason that stack-allocated objects are called "`auto`" in `C`.)

Storage managers typically use stack storage for a variety of purposes. When control enters a new program block, a structure called a *stack frame*, or *activation record*, is created on the top of the stack. The area past the end of the current stack frame is used for temporary buffers and for storing intermediate results while calculating long arithmetic expressions [Exhibit 6.17, right side].

A stack frame[3] includes several items: parameters, local variables, the return address, and the return value (if the block is a function body). It also contains two pointers, called the static link and dynamic link [Exhibit 6.17, left side].

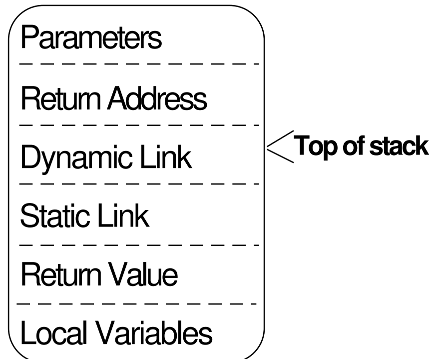
Let us define the *lexical parent* of a block to be that block which encloses it on the program listing. The lexical parent of the outermost block or blocks is the system. A *lexical ancestor* is a parent or the parent of a parent, and so on. The *static link* points to the stack frame of the current

[3]The rest of this section explains the structure of the stack for a lexically scoped, block-structured language.

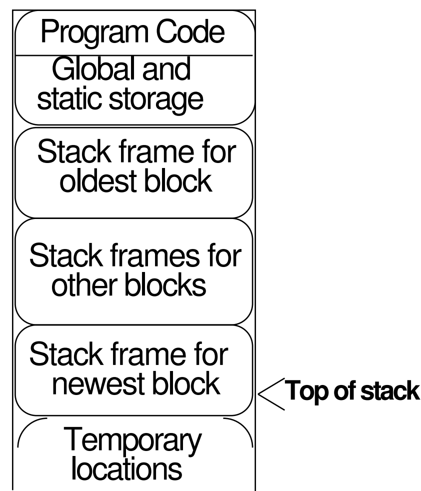
Exhibit 6.17. The structure of the run-time stack.

The order of the parts within a stack frame is arbitrary, as is the relationship of global storage and program code to the stack. The diagrams indicate a functional arrangement of the necessary kinds of information.

A single stack frame:



The program and stack at run time:



block's lexical parent. At run time, these links form a chain that leads back through the stack frames for all the blocks that lexically enclose the current block. Since the location of a lexical ancestor's frame is not predictable at compile time, the chain of static links must be followed to locate a storage object that was allocated by an ancestor. This is, of course, not as efficient as finding a local object, and it is one good reason to use parameters or local variables wherever possible.

The *dynamic parent* of a block is the block which called it during the course of execution and to which it must return at block exit time. The *dynamic link* points to the stack frame of the current block's dynamic parent. This link is used to pop the stack at block exit time.

The static and dynamic links are created when the stack frame is allocated at run time. During this process, several things are entered into the locations just past the end of the current frame. This process uses (and increments) the *local-allocation pointer* which points to the first free location on the stack. Before beginning the call process, this pointer is saved. The saved value will be used later to pop the stack. The sequence of events is as follows:

1. The calling program puts the argument values on the stack using the local-allocation pointer. Typically, the last argument in the function call is loaded on the stack first, followed by the second-last, and so on. The first argument ends up at the top of the stack.
2. The return address is written at the top of the stack, above the first argument.
3. The current top-of-stack pointer is copied to the top of the stack. This will become the new dynamic link field. The address of this location is stored into the top-of-stack pointer.
4. The static link for the new frame is written on the stack. This is the same as either the static link or the dynamic link of the calling block. Code is generated at compile time to copy the appropriate link.
5. The local allocation pointer is incremented by enough locations to store the return value and the local variables. If the locals have initializers, those values are also copied.
6. Control is transferred to the subroutine.

At block exit time, the stack frame must be deallocated. In our model, the return value is in the frame (rather than in a register), so the frame must be deallocated by the calling program. To do this, the value in the dynamic link field of the subroutine's frame is copied back into the top-of-stack pointer, and the local-allocation pointer is restored to its value prior to loading the arguments onto the stack.

Stack storage enables the implementation of recursive functions by permitting new storage objects to be allocated for parameters and local variables each time the function is invoked. An unlimited number of storage objects which correspond to each parameter or local name in the recursive function can exist at the same time: one set for every time a recursive block has been entered but not exited [Exhibits 6.18 and 6.19]. Each time a recursive procedure exits, the corresponding stack frame is deallocated, and when the original recursive call returns to the calling program, the last of these frames dies. The number of storage objects simultaneously in existence for a recursive program is limited only by the program logic and the amount of storage available for stack allocation, not by the number of declared identifiers in the program.

Dynamic Heap Storage

There are situations in which heap storage must be used because the birth or death patterns associated with stack storage are too restrictive. These include cases in which the size or number of storage objects needed is not known at block entry time and situations in which an object must outlive the block in which it was created.

Heap Allocation. *Heap allocation* is invoked by an explicit allocation command, which we will call ALLOC. Such commands can occur anywhere in a program, unlike local variable declarations which are restricted to the beginning of blocks. Thus a heap-allocated object can be born at any

Exhibit 6.18. A recursive function in Pascal.

The following (foolish) recursive function multiplies jj inputs together. Exhibit 6.19 traces an execution of this function.

```
FUNCTION product (jj:  integer):integer;
VAR kk:  integer;
BEGIN
    IF jj <= 0 THEN product := 1
    ELSE BEGIN
        readln(kk);
        product := kk * product(jj-1);
    END
END;
```

Exhibit 6.19. Stack frames for recursive calls.

If the function "product" in Exhibit 6.18, were called with the parameter 2, two recursions would happen. Assume the inputs 25 and 7 were supplied. Just before returning from the second recursion the stack would contain three stack frames as diagrammed. The "?" in a stack location indicates an undefined value.

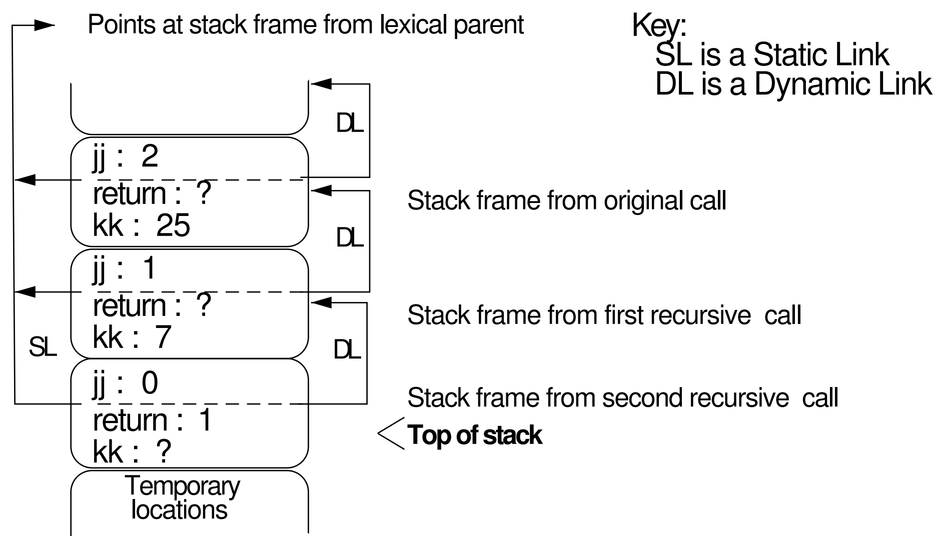


Exhibit 6.20. Dynamic allocation in FORTH.

- **Allocation:** `HERE ⟨expression⟩ ALLOT`

  Storage can be allocated dynamically in the "dictionary", which stores the symbol table and all global objects. The programmer is given access to the top-of-dictionary pointer through the system variable named HERE. The code above puts the current value of HERE on the stack. Then it evaluates the expression, which must produce an integer, N. Finally, ALLOT adds N bytes to the dictionary pointer. The address of the newly allocated area is left on the stack; the user must store it in a pointer variable.

- **Deallocation:** Users must write their own storage management routines if they wish to free and reuse dynamic storage.

time. ALLOC reserves storage in the heap and returns a reference to the new storage object. The allocation process for heap storage is somewhat more complicated than that for stack storage, since there may be two places to look in the heap for available storage. Initially there is only a large, empty area with an associated allocation pointer which is incremented (like the stack pointer) when storage is allocated from that area. After some objects have died, there may also be a *freelist* which contains references to these formerly used locations. Clearly, items on the freelist might be scattered all over the heap and be of quite varied sizes. The memory manager must contain algorithms to keep track of the sizes and merge adjacent free areas, and these algorithms must be fast to avoid degrading the performance of the system.

An ALLOC command takes some indication of the required size of the new object and finds and reserves that much memory. Either it returns a reference to the memory location, or it stores that reference in a pointer variable which thereafter gives access to the new object. The ways these actions are incorporated into current languages are truly varied [Exhibits 6.20 through 6.25]. The new storage object is used, later, by dereferencing this pointer, and it remains alive as long as the pointer or some copy of the pointer points at it.

Exhibit 6.21. Dynamic allocation in LISP.

- **Allocation:** `(cons ⟨expr1⟩ ⟨expr2⟩)`

  This allocates a new list cell and returns a reference to it. The left field of the cell is initialized to the result of evaluating ⟨expr1⟩ and its right field to ⟨expr2⟩.

- **Deallocation:** Most LISP systems rely on garbage collection.

Dead Heap Objects. We call a dead heap object *garbage*. Management of dead heap objects is very different from stack management. A heap object dies either when the last reference to the object is destroyed (let us call this a *natural death*) or when it is explicitly killed using a KILL command.

The run-time system of a language translator must manage heap storage allocation just as it manages stack frame allocation. However, when an object dies a natural death, both the programmer and the run-time system may be unaware of that death. A hole containing garbage is left in the heap at an unknown location.

A KILL command takes a reference to a storage object, kills the storage object, and puts the reference onto the freelist, where it can be recycled. In languages that implement KILL, programmers who use extensive amounts of dynamic storage are strongly urged to keep track of their objects and KILL them when they are no longer useful. (In general, this will be well before the object dies a natural death.) It is only through an explicit KILL command that the system can reclaim the storage.

Recycling a dead heap cell is more complex than recycling stack cells. The system cannot simply decrement the heap allocation pointer because, in general, dead objects are in the middle of the heap, not at the end. A data structure called a *freelist* is generally used to link together the recycled cells and provide a pool of cells available for future reuse. Conceptually, a freelist is just a list of reclaimed and reusable storage objects. However, that is not a simple thing to implement efficiently. The objects might all be interchangeable, or they might be of differing sizes. In any case, they are probably scattered all over heap memory. The language designer or system implementor must decide how to organize the freelist to maximize its benefits and minimize bookkeeping.

Ignore dead cells. The easiest implementation of KILL is to ignore it! Although this seems to be a misimplementation of a language, it has been done. The Pascal reference manual for the Data General MV8000 explicitly states that the Dispose command is implemented as a no-op. This compiler runs under the AOS-VS operating system, which is a time-shared, paged, virtual memory system.

The philosophy of the compiler writer was that most programs don't gobble up huge amounts of storage, and those that do can be paged. Old, dead storage objects will eventually be paged out. If all objects on the page have died, that page will never again be brought into memory. Thus the compiler depends on the storage management routines of the operating system to "deep-six" the garbage. This can work very well if objects with similar birth times have similar useful lifetimes. If not, each of many pages might end up holding a few scattered objects, vastly increasing the memory requirements of the process and degrading the performance of the entire system.

Keep one freelist. One possibility is to maintain a single list which links together all free areas. To do this, each area on the list must have at least enough bytes to store the size of the area and a link. (On most hardware that means 8 bytes.) Areas smaller than this are not reclaimable. Many or most C and Pascal compilers work this way.

Exhibit 6.22. Dynamic allocation in C.

- **Allocation:** In the commands that follow, `T` and `basetype` are types, `N` is an integer. The `malloc` function allocates one object of type `T` or size `N` bytes. `calloc` allocates an array of `N` objects of type `basetype` and initializes the entire array to zero. Both `malloc` and `calloc` return a reference to the new storage object. The programmer must cast that reference to the desired pointer type and assign it to some pointer variable.

```
malloc(sizeof(T))
malloc(N)
calloc(N, sizeof(basetype))
```

- **Deallocation:** `free(ptr);`

  `ptr` must be a pointer to a heap object that was previously allocated using `malloc` or `calloc`. That object is linked onto a freelist and becomes available for reuse.

A compiler could treat this 8-byte minimum object size in three ways. It could refuse to *allocate* anything smaller than 8 bytes; a request for a smaller area would be increased to this minimum. This is not as wasteful as it might seem. Those extra bytes often have to be allocated anyway because many machines require every object to start on a word or long-word boundary (a byte address that is divisible by 2 or 4).

Alternately, the compiler could refuse to *reclaim* anything smaller than the minimum. If a tiny object were freed, its bytes would simply be left as a hole in the heap. The philosophy here is that tiny objects are probably not worth bothering about. It takes a very large number of dead tiny objects to fill up a modern memory.

A fragmentation problem can occur with these methods for handling variable-sized dead objects.

Exhibit 6.23. Dynamic allocation in Pascal.

- **Allocation:** `New(⟨PtrName⟩);`

  The ⟨PtrName⟩ must be declared as a pointer to some type, say `T`. A new cell is allocated of type `T`, and the resulting reference is stored in the pointer variable.

- **Deallocation:** `Dispose(⟨PtrName⟩);`

  The object pointed at by `PtrName` is put onto the freelist.

With many rounds of allocation and deallocation, the average size of the objects can decrease, and the freelist may end up containing a huge number of tiny, worthless areas. If adjacent areas are not "glued together", one can end up with most of the memory free but no single area large enough to allocate a large object.

Joining adjacent areas is quick and easy, but one must first identify them. Ordinarily this would require keeping the freelist sorted in order of address and searching it each time an object is freed. This is certainly time-consuming, and the system designer must decide whether the time or the space is more valuable.

One final implementation of variable-sized deallocation addresses this problem. In this version, each allocation request results in an 8-byte header *plus* the number of bytes requested, rounded up to the nearest word boundary. At first this seems very wasteful, but using the extra space permits a more satisfactory implementation of the deallocation process.

The 8-byte header contains two pointers that are used to create a doubly linked circular list of dynamically allocated areas. One bit somewhere in the header is set to indicate whether the area is currently in use or free. The areas are arranged on this list in order of memory address. Areas that are adjacent in memory are adjacent in the list. Disposing of a dead object is very efficient with this implementation: one only needs to set this bit to indicate "free". Then if either of the neighboring areas is also free, the two can be combined into one larger area.

When a request is made for more storage, the list can be scanned for a free cell that is large enough to satisfy the new request. Scanning the list from the beginning every time would be very slow, since many areas that are in use would have to be bypassed before finding the first free area. But a scanning pointer can be kept pointing just past the most recently allocated block, and the search for a free area can thus start at the end of the in-use area. By the time the scanner comes back around to the beginning of the list, many of the old cells will have been freed. Thus we have a typical time/space trade-off. By allocating extra space we can reduce memory-management time.

Keep several freelists. A final strategy for managing free storage is to maintain one freelist for every size or type of storage object that can be freed. Thus all cells on the list are interchangeable, and their order doesn't matter. This simplifies reallocation, avoids the need for identifying adjacent areas, and, in general, is simpler and easier to implement. This reallocation strategy is used by Ada and Turing [Exhibits 6.24 and 6.25].

One of the problems with heap-allocated objects is in knowing when to kill them. It is all too easy to forget to kill an object at the end of its useful lifetime or to accidentally kill it too soon. This situation is complicated by the way in which pointer structures may share storage. A storage object could be shared by two data structures, one of which is no longer useful and apparently should be killed, while the other is still in use and must not be killed. If we `KILL` this structure we create a dangling pointer which will eventually cause trouble. Identifying such situations is difficult and error prone, but omitting `KILL` instructions can increase a program's storage requirements beyond what is readily available.

For this reason, some languages, such as LISP, automate the process of recycling dead heap

Exhibit 6.24. Dynamic allocation in Ada.

- **Allocation:** `NEW ⟨type⟩ ' (⟨expression⟩)`

  This allocates an object of the type requested. If the optional expression is supplied, it is evaluated and the result is used to initialize the new storage object. `NEW` is a function that returns a reference to the new object. The programmer must assign this reference to a variable of an `ACCESS` type (a pointer).

- **Deallocation:** Explicit deallocation is not generally used in Ada. In most Ada implementations, the dynamically allocated cells in a linked structure are automatically deallocated when the stack frame containing the pointer to the beginning of the structure is deallocated. Some Ada implementations contain full garbage collectors, like LISP.

  When it is necessary to recycle cells explicitly, a programmer may use a generic package named `Unchecked_Deallocation`.[4] This package must be *instantiated* (expanded, like a macro, at compile time) for each type of cell that is to be deallocated. Each instantiation produces a procedure, for which the programmer supplies a name, that puts that kind of cell on a freelist. (Different cell types go on different freelists.) Use of this facility is discouraged because it may lead to dangling pointers.

Exhibit 6.25. Dynamic allocation in Turing.

- **Allocation:** `new ⟨collection⟩, ⟨ptr⟩`

  To dynamically allocate cells of a particular type, the programmer must explicitly declare that the type forms a "collection". The `new` command allocates one cell from the desired collection and stores the reference in ⟨ptr⟩.

- **Deallocation:** `free ⟨collection⟩, ⟨ptr⟩`

  The object pointed at by `ptr` is returned to the collection it came from, where it will be available for reuse. The pointer object `ptr` is set to `nil`.

objects. In cases where a KILL command does not exist or is not used, heap objects still die, but the memory manager is not aware of the deaths when they happen. To actually recycle these dead heap cells requires a nontrivial mechanism called a *garbage collector*, which is invoked to recycle the dead storage objects when the heap becomes full or nearly full.

A garbage collector looks through storage, locating and marking all the live objects. It can tell that an object is alive if it is static or stack-allocated or if there is a pointer to it from some other live object anywhere in storage. The garbage collector then puts references to all of the unmarked, dead areas on the freelist, where the allocator will look for reusable cells.

While this scheme offers a lot of advantages, it is still incumbent on the programmer to destroy references to objects that are no longer needed. Furthermore, garbage collection is slow and costly. On the positive side, the garbage collector needs to be run only when the supply of free storage is low, which is an infrequent problem with large, modern memories. Thus garbage collection has become a practical solution to the storage management problem.

6.3.2 Dangling References

The case of a name or a pointer that refers to a dead object is problematical. This can happen with heap storage where the programming language provides an explicit KILL command. The programmer could allocate a heap-object, copy the resulting reference several times, then KILL the object and one of its references. The other references will still exist and point at garbage. These pointers are called *dangling references* or *dangling pointers*.

This situation could also arise if the program is able to store references to stack-allocated objects. Assume that a reference to a stack-allocated variable declared in an inner block could be stored in a pointer from an outer block. During the lifetime of the inner block, this can make good sense. When storage for the inner block is deallocated, though, the reference stored in the outer block becomes garbage. If it were then used, it would be an undefined reference.

Initially, a dangling reference points at the value of the deallocated variable. Later, when the storage is reused for another block, the address will contain useful information that is not relevant to the pointer. Thus the pointer provides a way of accessing and modifying some random piece of storage.

Serious errors can be caused by the accidental use of a dangling reference. Because the storage belonging to any inner block might be affected, the symptoms of this kind of error are varied and confusing. The apparent error happens at a point in the program that is distant from the block containing the dangling reference. If the inner blocks are modified, the symptoms may change; the part that was malfunctioning may start to work, and some other part may suddenly malfunction. This kind of error is extremely difficult to trace to its cause and debug.

Because of the potential severe problems involved, pointers into the stack are completely prohibited in Pascal. Pascal was designed to be simple and as foolproof as possible. The designer's opinion is that all programmers are occasionally "fools", and the language should provide as much protection as possible without prohibiting useful things.

Pascal completely prevents dangling pointers that point into the stack by prohibiting *all pointers*

to stack-allocated objects. The use of Pascal pointers is thus restricted to "heap" storage. Linked lists and trees, which require the use of pointers, are allocated in the heap. Simple variables and arrays can be allocated on the stack. Address arithmetic is not defined. Although this seems like a severe restriction, its primary bad effect is that subscripts must be used to process arrays, rather than the more efficient indexing methods which use pointers and address arithmetic.

In contrast, the use of pointers is not at all restricted in C. The "`&`" operator can be used freely and lets the programmer point at any object, including stack objects that have been deallocated. When control leaves an inner block, and its stack frame is deallocated, any pointer that points into that block will contain garbage. (An example of such code and corresponding diagrams are given in Exhibits 9.25 and 9.26.)

A language, such as C, which permits unrestricted use of addresses must either forgo the use of an execution stack or cope with the problem of dangling references. Allocation of parameters and local variables on the execution stack is a simple and efficient method of providing dynamically expandable storage, which is necessary to support recursion. Alternatives to using a stack exist but have high run-time overhead.

The other possibility is to permit the programmer to create dangling references and make it the programmer's responsibility to avoid using them meaninglessly. A higher level of programming skill is then required because misuse of pointers is always possible. A high premium is placed on developing clean, structured methods for handling pointers.

One design principle behind the original C was that a systems programmer does not need a foolproof language but does need free access to all the objects in his or her programs. Permitting free use of pointers was also important in the original C because it lacked other important features. Since structured objects could not be passed coherently to and from subroutines, any subroutine that worked on a structure had to communicate with its calling program using a pointer to the structure.

In the new ANSI C, this weakness is changed but not eliminated. It permits coherent assignment of structures, but not arrays. Similarly, structures may be passed to and from functions without using pointers, but an array parameter is always passed by pointer. Thus if C had the same restriction on pointers that Pascal has, the language would be much less powerful, perhaps not even usable.

How, then, can Pascal avoid the need to have pointers to stack objects? It has two facilities that are missing in C:

- Compound stack-allocated objects are coherent. They can be operated on, assigned, compared, and passed as parameters coherently.
- References to objects can be passed as parameters by using the `VAR` parameter declarator. Unfortunately, returning a compound value from a function is not permitted in the standard language and must be accomplished by storing the answer in a `VAR` parameter.

Most standard algorithms and data structures can be coded easily within these restrictions, using the fact that compound objects are coherent. Some experts assert that Pascal is a "better" language

for these applications because the programmer does not need to exercise as much care.

On the other hand, there are, occasionally, situations in which the pointer restrictions in Pascal prevent the programmer from coding an algorithm at all, and others in which the code would have been much more efficient if the programmer had used pointers to stack objects. One might say that C is a "better" language for these applications.

Exercises

1. Define and explain the relationship among: external object, program object, storage object, and pointer object.

2. What is the difference between a variable and a pointer variable?

3. By what two means can a value be placed in a storage object? What is the difference between the two processes?

4. What is the difference between destructive assignment and coherent assignment?

5. What is multiple assignment? How is it used?

6. Why do languages that implement assignment as a function allow the programmer more flexibility than those that implement assignment as a statement?

7. Define dereferencing. Which abstract function implements it?

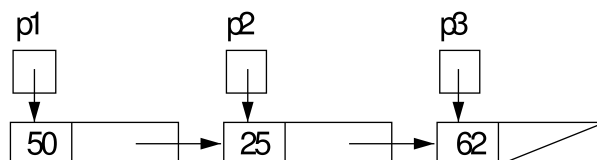
8. Choose a programming language with which you are familiar. Give a sample of code in which dereferencing is explicit. Give another in which it is implicit.

9. What language contexts must be enumerated in order to define the implicit dereferencing rules in a language?

10. Give some examples of FETCH operators which are used to dereference pointer expressions.

11. How does a pointer assignment differ from an ordinary assignment?

12. Given `p1` pointing at this list of integers, write legal pointer assignments for `p2` and `p3`.



13. In the C language, what are the meanings of "&" and "*" in pointer notation?

14. In C, what is the meaning of the name of an array? Do you need an "&" when assigning a pointer to an array? Why or why not?

15. What is the lifetime of a storage object?

16. What is the difference between a static storage object and a dynamic one?

17. Why is a language that only supports static storage items limiting?

18. What is memory management? Why is it important?

19. What is a run-time stack? A stack-frame? A heap?

20. What are the purposes of the static and dynamic links in a stack frame?

21. Name two languages in which local variables can be declared and are allocated each time a subroutine is entered. Give examples of local variable declarations.

22. Name a language in which local variables cannot be defined.

23. What is static local storage? In what ways is it better than global storage and ordinary local storage? Give an example, in some language, of a declaration that creates static local storage.

24. Suppose a language allows initial values to be specified for local variables, for example, the following declarations which define X and initialize it to 50:

Language	Declaration
Ada	`X: integer := 50;`
C	`int X=50;`

When and how often does initialization happen in the following two cases?

a. X is an ordinary local variable.

b. X is a static local variable.

25. Explain the differences in lifetime, accessibility, and creation time between:

a. An ordinary local variable.

b. A static local variable.

c. A global variable.

See page ??.

26. Name a language that does not support dynamic storage at all. (All storage is static.) Explain two ways in which this limits the power of the language.

27. What is the purpose of an ALLOC command? What is garbage? A freelist? What is the function of a KILL command?

28. Give examples in LISP and C of expressions that allocate nonstack (heap) storage dynamically.

29. Explain the reallocation strategy used by Turing. What are its advantages?

30. What is a dangling reference, and what problems can be caused by it?

31. Name a language in which pointers exist but can only point at dynamically allocated heap objects, not at objects allocated in the stack.

32. Name a language in which a pointer can point at any variable and pointer arithmetic is possible. Give an example of code.

33. Write a paragraph discussing the following questions: In what sense does FORTH or assembly language have pointers? What can be done with them? Are there any restrictions? Name two common errors that can occur with this kind of pointers.

34. Choose three languages from the list: APL, Ada, Pascal, C, FORTH, and assembler. What restrictions are there on the use of pointers in each? What effect do these restrictions have on flexibility and ease of use of the language? What effect do they have on the "safety" of code?