hashed access to elements. An array can have elements that are created with simple numeric indices and elements that are created with string hash keys.

In Lua, the table type is the only data structure. A Lua table is an associative array in which both the keys and the values can be any type. A table can be used as a traditional array, an associative array, or a record (struct). When used as a traditional array or an associative array, brackets are used around the keys. When used as a record, the keys are the field names and references to fields can use dot notation (record_name . field_name).

The use of Lua's associative arrays as records is discussed in Section 6.7.

C# and F# support associative arrays through a .NET class.

An associative array is much better than an array if searches of the elements are required, because the implicit hashing operation used to access elements is very efficient. Furthermore, associative arrays are ideal when the data to be stored is paired, as with employee names and their salaries. On the other hand, if every element of a list must be processed, it is more efficient to use an array.

6.6.2 Implementing Associative Arrays

The implementation of Perl's associative arrays is optimized for fast lookups, but it also provides relatively fast reorganization when array growth requires it. A 32-bit hash value is computed for each entry and is stored with the entry, although an associative array initially uses only a small part of the hash value. When an associative array must be expanded beyond its initial size, the hash function need not be changed; rather, more bits of the hash value are used. Only half of the entries must be moved when this happens. So, although expansion of an associative array is not free, it is not as costly as might be expected.

The elements in PHP's arrays are placed in memory through a hash function. However, all elements are linked together in the order in which they were created. The links are used to support iterative access to elements through the `current` and `next` functions.

6.7 Record Types

A **record** is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure.

There is frequently a need in programs to model a collection of data in which the individual elements are not of the same type or size. For example, information about a college student might include name, student number, grade point average, and so forth. A data type for such a collection might use a character string for the name, an integer for the student number, a floating-point for the grade point average, and so forth. Records are designed for this kind of need.

It may appear that records and heterogeneous arrays are the same, but that is not the case. The elements of a heterogeneous array are all references to data

objects that reside in scattered locations, often on the heap. The elements of a record are of potentially different sizes and reside in adjacent memory locations.

Records have been part of all of the most popular programming languages, except pre-90 versions of Fortran, since the early 1960s, when they were introduced by COBOL. In some languages that support object-oriented programming, data classes serve as records.

In C, C++, and C#, records are supported with the **`struct`** data type. In C++, structures are a minor variation on classes. In C#, structs are also related to classes, but are also quite different. C# structs are stack-allocated value types, as opposed to class objects, which are heap-allocated reference types. Structs in C++ and C# are normally used as encapsulation structures, rather than data structures. They are further discussed in this capacity in Chapter 11.Structs are also included in ML and F#.

In Python and Ruby, records can be implemented as hashes, which themselves can be elements of arrays.

The following sections describe how records are declared or defined, how references to fields within records are made, and the common record operations.

The following design issues are specific to records:

- What is the syntactic form of references to fields?
- Are elliptical references allowed?

6.7.1 Definitions of Records

The fundamental difference between a record and an array is that record elements, or **fields**, are not referenced by indices. Instead, the fields are named with identifiers, and references to the fields are made using these identifiers. Another difference between arrays and records is that records in some languages are allowed to include unions, which are discussed in Section 6.10.

The COBOL form of a record declaration, which is part of the data division of a COBOL program, is illustrated in the following example:

```
01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        05  FIRST   PICTURE IS X(20).
        05  MIDDLE  PICTURE IS X(10).
        05  LAST    PICTURE IS X(20).
    02  HOURLY-RATE PICTURE IS 99V99.
```

The `EMPLOYEE-RECORD` record consists of the `EMPLOYEE-NAME` record and the `HOURLY-RATE` field. The numerals `01`, `02`, and `05` that begin the lines of the record declaration are **level numbers**, which indicate by their relative values the hierarchical structure of the record. Any line that is followed by a line with a higher-level number is itself a record. The `PICTURE` clauses show the formats of the field storage locations, with `X(20)` specifying 20 alphanumeric characters and `99V99` specifying four decimal digits with the decimal point in the middle.

Ada uses a different syntax for records; rather than using the level numbers of COBOL, record structures are indicated in an orthogonal way by simply nesting record declarations inside record declarations. In Ada, records cannot be anonymous—they must be named types. Consider the following Ada declaration:

```
type Employee_Name_Type is record
   First : String (1..20);
   Middle : String (1..10);
   Last : String (1..20);
end record;
type Employee_Record_Type is record
   Employee_Name: Employee_Name_Type;
   Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

In Java and C#, records can be defined as data classes, with nested records defined as nested classes. Data members of such classes serve as the record fields.

As stated previously, Lua's associative arrays can be conveniently used as records. For example, consider the following declaration:

```
employee.name = "Freddie"
employee.hourlyRate = 13.20
```

These assignment statements create a table (record) named `employee` with two elements (fields) named `name` and `hourlyRate`, both initialized.

6.7.2 References to Record Fields

References to the individual fields of records are syntactically specified by several different methods, two of which name the desired field and its enclosing records. COBOL field references have the form

field_name `OF` record_name_1 `OF` **`...`** `OF` record_name_n

where the first record named is the smallest or innermost record that contains the field. The next record name in the sequence is that of the record that contains the previous record, and so forth. For example, the `MIDDLE` field in the COBOL record example above can be referenced with

```
MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
```

Most of the other languages use **dot notation** for field references, where the components of the reference are connected with periods. Names in dot notation have the opposite order of COBOL references: They use the name of the largest enclosing record first and the field name last. For example, the following is a reference to the field `Middle` in the earlier Ada record example:

```
Employee_Record.Employee_Name.Middle
```

C and C++ use this same syntax for referencing the members of their structures.

References to elements in a Lua table can appear in the syntax of record field references, as seen in the assignment statements in Section 6.7.1. Such references could also have the form of normal table elements—for example, `employee["name"]`.

A **fully qualified reference** to a record field is one in which all intermediate record names, from the largest enclosing record to the specific field, are named in the reference. Both the COBOL and the Ada example field references above are fully qualified. As an alternative to fully qualified references, COBOL allows **elliptical references** to record fields. In an elliptical reference, the field is named, but any or all of the enclosing record names can be omitted, as long as the resulting reference is unambiguous in the referencing environment. For example, `FIRST`, `FIRST OF EMPLOYEE-NAME`, and `FIRST OF EMPLOYEE-RECORD` are elliptical references to the employee's first name in the COBOL record declared above. Although elliptical references are a programmer convenience, they require a compiler to have elaborate data structures and procedures in order to correctly identify the referenced field. They are also somewhat detrimental to readability.

6.7.3 Evaluation

Records are frequently valuable data types in programming languages. The design of record types is straightforward, and their use is safe.

Records and arrays are closely related structural forms, and it is therefore interesting to compare them. Arrays are used when all the data values have the same type and/or are processed in the same way. This processing is easily done when there is a systematic way of sequencing through the structure. Such processing is well supported by using dynamic subscripting as the addressing method.

Records are used when the collection of data values is heterogeneous and the different fields are not processed in the same way. Also, the fields of a record often need not be processed in a particular order. Field names are like literal, or constant, subscripts. Because they are static, they provide very efficient access to the fields. Dynamic subscripts could be used to access record fields, but it would disallow type checking and would also be slower.

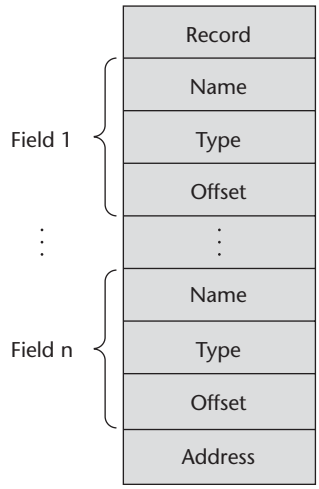
Records and arrays represent thoughtful and efficient methods of fulfilling two separate but related applications of data structures.

6.7.4 Implementation of Record Types

The fields of records are stored in adjacent memory locations. But because the sizes of the fields are not necessarily the same, the access method used for arrays is not used for records. Instead, the offset address, relative to the beginning of the record, is associated with each field. Field accesses are all handled using these offsets. The compile-time descriptor for a record has the general form shown in Figure 6.7. Run-time descriptors for records are unnecessary.

Figure 6.7

A compile-time descriptor for a record



6.8 Tuple Types

A tuple is a data type that is similar to a record, except that the elements are not named.

Python includes an immutable tuple type. If a tuple needs to be changed, it can be converted to an array with the `list` function. After the change, it can be converted back to a tuple with the `tuple` function. One use of tuples is when an array must be write protected, such as when it is sent as a parameter to an external function and the user does not want the function to be able to modify the parameter.

Python's tuples are closely related to its lists, except that tuples are immutable. A tuple is created by assigning a tuple literal, as in the following example:

```
myTuple = (3, 5.8, 'apple')
```

Notice that the elements of a tuple need not be of the same type.

The elements of a tuple can be referenced with indexing in brackets, as in the following:

```
myTuple[1]
```

This references the first element of the tuple, because tuple indexing begins at `1`.

Tuples can be catenated with the plus (`+`) operator. They can be deleted with the **`del`** statement. There are also other operators and functions that operate on tuples.

ML includes a tuple data type. An ML tuple must have at least two elements, whereas Python's tuples can be empty or contain one element. As in

Python, an ML tuple can include elements of mixed types. The following statement creates a tuple:

```
val myTuple = (3, 5.8, 'apple');
```

The syntax of a tuple element access is as follows:

```
#1(myTuple);
```

This references the first element of the tuple.

A new tuple type can be defined in ML with a type declaration, such as the following:

```
type intReal = int * real;
```

Values of this type consist of an integer and a real.

F# also has tuples. A tuple is created by assigning a tuple value, which is a list of expressions separated by commas and delimited by parentheses, to a name in a **let** statement. If a tuple has two elements, they can be referenced with the functions `fst` and `snd`, respectively. The elements of a tuple with more than two elements are often referenced with a tuple pattern on the left side of a **let** statement. A tuple pattern is simply a sequence of names, one for each element of the tuple, with or without the delimiting parentheses. When a tuple pattern is the left side of a **let** construct, it is a multiple assignment. For example, consider the following **let** constructs:

```
let tup = (3, 5, 7);;
let a, b, c  = tup;;
```

This assigns `3` to `a`, `5` to `b`, and `7` to `c`.

Tuples are used in Python, ML, and F# to allow functions to return multiple values.

6.9 List Types

Lists were first supported in the first functional programming language, LISP. They have always been part of the functional languages, but in recent years they have found their way into some imperative languages.

Lists in Scheme and Common LISP are delimited by parentheses and the elements are not separated by any punctuation. For example,

```
(A B C D)
```

Nested lists have the same form, so we could have

```
(A (B C) D)
```

In this list, `(B C)` is a list nested inside the outer list.

Data and code have the same syntactic form in LISP and its descendants. If the list `(A B C)` is interpreted as code, it is a call to the function `A` with parameters `B` and `C`.

The fundamental list operations in Scheme are two functions that take lists apart and two that build lists. The `CAR` function returns the first element of its list parameter. For example, consider the following example:

```
(CAR '(A B C))
```

The quote before the parameter list is to prevent the interpreter from considering the list a call to the `A` function with the parameters `B` and `C`, in which case it would interpret it. This call to `CAR` returns `A`.

The `CDR` function returns its parameter list minus its first element. For example, consider the following example:

```
(CDR '(A B C))
```

This function call returns the list `(B C)`.

Common LISP also has the functions `FIRST` (same as `CAR`), `SECOND`, . . . , `TENTH`, which return the element of their list parameters that is specified by their names.

In Scheme and Common LISP, new lists are constructed with the `CONS` and `LIST` functions. The function `CONS` takes two parameters and returns a new list with its first parameter as the first element and its second parameter as the remainder of that list. For example, consider the following:

```
(CONS 'A '(B C))
```

This call returns the new list `(A B C)`.

The `LIST` function takes any number of parameters and returns a new list with the parameters as its elements. For example, consider the following call to `LIST`:

```
(LIST 'A 'B '(C D))
```

This call returns the new list `(A B (C D))`.

ML has lists and list operations, although their appearance is not like those of Scheme. Lists are specified in square brackets, with the elements separated by commas, as in the following list of integers:

```
[5, 7, 9]
```

`[]` is the empty list, which could also be specified with `nil`.

The Scheme `CONS` function is implemented as a binary infix operator in ML, represented as `::`. For example,

```
3 :: [5, 7, 9]
```

returns the following new list: `[3, 5, 7, 9]`.

The elements of a list must be of the same type, so the following list would be illegal:

```
[5, 7.3, 9]
```

ML has functions that correspond to Scheme's `CAR` and `CDR`, named `hd` (head) and `tl` (tail). For example,

```
hd [5, 7, 9] is 5
tl [5, 7, 9] is [7, 9]
```

Lists and list operations in Scheme and ML are more fully discussed in Chapter 15.

Lists in F# are related to those of ML with a few notable differences. Elements of a list in F# are separated by semicolons, rather than the commas of ML. The operations `hd` and `tl` are the same, but they are called as methods of the `List` class, as in `List.hd [1; 3; 5; 7]`, which returns `1`. The `CONS` operation of F# is specified as two colons, as in ML.

Python includes a list data type, which also serves as Python's arrays. Unlike the lists of Scheme, Common LISP, ML, and F#, the lists of Python are mutable. They can contain any data value or object. A Python list is created with an assignment of a list value to a name. A list value is a sequence of expressions that are separated by commas and delimited with brackets. For example, consider the following statement:

```
myList = [3, 5.8, "grape"]
```

The elements of a list are referenced with subscripts in brackets, as in the following example:

```
x = myList[1]
```

This statement assigns `5.8` to `x`. The elements of a list are indexed starting at zero. List elements also can be updated by assignment. A list element can be deleted with `del`, as in the following statement:

```
del myList[1]
```

This statement removes the second element of `myList`.

Python includes a powerful mechanism for creating arrays called **list comprehensions**. A list comprehension is an idea derived from set notation. It first appeared in the functional programming language Haskell (see Chapter 15). The mechanics of a list comprehension is that a function is applied to each of the elements of a given array and a new array is constructed from the results. The syntax of a Python list comprehension is as follows:

[expression **for** iterate_var **in** array **if** condition]

Consider the following example:

```
[x * x for x in range(12) if x % 3 == 0]
```

The **range** function creates the array `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]`. The conditional filters out all numbers in the array that are not evenly divisible by `3`. Then, the expression squares the remaining numbers. The results of the squaring are collected in an array, which is returned. This list comprehension returns the following array:

```
[0, 9, 36, 81]
```

Slices of lists are also supported in Python.

Haskell's list comprehensions have the following form:

[body | qualifiers]

For example, consider the following definition of a list:

```
[n * n | n <- [1..10]]
```

This defines a list of the squares of the numbers from `1` to `10`.

F# includes list comprehensions, which in that language can also be used to create arrays. For example, consider the following statement:

```
let myArray = [|for i in 1 .. 5 -> (i * i) |];;
```

This statement creates the array `[1; 4; 9; 16; 25]` and names it `myArray`.

Recall from Section 6.5 that C# and Java support generic heap-dynamic collection classes, `List` and `ArrayList`, respectively. These structures are actually lists.

6.10 Union Types

A **union** is a type whose variables may store different type values at different times during program execution. As an example of the need for a union type, consider a table of constants for a compiler, which is used to store the constants found in a program being compiled. One field of each table entry is for the value of the constant. Suppose that for a particular language being compiled, the types of constants were integer, floating point, and Boolean. In terms of table management, it would be convenient if the same location, a table field, could store a value of any of these three types. Then all constant values could be addressed in the same way. The type of such a location is, in a sense, the union of the three value types it can store.

6.10.1 Design Issues

The problem of type checking union types, which is discussed in Section 6.12, leads to one major design issue. The other fundamental question is how to syntactically represent a union. In some designs, unions are confined to be parts of record structures, but in others they are not. So, the primary design issues that are particular to union types are the following:

- Should type checking be required? Note that any such type checking must be dynamic.
- Should unions be embedded in records?

6.10.2 Discriminated Versus Free Unions

C and C++ provide union constructs in which there is no language support for type checking. In C and C++, the **union** construct is used to specify union structures. The unions in these languages are called **free unions**, because programmers are allowed complete freedom from type checking in their use. For example, consider the following C union:

```
union flexType {
   int intEl;
   float floatEl;
};
union flexType el1;
float x;
...
el1.intEl = 27;
x = el1.floatEl;
```

This last assignment is not type checked, because the system cannot determine the current type of the current value of `el1`, so it assigns the bit string representation of `27` to the **`float`** variable `x`, which of course is nonsense.

Type checking of unions requires that each union construct include a type indicator. Such an indicator is called a **tag**, or **discriminant**, and a union with a discriminant is called a **discriminated union**. The first language to provide discriminated unions was ALGOL 68. They are now supported by Ada, ML, Haskell, and F#.

6.10.3 Ada Union Types

The Ada design for discriminated unions, which is based on that of its predecessor language, Pascal, allows the user to specify variables of a variant record type that will store only one of the possible type values in the variant. In this way, the user can tell the system when the type checking can be static. Such a restricted variable is called a **constrained variant variable**.

The tag of a constrained variant variable is treated like a named constant. Unconstrained variant records in Ada allow the values of their variants to change types during execution. However, the type of the variant can be changed only by assigning the entire record, including the discriminant. This disallows inconsistent records because if the newly assigned record is a constant data aggregate, the value of the tag and the type of the variant can be statically checked for consistency.[7] If the assigned value is a variable, its consistency was guaranteed when it was assigned, so the new value of the variable now being assigned is sure to be consistent.

The following example shows an Ada variant record:

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form : Shape) is
  record
    Filled : Boolean;
    Color : Colors;
    case Form is
      when Circle =>
        Diameter : Float;
      when Triangle =>
        Left_Side : Integer;
        Right_Side : Integer;
        Angle : Float;
      when Rectangle =>
        Side_1 : Integer;
        Side_2 : Integer;
    end case;
  end record;
```

The structure of this variant record is shown in Figure 6.8. The following two statements declare variables of type `Figure`:

```
Figure_1 : Figure;
Figure_2 : Figure(Form => Triangle);
```

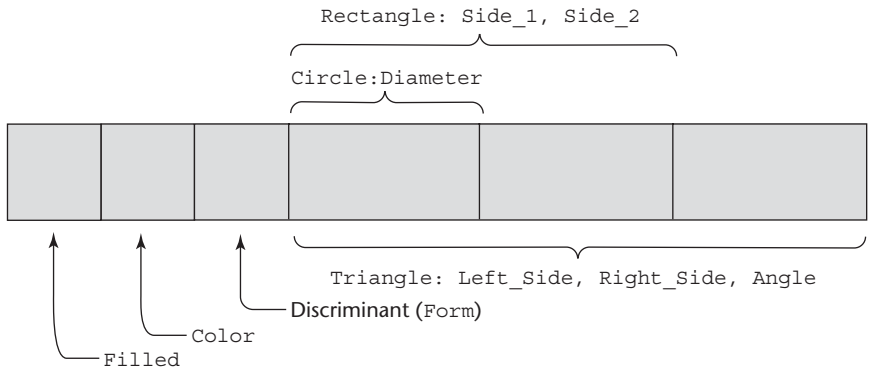
`Figure_1` is declared to be an unconstrained variant record that has no initial value. Its type can change by assignment of a whole record, including the discriminant, as in the following:

```
Figure_1 := (Filled => True,
             Color => Blue,
             Form => Rectangle,
             Side_1 => 12,
             Side_2 => 3);
```

7. Consistency here means that if the tag indicates the current type of the union is `Integer`, the current value of the union is in fact `Integer`.

Figure 6.8

A discriminated union of three shape variables (assume all variables are the same size)



The right side of this assignment is a data aggregate.

The variable `Figure_2` is declared constrained to be a triangle and cannot be changed to another variant.

This form of discriminated union is safe, because it always allows type checking, although the references to fields in unconstrained variants must be dynamically checked. For example, suppose we have the following statement:

```
if(Figure_1.Diameter > 3.0) ...
```

The run-time system would need to check `Figure_1` to determine whether its `Form` tag was `Circle`. If it was not, it would be a type error to reference its `Diameter`.

6.10.4 Unions in F#

A union is declared in F# with a type statement using OR operators (|) to define the components. For example, we could have the following:

```
type intReal =
   | IntValue of int
   | RealValue of float;;
```

In this example, `intReal` is the union type. `IntValue` and `RealValue` are constructors. Values of type `intReal` can be created using the constructors as if they were a function, as in the following examples:[8]

```
let ir1 = IntValue 17;;
let ir2 = RealValue 3.4;;
```

8. The **`let`** statement is used to assign values to names and to create a static scope; the double semicolons are used to terminate statements when the F# interactive interpreter is being used.

Accessing the value of a union is done with a pattern-matching structure. Pattern matching in F# is specified with the **match** reserved word. The general form of the construct is as follows:

match pattern **with**
  | expression_list$_1$ − > expression$_1$
  | . . .
  | expression_list$_n$ − > expression$_n$

The pattern can be any data type. The expression list can include wild card characters (_) or be solely a wild card character. For example, consider the following match construct:

```
let a = 7;;
let b = "grape";;
let x = match (a, b) with
    | 4, "apple" -> apple
    | _, "grape" -> grape
    | _ -> fruit;;
```

To display the type of the `intReal` union, the following function could be used:

```
let printType value =
    match value with
        | IntValue value -> printfn "It is an integer"
        | RealValue value -> printfn "It is a float";;
```

The following lines show calls to this function and the output:

```
printType ir1;;
It is an integer
printType ir2;;
It is a float
```

6.10.5 Evaluation

Unions are potentially unsafe constructs in some languages. They are one of the reasons why C and C++ are not strongly typed: These languages do not allow type checking of references to their unions. On the other hand, unions can be safely used, as in their design in Ada, ML, Haskell, and F#.

Neither Java nor C# includes unions, which may be reflective of the growing concern for safety in some programming languages.

6.10.6 Implementation of Union Types

Unions are implemented by simply using the same address for every possible variant. Sufficient storage for the largest variant is allocated. The tag of a discriminated union is stored with the variant in a recordlike structure.

At compile time, the complete description of each variant must be stored. This can be done by associating a case table with the tag entry in the descriptor. The case table has an entry for each variant, which points to a descriptor for that particular variant. To illustrate this arrangement, consider the following Ada example:

```
type Node (Tag : Boolean) is
  record
  case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
  end case;
 end record;
```

The descriptor for this type could have the form shown in Figure 6.9.

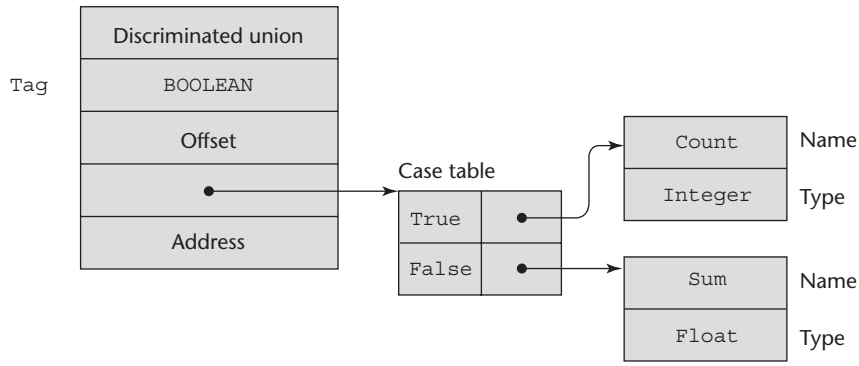


Figure 6.9

A compile-time descriptor for a discriminated union

6.11 Pointer and Reference Types

A **pointer** type is one in which the variables have a range of values that consists of memory addresses and a special value, **nil**. The value nil is not a valid address and is used to indicate that a pointer cannot currently be used to reference a memory cell.

Pointers are designed for two distinct kinds of uses. First, pointers provide some of the power of indirect addressing, which is frequently used in assembly language programming. Second, pointers provide a way to manage dynamic storage. A pointer can be used to access a location in an area where storage is dynamically allocated called a **heap**.

Variables that are dynamically allocated from the heap are called **heap-dynamic variables**. They often do not have identifiers associated with them and thus can be referenced only by pointer or reference type variables. Variables without names are called **anonymous variables**. It is in this latter application area of pointers that the most important design issues arise.

Pointers, unlike arrays and records, are not structured types, although they are defined using a type operator (`*` in C and C++ and `access` in Ada). Furthermore, they are also different from scalar variables because they are used to reference some other variable, rather than being used to store data. These two categories of variables are called **reference types** and **value types**, respectively.

Both kinds of uses of pointers add writability to a language. For example, suppose it is necessary to implement a dynamic structure like a binary tree in a language like Fortran 77, which does not have pointers. This would require the programmer to provide and maintain a pool of available tree nodes, which would probably be implemented in parallel arrays. Also, because of the lack of dynamic storage in Fortran 77, it would be necessary for the programmer to guess the maximum number of required nodes. This is clearly an awkward and error-prone way to deal with binary trees.

Reference variables, which are discussed in Section 6.11.6, are closely related to pointers.

6.11.1 Design Issues

The primary design issues particular to pointers are the following:

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable (the value a pointer references)?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

6.11.2 Pointer Operations

Languages that provide a pointer type usually include two fundamental pointer operations: assignment and dereferencing. The first operation sets a pointer variable's value to some useful address. If pointer variables are used only to manage dynamic storage, then the allocation mechanism, whether by operator or built-in subprogram, serves to initialize the pointer variable. If pointers are used for indirect addressing to variables that are not heap dynamic, then there must be an explicit operator or built-in subprogram for fetching the address of a variable, which can then be assigned to the pointer variable.

An occurrence of a pointer variable in an expression can be interpreted in two distinct ways. First, it could be interpreted as a reference to the contents

of the memory cell to which it is bound, which in the case of a pointer is an address. This is exactly how a nonpointer variable in an expression would be interpreted, although in that case its value likely would not be an address. However, the pointer could also be interpreted as a reference to the value in the memory cell pointed to by the memory cell to which the pointer variable is bound. In this case, the pointer is interpreted as an indirect reference. The former case is a normal pointer reference; the latter is the result of **dereferencing** the pointer. Dereferencing, which takes a reference through one level of indirection, is the second fundamental pointer operation.

Dereferencing of pointers can be either explicit or implicit. In Fortran 95+ it is implicit, but in many other contemporary languages, it occurs only when explicitly specified. In C++, it is explicitly specified with the asterisk (`*`) as a prefix unary operator. Consider the following example of dereferencing: If `ptr` is a pointer variable with the value 7080 and the cell whose address is 7080 has the value 206, then the assignment

```
j = *ptr
```

sets `j` to 206. This process is shown in Figure 6.10.

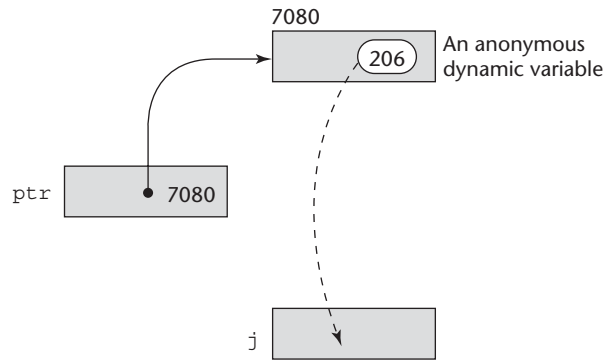


Figure 6.10

The assignment operation `j = *ptr`

When pointers point to records, the syntax of the references to the fields of these records varies among languages. In C and C++, there are two ways a pointer to a record can be used to reference a field in that record. If a pointer variable `p` points to a record with a field named `age`, `(*p).age` can be used to refer to that field. The operator `->`, when used between a pointer to a record and a field of that record, combines dereferencing and field reference. For example, the expression `p -> age` is equivalent to `(*p).age`. In Ada, `p.age` can be used, because such uses of pointers are implicitly dereferenced.

Languages that provide pointers for the management of a heap must include an explicit allocation operation. Allocation is sometimes specified with a subprogram, such as `malloc` in C. In languages that support object-oriented programming, allocation of heap objects is often specified with the **`new`** operator. C++, which does not provide implicit deallocation, uses **`delete`** as its deallocation operator.

6.11.3 Pointer Problems

The first high-level programming language to include pointer variables was PL/I, in which pointers could be used to refer to both heap-dynamic variables and other program variables. The pointers of PL/I were highly flexible, but their use could lead to several kinds of programming errors. Some of the problems of PL/I pointers are also present in the pointers of subsequent languages. Some recent languages, such as Java, have replaced pointers completely with reference types, which, along with implicit deallocation, minimize the primary problems with pointers. A reference type is really only a pointer with restricted operations.

6.11.3.1 Dangling Pointers

A **dangling pointer**, or **dangling reference**, is a pointer that contains the address of a heap-dynamic variable that has been deallocated. Dangling pointers are dangerous for several reasons. First, the location being pointed to may have been reallocated to some new heap-dynamic variable. If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid. Even if the new dynamic variable is the same type, its new value will have no relationship to the old pointer's dereferenced value. Furthermore, if the dangling pointer is used to change the heap-dynamic variable, the value of the new heap-dynamic variable will be destroyed. Finally, it is possible that the location now is being temporarily used by the storage management system, possibly as a pointer in a chain of available blocks of storage, thereby allowing a change to the location to cause the storage manager to fail.

The following sequence of operations creates a dangling pointer in many languages:

1. A new heap-dynamic variable is created and pointer `p1` is set to point at it.
2. Pointer `p2` is assigned `p1`'s value.
3. The heap-dynamic variable pointed to by `p1` is explicitly deallocated (possibly setting `p1` to `nil`), but `p2` is not changed by the operation. `p2` is now a dangling pointer. If the deallocation operation did not change `p1`, both `p1` and `p2` would be dangling. (Of course, this is a problem of aliasing—`p1` and `p2` are aliases.)

For example, in C++ we could have the following:

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Now, arrayPtr1 is dangling, because the heap storage
// to which it was pointing has been deallocated.
```

In C++, both arrayPtr1 and arrayPtr2 are now dangling pointers, because the C++ **`delete`** operator has no effect on the value of its operand pointer. In C++, it is common (and safe) to follow a **`delete`** operator with an assignment of zero, which represents null, to the pointer whose pointed-to value has been deallocated.

Notice that the explicit deallocation of dynamic variables is the cause of dangling pointers.

history note

Pascal included an explicit deallocate operator: dispose. Because of the problem of dangling pointers caused by dispose, some Pascal implementations simply ignored dispose when it appeared in a program. Although this effectively prevents dangling pointers, it also disallows the reuse of heap storage that the program no longer needs. Recall that Pascal initially was designed as a teaching language, rather than as an industrial tool.

6.11.3.2 Lost Heap-Dynamic Variables

A **lost heap-dynamic variable** is an allocated heap-dynamic variable that is no longer accessible to the user program. Such variables are often called **garbage**, because they are not useful for their original purpose, and they also cannot be reallocated for some new use in the program. Lost heap-dynamic variables are most often created by the following sequence of operations:

1. Pointer `p1` is set to point to a newly created heap-dynamic variable.
2. `p1` is later set to point to another newly created heap-dynamic variable.

The first heap-dynamic variable is now inaccessible, or lost. This is sometimes called **memory leakage**. Memory leakage is a problem, regardless of whether the language uses implicit or explicit deallocation. In the following sections, we investigate how language designers have dealt with the problems of dangling pointers and lost heap-dynamic variables.

6.11.4 Pointers in Ada

Ada's pointers are called **access** types. The dangling-pointer problem is partially alleviated by Ada's design, at least in theory. A heap-dynamic variable may be (at the implementor's option) implicitly deallocated at the end of the scope of its pointer type; thus, dramatically lessening the need for explicit deallocation. However, few if any Ada compilers implement this form of garbage collection, so the advantage is nearly always in theory only. Because heap-dynamic variables can be accessed by variables of only one type, when the end of the scope of that type declaration is reached, no pointers can be left pointing at the dynamic variable. This diminishes the problem, because improperly implemented explicit deallocation is the major source of dangling pointers. Unfortunately, the Ada language also has an explicit deallocator, `Unchecked_Deallocation`. Its name is meant to discourage its use, or at least warn the user of its potential problems. `Unchecked_Deallocation` can cause dangling pointers.

The lost heap-dynamic variable problem is not eliminated by Ada's design of pointers.