6

Data Types

This chapter first introduces the concept of a data type and the characteristics of the common primitive data types. Then, the designs of enumeration and subrange types are discussed. Next, the details of structured data types—specifically arrays, associative arrays, records, tuples, lists, and unions—are investigated. This section is followed by an in-depth look at pointers and references. The last category of types discussed are the optional types.

For each of the various categories of data types, the design issues are stated and the design choices made by the designers of some common languages are described. These designs are then evaluated.

The next three sections provide a thorough investigation of type checking, strong typing, and type equivalence rules. The last section of the chapter briefly introduces the fundamentals of the theory of data types.

Implementation methods for data types sometimes have a significant impact on their design. Therefore, implementation of the various data types is another important part of this chapter, especially arrays.

6.1 Introduction

A **data type** defines a collection of data values and a set of predefined operations on those values. Computer programs produce results by manipulating data. An important factor in determining the ease with which they can perform this task is how well the data types available in the language being used match the objects in the real world of the problem being addressed. Therefore, it is crucial that a language supports an appropriate collection of data types and structures.

The contemporary concepts of data typing have evolved over the last 60 years. In the earliest languages, all problem space data structures had to be modeled with only a few basic language-supported data structures. For example, in pre-90 Fortrans, linked lists and binary trees were implemented with arrays.

The data structures of COBOL took the first step away from the Fortran I model by allowing programmers to specify the accuracy of decimal data values, and also by providing a structured data type for records of information. PL/I extended the capability of accuracy specification to integer and floating-point types. The designers of PL/I included many data types, with the intent of supporting a large range of applications. A better approach, introduced in ALGOL 68, is to provide a few basic types and a few flexible structure-defining operators that allow a programmer to design a data structure for each need. Clearly, this was one of the most important advances in the evolution of data type design. User-defined types also provide improved readability through the use of meaningful names for types. They allow type checking of the variables of a special category of use, which would otherwise not be possible. User-defined types also aid modifiability: A programmer can change the type of a category of variables in a program by changing a type definition statement only.

Taking the concept of a user-defined type a step further, we arrive at abstract data types, which are supported by most programming languages designed since the mid-1980s. The fundamental idea of an abstract data type

is that the interface of a type, which is visible to the user, is separated from the representation and set of operations on values of that type, which are hidden from the user. All of the types provided by a high-level programming language are abstract data types. User-defined abstract data types are discussed in detail in Chapter 11.

There are a number of uses of the type system of a programming language. The most practical of these is error detection. The process and value of type checking, which is directed by the type system of the language, are discussed in Section 6.12. A second use of a type system is the assistance it provides for program modularization. This results from the cross-module type checking that ensures the consistency of the interfaces among modules. Another use of a type system is documentation. The type declarations in a program document information about its data, which provides clues about the program's behavior.

The type system of a programming language defines how a type is associated with each expression in the language and includes its rules for type equivalence and type compatibility. Certainly, one of the most important parts of understanding the semantics of a programming language is understanding its type system.

The two most common structured (nonscalar) data types in the imperative languages are arrays and records, although the popularity of associative arrays has increased significantly in recent years. Lists have been a central part of functional programming languages since the first such language appeared in 1959 (Lisp). Over the last decade, the increasing popularity of functional programming has led to lists being added to primarily imperative languages, such as Python and C#.

The structured data types are defined with type operators, or constructors, which are used to form type expressions. For example, C uses brackets and asterisks as type operators to specify arrays and pointers.

It is convenient, both logically and concretely, to think of variables in terms of descriptors. A **descriptor** is the collection of the attributes of a variable. In an implementation, a descriptor is an area of memory that stores the attributes of a variable. If the attributes are all static, descriptors are required only at compile time. These descriptors are built by the compiler, usually as a part of the symbol table, and are used during compilation. For dynamic attributes, however, part or all of the descriptor must be maintained during execution. In this case, the descriptor is used by the run-time system. In all cases, descriptors are used for type checking and building the code for the allocation and deallocation operations.

Care must be taken when using the term *variable*. One who uses only traditional imperative languages may think of identifiers as variables, but that can lead to confusion when considering data types. Identifiers do not have data types in some programming languages. It is wise to remember that identifiers are just one of the attributes of a variable.

The word *object* is often associated with the value of a variable and the space it occupies. In this book, however, we reserve *object* exclusively for instances of user-defined and language-defined abstract data types, rather than for the

values of all program variables of predefined types. Objects are discussed in detail in Chapters 11 and 12.

In the following sections, many common data types are discussed. For most, design issues particular to the type are stated. For all, one or more example designs are described. One design issue is fundamental to all data types: What operations are provided for variables of the type, and how are they specified?

6.2 Primitive Data Types

Data types that are not defined in terms of other types are called **primitive data types**. Nearly all programming languages provide a set of primitive data types. Some of the primitive types are merely reflections of the hardware—for example, most integer types. Others require only a little nonhardware support for their implementation.

To specify the structured types, the primitive data types of a language are used, along with one or more type constructors.

6.2.1 Numeric Types

Some early programming languages only had numeric primitive types. Numeric types still play a central role among the collections of types supported by contemporary languages.

6.2.1.1 Integer

The most common primitive numeric data type is **`integer`**. The hardware of many computers supports several sizes of integers. These sizes of integers, and often a few others, are supported by some programming languages. For example, Java includes four signed integer sizes: **`byte`**, **`short`**, **`int`**, and **`long`**. Some languages, for example, C++ and C#, include unsigned integer types, which are types for integer values without signs. Unsigned types are often used for binary data.

A signed integer value is represented in a computer by a string of bits, with one of the bits (typically the leftmost) representing the sign. Most integer types are supported directly by the hardware. One example of an integer type that is not supported directly by the hardware is the long integer type of Python (F# also provides such integers). Values of this type can have unlimited length. Long integer values can be specified as literals, as in the following example:

```
243725839182756281923L
```

Integer arithmetic operations in Python that produce values too large to be represented with **`int`** type store them as long integer type values.

A negative integer could be stored in sign-magnitude notation, in which the sign bit is set to indicate negative and the remainder of the bit string represents the absolute value of the number. Sign-magnitude notation, however, does not lend itself to computer arithmetic. Most computers now use a notation called **twos complement** to store negative integers, which is convenient for addition and subtraction. In twos-complement notation, the representation of a negative integer is formed by taking the logical complement of the positive version of the number and adding one. Ones-complement notation is still used by some computers. In ones-complement notation, the negative of an integer is stored as the logical complement of its absolute value. Ones-complement notation has the disadvantage that it has two representations of zero. See any book on assembly language programming for details of integer representations.

6.2.1.2 Floating-Point

Floating-point data types model real numbers, but the representations are only approximations for many real values. For example, neither of the fundamental numbers π or e (the base for the natural logarithms) can be correctly represented in floating-point notation. Of course, neither of these numbers can be precisely represented in any finite amount of computer memory. On most computers, floating-point numbers are stored in binary, which exacerbates the problem. For example, even the value 0.1 in decimal cannot be represented by a finite number of binary digits.[1] Another problem with floating-point types is the loss of accuracy through arithmetic operations. For more information on the problems of floating-point notation, see any book on numerical analysis.

Floating-point values are represented as fractions and exponents, a form that is borrowed from scientific notation. Older computers used a variety of different representations for floating-point values. However, most newer machines use the IEEE Floating-Point Standard 754 format. Language implementors use whatever representation is supported by the hardware. Most languages include two floating-point types, often called **float** and **double**. The float type is the standard size, usually stored in four bytes of memory. The double type is provided for situations where larger fractional parts and/or a larger range of exponents is needed. Double-precision variables usually occupy twice as much storage as float variables and provide at least twice the number of bits of fraction.

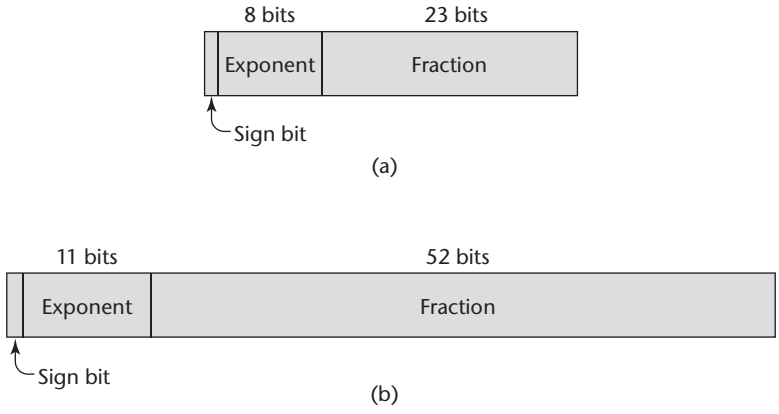
The collection of values that can be represented by a floating-point type is defined in terms of precision and range. **Precision** is the accuracy of the fractional part of a value, measured as the number of bits. **Range** is a combination of the range of fractions and, more important, the range of exponents.

Figure 6.1 shows the IEEE Floating-Point Standard 754 format for single- and double-precision representation (IEEE, 1985). Details of the IEEE formats can be found in Tanenbaum (2005).

1. 0.1 in decimal is 0.0001100110011 . . . in binary.

Figure 6.1

IEEE floating-point formats: (a) single precision, (b) double precision



6.2.1.3 Complex

Some programming languages support a complex data type—for example, Fortran and Python. Complex values are represented as ordered pairs of floating-point values. In Python, the imaginary part of a complex literal is specified by following it with a `j` or `J`—for example,

```
(7 + 3j)
```

Languages that support a complex type include operations for arithmetic on complex values.

6.2.1.4 Decimal

Most larger computers that are designed to support business systems applications have hardware support for **decimal** data types. Decimal data types store a fixed number of decimal digits, with the implied decimal point at a fixed position in the value. These are the primary data types for business data processing and are therefore essential to COBOL. C# and F# also have decimal data types.

Decimal types have the advantage of being able to precisely store decimal values, at least those within a restricted range, which cannot be done with floating-point. For example, the number 0.1 (in decimal) can be exactly represented in a decimal type, but not in a floating-point type, as is noted in Section 6.2.1.2. The disadvantages of decimal types are that the range of values is restricted because no exponents are allowed, and their representation in memory is mildly wasteful, for reasons discussed in the following paragraph.

Decimal types are stored very much like character strings, using binary codes for the decimal digits. These representations are called **binary coded decimal (BCD)**. In some cases, they are stored one digit per byte, but in others, they are packed two digits per byte. Either way, they take more storage than binary representations. It takes at least four bits to code a decimal digit. Therefore, to store a six-digit coded decimal number requires 24 bits of memory.

However, it takes only 20 bits to store the same number in binary.[2] The operations on decimal values are done in hardware on machines that have such capabilities; otherwise, they are simulated in software.

6.2.2 Boolean Types

Boolean types are perhaps the simplest of all types. Their range of values has only two elements: one for true and one for false. They were introduced in ALGOL 60 and have been included in most general-purpose languages designed since 1960. One popular exception is C89, in which numeric expressions are used as conditionals. In such expressions, all operands with nonzero values are considered true, and zero is considered false. Although C99 and C++ have a Boolean type, they also allow numeric expressions to be used as if they were Boolean. This is not the case in the subsequent languages, Java and C#.

Boolean types are often used to represent switches or flags in programs. Although other types, such as integers, can be used for these purposes, the use of Boolean types is more readable.

A Boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte.

6.2.3 Character Types

Character data are stored in computers as numeric codings. Traditionally, the most commonly used coding was the 8-bit code ASCII (American Standard Code for Information Interchange), which uses the values 0 to 127 to code 128 different characters. ISO 8859-1 is another 8-bit character code, but it allows 256 different characters.

Because of the globalization of business and the need for computers to communicate with other computers around the world, the ASCII character set became inadequate. In response, in 1991, the Unicode Consortium published the UCS-2 standard, a 16-bit character set. This character code is often called Unicode. Unicode includes the characters from most of the world's natural languages. For example, Unicode includes the Cyrillic alphabet, as used in Serbia, and the Thai digits. The first 128 characters of Unicode are identical to those of ASCII. Java was the first widely used language to use the Unicode character set. Since then, it has found its way into JavaScript, Python, Perl, C#, F#, and Swift.

After 1991, the Unicode Consortium, in cooperation with the International Standards Organization (ISO), developed a 4-byte character code named UCS-4, or UTF-32, which is described in the ISO/IEC 10646 Standard, published in 2000.

---

2. Of course, unless a program needs to maintain a large number of large decimal values, the difference is insignificant.

To provide the means of processing codings of single characters, most programming languages include a primitive type for them. However, Python supports single characters only as character strings of length 1.

6.3 Character String Types

A **character string type** is one in which the values consist of sequences of characters. Character string constants are used to label output, and the input and output of all kinds of data are often done in terms of strings. Of course, character strings also are an essential type for all programs that do character manipulation.

6.3.1 Design Issues

The two most important design issues that are specific to character string types are the following:

- Should strings be a special kind of character array or a primitive type?
- Should strings have static or dynamic length?

6.3.2 Strings and Their Operations

The most common string operations are assignment, catenation, substring reference, comparison, and pattern matching.

A **substring reference** is a reference to a substring of a given string. Substring references are discussed in the more general context of arrays, where the substring references are called **slices**.

In general, both assignment and comparison operations on character strings are complicated by the possibility of string operands of different lengths. For example, what happens when a longer string is assigned to a shorter string, or vice versa? Usually, simple and sensible choices are made for these situations, although programmers often have trouble remembering them.

In some languages, pattern matching is supported directly in the language. In others, it is provided by a function or class library.

If strings are not defined as a primitive type, string data is usually stored in arrays of single characters and referenced as such in the language. This is the approach taken by C and C++, which use `char` arrays to store character strings. These languages provide a collection of string operations through standard libraries. Many users of strings and many of the library functions use the convention that character strings are terminated with a special character, null, which is represented with zero. This is an alternative to maintaining the length of string variables. The library operations simply carry out their operations until the null character appears in the string being operated on. Library functions that produce strings often supply the null character. The character string

literals that are built by the compiler also have the null character. For example, consider the following declaration:

```
char str[] = "apples";
```

In this example, `str` represents an array of **`char`** elements, specifically `apples0`, where `0` is the null character.

Some of the most commonly used library functions for character strings in C and C++ are `strcpy`, which moves strings; `strcat`, which catenates one given string onto another; `strcmp`, which lexicographically compares (by the order of their character codes) two given strings; and `strlen`, which returns the number of characters, not counting the null character, in the given string. The parameters and return values for most of the string manipulation functions are **`char`** pointers that point to arrays of **`char`**. Parameters can also be string literals.

The string manipulation functions of the C standard library, which are also available in C++, are inherently unsafe and have led to numerous programming errors. The problem is that the functions in this library that move string data do not guard against overflowing the destination. For example, consider the following call to `strcpy`:

```
strcpy(dest, src);
```

If the length of `dest` is 20 and the length of `src` is 50, `strcpy` will write over the 30 bytes that follow `dest`. The point is that `strcpy` does not know the length of `dest`, so it cannot ensure that the memory following it will not be overwritten. The same problem can occur with several of the other functions in the C string library. In addition to C-style strings, C++ also supports strings through its standard class library, which is also similar to that of Java. Because of the insecurities of the C string library, C++ programmers should use the `string` class from the standard library, rather than **`char`** arrays and the C string library.

In Java, strings are supported by the `String` class, whose values are constant strings, and the `StringBuffer` class, whose values are changeable and are more like arrays of single characters. These values are specified with methods of the `StringBuffer` class. C# and Ruby include string classes that are similar to those of Java.

Python includes strings as a primitive type and has operations for substring reference, catenation, indexing to access individual characters, as well as methods for searching and replacement. There is also an operation for character membership in a string. So, even though Python's strings are primitive types, for character and substring references, they act very much like arrays of characters. However, Python strings are immutable, similar to the `String` class objects of Java.

history note

SNOBOL 4 was the first widely known language to support pattern matching.

In F#, strings are a class. Individual characters, which are represented in Unicode UTF-16, can be accessed, but not changed. Strings can be catenated with the + operator. In ML, string is a

primitive immutable type. It uses ^ for its catenation operator and includes functions for substring referencing and getting the size of a string.

In Swift, the `String` class supports its character strings. `String` objects can be either constants or variables. The binary + operator catenates `String` variables. The `append` method is used to add a `Character` object to a `String` object. The `characters` method of `String` is used to examine individual characters of a `String` object.

Perl, JavaScript, Ruby, and PHP include built-in pattern-matching operations. In these languages, the pattern-matching expressions are somewhat loosely based on mathematical regular expressions. In fact, they are often called **regular expressions**. They evolved from the early UNIX line editor, `ed`, to become part of the UNIX shell languages. Eventually, they grew to their current complex form. There is at least one complete book on this kind of pattern-matching expressions (Friedl, 2006). In this section, we provide a brief look at the style of these expressions through two relatively simple examples.

Consider the following pattern expression:

```
/[A-Za-z][A-Za-z\d]+/
```

This pattern matches (or describes) the typical name form in programming languages. The brackets enclose character classes. The first character class specifies all letters; the second specifies all letters and digits (a digit is specified with the abbreviation `\d`). If only the second character class were included, we could not prevent a name from beginning with a digit. The plus operator following the second category specifies that there must be one or more of what is in the category. So, the whole pattern matches strings that begin with a letter, followed by one or more letters or digits.

Next, consider the following pattern expression:

```
/\d+\.?\d*|\.\d+/
```

This pattern matches numeric literals. The `\.` specifies a literal decimal point.[3] The question mark quantifies what it follows to have zero or one appearance. The vertical bar (`|`) separates two alternatives in the whole pattern. The first alternative matches strings of one or more digits, possibly followed by a decimal point, followed by zero or more digits; the second alternative matches strings that begin with a decimal point, followed by one or more digits.

Pattern-matching capabilities using regular expressions are included in the class libraries of C++, Java, Python, C#, and F#.

6.3.3 String Length Options

There are several design choices regarding the length of string values. First, the length can be static and set when the string is created. Such a string is called a **static length string**. This is the choice for the strings of Python, the

3. The period must be "escaped" with the backslash because period has special meaning in a regular expression.

immutable objects of Java's `String` class, as well as similar classes in the C++ standard class library, Ruby's built-in `String` class, and the .NET class library available to C# and F#.

The second option is to allow strings to have varying length up to a declared and fixed maximum set by the variable's definition, as exemplified by the strings in C and the C-style strings of C++. These are called **limited dynamic length strings**. Such string variables can store any number of characters between zero and the maximum. Recall that strings in C use a special character to indicate the end of the string's characters, rather than maintaining the string length.

The third option is to allow strings to have varying length with no maximum, as in JavaScript, Perl, and the standard C++ library. These are called **dynamic length strings**. This option requires the overhead of dynamic storage allocation and deallocation but provides maximum flexibility.

6.3.4 Evaluation

String types are important to the writability of a language. Dealing with strings as arrays can be more cumbersome than dealing with a primitive string type. For example, consider a language that treats strings as arrays of characters and does not have a predefined function that does what `strcpy` in C does. Then, a simple assignment of one string to another would require a loop. The addition of strings as a primitive type to a language is not costly in terms of either language or compiler complexity. Therefore, it is difficult to justify the omission of primitive string types in some contemporary languages. Of course, providing strings through a standard library is nearly as convenient as having them as a primitive type.

String operations such as simple pattern matching and catenation are essential and should be included for string type values. Although dynamic length strings are obviously the most flexible, the overhead of their implementation must be weighed against that additional flexibility.

6.3.5 Implementation of Character String Types

Character string types could be supported directly in hardware; but in most cases, software is used to implement string storage, retrieval, and manipulation. When character string types are represented as character arrays, the language often supplies few operations.

A descriptor for a static character string type, which is required only during compilation, has three fields. The first field of every descriptor is the name of the type. In the case of static character strings, the second field is the type's length (in characters). The third field is the address of the first character. This descriptor is shown in Figure 6.2. Limited dynamic strings require a run-time descriptor to store the fixed maximum length, the current length, and the address, as shown in Figure 6.3. Dynamic length strings require a simpler run-time descriptor because only the current length and the address need to be stored. Although we depict descriptors as independent blocks of storage, in most cases, they are stored in the symbol table.

Figure 6.2

Compile-time descriptor for static strings

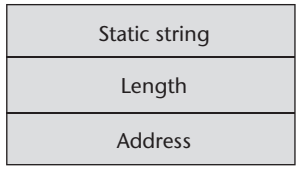
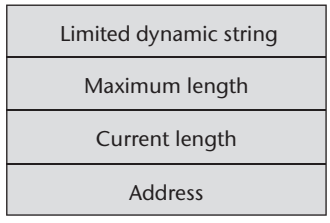


Figure 6.3

Run-time descriptor for limited dynamic strings



The limited dynamic strings of C and C++ do not require run-time descriptors, because the end of a string is marked with the null character. They do not need the maximum length, because index values in array references are not range checked in these languages.

Static length and limited dynamic length strings require no special dynamic storage allocation. In the case of limited dynamic length strings, sufficient storage for the maximum length is allocated when the string variable is bound to storage, so only a single allocation process is involved.

Dynamic length strings require more complex storage management. The length of a string, and therefore the storage to which it is bound, must grow and shrink dynamically.

There are three approaches to supporting the dynamic allocation and deallocation that is required for dynamic length strings. First, strings can be stored in a linked list, so that when a string grows, the newly required cells can come from anywhere in the heap. The drawbacks to this method are the extra storage occupied by the links in the list representation and the necessary complexity of string operations.

The second approach is to store strings as arrays of pointers to individual characters allocated in the heap. This method still uses extra memory, but string processing can be faster than with the linked-list approach.

The third alternative is to store complete strings in adjacent storage cells. The problem with this method arises when a string grows: How can storage that is adjacent to the existing cells continue to be allocated for the string variable? Frequently, such storage is not available. Instead, a new area of memory is found that can store the complete new string, and the old part is moved to this area. Then, the memory cells used for the old string are deallocated. This latter approach is the one typically used. The general problem of managing allocation and deallocation of variable-size segments is discussed in Section 6.11.7.3.

Although the linked-list method requires more storage, the associated allocation and deallocation processes are simple. However, some string operations

are slowed by the required pointer chasing. On the other hand, using adjacent memory for complete strings results in faster string operations and requires significantly less storage, but the allocation and deallocation processes are slower.

6.4 Enumeration Types

An **enumeration type** is one in which all of the possible values, which are named constants, are provided, or enumerated, in the definition. Enumeration types provide a way of defining and grouping collections of named constants, which are called **enumeration constants**. The definition of a typical enumeration type is shown in the following C# example:

```
enum  days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

The enumeration constants are typically implicitly assigned the integer values, 0, 1, . . . but can be explicitly assigned any integer literal in the type's definition.

6.4.1 Design Issues

The design issues for enumeration types are as follows:

- Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?
- Are enumeration values coerced to integer?
- Are any other types coerced to an enumeration type?

All of these design issues are related to type checking. If an enumeration variable is coerced to a numeric type, then there is little control over its range of legal operations or its range of values. If an `int` type value is coerced to an enumeration type, then an enumeration type variable could be assigned any integer value, whether it represented an enumeration constant or not.

6.4.2 Designs

In languages that do not have enumeration types, programmers usually simulate them with integer values. For example, suppose we needed to represent colors in a C program and C did not have an enumeration type. We might use 0 to represent blue, 1 to represent red, and so forth. These values could be defined as follows:

```
int    red = 0, blue = 1;
```

Now, in the program, we could use `red` and `blue` as if they were of a color type. The problem with this approach is that because we have not defined a type for

our colors, there is no type checking when they are used. For example, it would be legal to add the two together, although that would rarely be an intended operation. They could also be combined with any other numeric type operand using any arithmetic operator, which would also rarely be useful. Furthermore, because they are just variables, they could be assigned any integer value, thereby destroying the relationship with the colors. This latter problem could be prevented by making them named constants.

C and Pascal were the first widely used languages to include an enumeration data type. C++ includes C's enumeration types. In C++, we could have the following:

```
enum  colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
```

The `colors` type uses the default internal values for the enumeration constants, 0, 1, . . . , although the constants could have been specifically assigned any integer literal (or any constant-valued expression) by the programmer. The enumeration values are coerced to **`int`** when they are put in integer context. This allows their use in any numeric expression. For example, if the current value of `myColor` is `blue`, then the expression

```
myColor++
```

would assign the integer code for `green` to `myColor`.

C++ also allows enumeration constants to be assigned to variables of any numeric type, though that would likely be an error. However, no other type value is coerced to an enumeration type in C++. For example,

```
myColor = 4;
```

is illegal in C++. This assignment would be legal if the right side had been cast to `colors` type. This prevents some potential errors.

C++ enumeration constants can appear in only one enumeration type in the same referencing environment.

In 2004, an enumeration type was added to Java in Java 5.0. All enumeration types in Java are implicitly subclasses of the predefined class `Enum`. Because enumeration types are classes, they can have instance data fields, constructors, and methods. Syntactically, Java enumeration type definitions appear like those of C++, except that they can include fields, constructors, and methods. The possible values of an enumeration are the only possible instances of the class. All enumeration types inherit `toString`, as well as a few other methods. An array of the instances of an enumeration type can be fetched with the static method `values`. The internal numeric value of an enumeration variable can be fetched with the `ordinal` method. No expression of any other type can be assigned to an enumeration variable. Also, an enumeration variable is never coerced to any other type.

C# enumeration types are like those of C++, except that they are never coerced to integer. So, operations on enumeration types are restricted to those that make sense. Also, the range of values is restricted to that of the particular enumeration type.

In ML, enumeration types are defined as new types with **datatype** declarations. For example, we could have the following:

```
datatype weekdays = Monday | Tuesday | Wednesday |
Thursday | Friday
```

The type of the elements of `weekdays` is integer.

F# has enumeration types that are similar to those of ML, except the reserved word **type** is used instead of **datatype** and the first value is preceded by an OR operator (`|`).

Swift has an enumeration type in which the enumeration values are names, which represent themselves, rather than having internal integer values. An enumeration type is defined in a structure that is similar to a switch structure, as in:

```
enum fruit {
   case orange
   case apple
   case banana
}
```

Dot notation is used to reference enumeration values, so in our example, the value of `apple` is referenced as `fruit.apple`.

Interestingly, none of the relatively recent scripting languages include enumeration types. These include Perl, JavaScript, PHP, Python, and Ruby. Even Java was a decade old before enumeration types were added.

6.4.3 Evaluation

Enumeration types can provide advantages in both readability and reliability. Readability is enhanced very directly: Named values are easily recognized, whereas coded values are not.

In the area of reliability, the enumeration types of C#, F#, Java 5.0, and Swift provide two advantages: (1) No arithmetic operations are legal on enumeration types; this prevents adding days of the week, for example, and (2) second, no enumeration variable can be assigned a value outside its defined range.[4] If the `colors` enumeration type has 10 enumeration constants and uses `0..9` as its internal values, no number greater than `9` can be assigned to a `colors` type variable.

Because C treats enumeration variables like integer variables, it does not provide either of these two advantages.

C++ is a little better. Numeric values can be assigned to enumeration type variables only if they are cast to the type of the assigned variable. Numeric values assigned to enumeration type variables are checked to determine whether they are in the range of the internal values of the enumeration type.

4. In C# and F#, an integer value can be cast to an enumeration type and assigned to the name of an enumeration variable. Such values must be tested with `Enum.IsDefined` method before assigning them to the name of an enumeration variable.

Unfortunately, if the user uses a wide range of explicitly assigned values, this checking is not effective. For example,

```
enum  colors {red = 1, blue = 1000, green = 100000}
```

In this example, a value assigned to a variable of `colors` type will only be checked to determine whether it is in the range of `1..100000`.

6.5 Array Types

An **array** is a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element. The individual data elements of an array are of the same type. References to individual array elements are specified using subscript expressions. If any of the subscript expressions in a reference include variables, then the reference will require an additional run-time calculation to determine the address of the memory location being referenced.

In many languages, such as C, C++, Java, and C#, all of the elements of an array are required to be of the same type. In these languages, pointers and references are restricted to point to or reference a single type. So the objects or data values being pointed to or referenced are also of a single type. In some other languages, such as JavaScript, Python, and Ruby, variables are typeless references to objects or data values. In these cases, arrays still consist of elements of a single type, but the elements can reference objects or data values of different types. Such arrays are still homogeneous, because the array elements are of the same type. In Swift, arrays can be typed, that is, they will contain values only of a single type, or untyped, which means they can contain values of any type.

C# and Java 5.0 provide generic arrays, that is, arrays whose elements are references to objects, through their class libraries. These are discussed in Section 6.5.3.

6.5.1 Design Issues

The primary design issues specific to arrays are the following:

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does array allocation take place?
- Are ragged or rectangular multidimensioned arrays allowed, or both?
- Can arrays be initialized when they have their storage allocated?
- What kinds of slices are allowed, if any?

In the following sections, examples of the design choices made for the arrays of the most common programming languages are discussed.

6.5.2 Arrays and Indices

Specific elements of an array are referenced by means of a two-level syntactic mechanism, where the first part is the aggregate name, and the second part is a possibly dynamic selector consisting of one or more items known as **subscripts** or **indices**. If all of the subscripts in a reference are constants, the selector is static; otherwise, it is dynamic. The selection operation can be thought of as a mapping from the array name and the set of subscript values to an element in the aggregate. Indeed, arrays are sometimes called **finite mappings**. Symbolically, this mapping can be shown as

array_name(subscript_value_list) → element

The syntax of array references is fairly universal: The array name is followed by the list of subscripts, which is surrounded by either parentheses or brackets. In some languages that provide multidimensioned arrays as arrays of arrays, each subscript appears in its own brackets. A problem with using parentheses to enclose subscript expressions is that they often are also used to enclose the parameters in subprogram calls; this use makes references to arrays appear exactly like those calls. For example, consider the following Ada assignment statement:

```
Sum := Sum + B(I);
```

Because parentheses are used for both subprogram parameters and array subscripts in Ada, both program readers and compilers are forced to use other information to determine whether `B(I)` in this assignment is a function call or a reference to an array element. This results in reduced readability.

The designers of Ada specifically chose parentheses to enclose subscripts so there would be uniformity between array references and function calls in expressions, in spite of potential readability problems. They made this choice in part because both array element references and function calls are mappings. Array element references map the subscripts to a particular element of the array. Function calls map the actual parameters to the function definition and, eventually, a functional value.

Most languages other than Fortran and Ada use brackets to delimit their array indices.

Two distinct types are involved in an array type: the element type and the type of the subscripts. The type of the subscripts is often integer.

Early programming languages did not specify that subscript ranges must be implicitly checked. Range errors in subscripts are common in programs, so requiring range checking is an important factor in the reliability of languages. Many contemporary languages also do not specify range checking of subscripts, but Java, ML, and C# do.

Subscripting in Perl is a bit unusual in that although the names of all arrays begin with at signs (`@`), because array elements are always scalars and the names

history note

The designers of pre-90 Fortrans and PL/I chose parentheses for array subscripts because no other suitable characters were available at the time. Card punches did not include bracket characters.

history note

Fortran I limited the number of array subscripts to three, because at the time of the design, execution efficiency was a primary concern. Fortran I designers had developed a very fast method for accessing the elements of arrays of up to three dimensions, using the three index registers of the IBM 704. Fortran IV was first implemented on an IBM 7094, which had seven index registers. This allowed Fortran IV's designers to allow arrays with up to seven subscripts. Most other contemporary languages enforce no such limits.

of scalars always begin with dollar signs ($), references to array elements use dollar signs rather than at signs in their names. For example, for the array @list, the second element is referenced with $list[1].

One can reference an array element in Perl with a negative subscript, in which case the subscript value is an offset from the end of the array. For example, if the array @list has five elements with the subscripts 0..4, $list[−2] references the element with the subscript 3. A reference to a nonexistent element in Perl yields **undef**, but no error is reported.

6.5.3 Subscript Bindings and Array Categories

The binding of the subscript type to an array variable is usually static, but the subscript value ranges are sometimes dynamically bound.

In some languages, the lower bound of the subscript range is implicit. For example, in the C-based languages, the lower bound of all subscript ranges is fixed at 0. In some other languages, the lower bounds of the subscript ranges must be specified by the programmer.

There are four categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated. The category names indicate the design choices of these three. In the first three of these categories, once the subscript ranges are bound and the storage is allocated, they remain fixed for the lifetime of the variable. Of course, when the subscript ranges are fixed, the array cannot change size.

A **static array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time). The advantage of static arrays is efficiency: No dynamic allocation or deallocation is required. The disadvantage is that the storage for the array is fixed for the entire execution time of the program.

A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution. The advantage of fixed stack-dynamic arrays over static arrays is space efficiency. A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time. The same is true if the two arrays are in different blocks that are not active at the same time. The disadvantage is the required allocation and deallocation time.

A **fixed heap-dynamic array** is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated. The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution, and the storage is allocated from the heap, rather than the stack. The advantage of fixed heap-dynamic arrays is flexibility—the array's size always fits the problem. The disadvantage is allocation time from the heap, which is longer than allocation time from the stack.

A **heap-dynamic array** is one in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime. The advantage of heap-dynamic arrays over the others is flexibility: Arrays can grow and shrink during program execution as the need for space changes. The disadvantage is that allocation and deallocation take

longer and may happen many times during execution of the program. Examples of the four categories are given in the following paragraphs.

Arrays declared in C and C++ functions that include the **static** modifier are static.

Arrays that are declared in C and C++ functions without the **static** specifier are examples of fixed stack-dynamic arrays.

C and C++ also provide fixed heap-dynamic arrays. The standard C library functions `malloc` and `free`, which are general heap allocation and deallocation operations, respectively, can be used for C arrays. C++ uses the operators **new** and **delete** to manage heap storage. An array is treated as a pointer to a collection of storage cells, where the pointer can be indexed, as discussed in Section 6.11.5.

In Java, all non-generic arrays are fixed heap-dynamic. Once created, these arrays keep the same subscript ranges and storage. C# also provides fixed heap-dynamic arrays.

Objects of the C# `List` class are generic heap-dynamic arrays. These array objects are created without any elements, as in

```
List<String> stringList =  new  List<String>();
```

Elements are added to this object with the `Add` method, as in

```
stringList.Add("Michael");
```

Access to elements of these arrays is through subscripting.

Java includes a generic class similar to C#'s `List`, named `ArrayList`. It is different from C#'s `List` in that subscripting is not supported—`get` and `set` methods must be used to access the elements.

A Perl array can be made to grow by using the `push` (puts one or more new elements on the end of the array) and `unshift` (puts one or more new elements on the beginning of the array), or by assigning a value to the array specifying a subscript beyond the highest current subscript of the array. An array can be made to shrink to no elements by assigning it the empty list, `()`. The length of an array is defined to be the largest subscript plus one.

Like Perl, JavaScript allows arrays to grow with the `push` and `unshift` methods and shrink by setting them to the empty list. However, negative subscripts are not supported.

JavaScript arrays can be sparse, meaning the subscript values need not be contiguous. For example, suppose we have an array named list that has 10 elements with the subscripts 0..9.[5] Consider the following assignment statement:

```
list[50] = 42;
```

Now, `list` has 11 elements and length 51. The elements with subscripts `11..49` are not defined and therefore do not require storage. A reference to a nonexistent element in a JavaScript array yields **undefined**.

5. The subscript range could just as easily have been 1000 .. 1009.

Arrays in Python and Ruby can be made to grow only through methods to add elements or catenate other arrays. Ruby and Perl support negative subscripts, but Python does not. In Python an element or slice of an array can be deleted. A reference to a nonexistent element in Python results in a run-time error, whereas a similar reference in Ruby yields **nil** and no error is reported.

Swift dynamic arrays are objects that use integer subscripts, beginning at zero, and include several useful methods. The `append` method adds an element to the end of an array. The `insert` method inserts a new element at any position in the array, but results in an error if the insertion is at a subscript beyond the current length of the array. Elements can be removed from an array with the `removeAtIndex` method. There are also `reverse` and `count` methods.

Although the ML definition does not include arrays, its widely used implementation, SML/NJ, does.

The only predefined collection type that is part of F# is the array (other collection types are provided through the .NET Framework Library). These arrays are like those of C#. A **foreach** statement is included in the language for array processing.

6.5.4 Array Initialization

Some languages provide the means to initialize arrays at the time their storage is allocated. C, C++, Java, Swift, and C# allow initialization of their arrays. Consider the following C declaration:

```
int  list [] = {4, 5, 7, 83};
```

The array `list` is created and initialized with the values `4`, `5`, `7`, and `83`. The compiler also sets the length of the array. This is meant to be a convenience but is not without cost. It effectively removes the possibility that the system could detect some kinds of programmer errors, such as mistakenly leaving a value out of the list.

As discussed in Section 6.3.2, character strings in C and C++ are implemented as arrays of **char**. These arrays can be initialized to string constants, as in

```
char  name [] = "freddie";
```

The array `name` will have eight elements, because all strings are terminated with a null character (zero), which is implicitly supplied by the system for string constants.

Arrays of strings in C and C++ can also be initialized with string literals. For example,

```
char  *names [] = {"Bob", "Jake", "Darcie"};
```

This example illustrates the nature of character literals in C and C++. In the previous example of a string literal being used to initialize the **char** array `name`, the literal is taken to be a **char** array. But in the latter example (`names`), the

literals are taken to be pointers to characters, so the array is an array of pointers to characters. For example, `names[0]` is a pointer to the letter `'B'` in the literal character array that contains the characters `'B'`, `'o'`, `'b'`, and the null character.

In Java, similar syntax is used to define and initialize an array of references to `String` objects. For example,

```
String[] names = ["Bob", "Jake", "Darcie"];
```

6.5.5 Array Operations

An array operation is one that operates on an array as a unit. The most common array operations are assignment, catenation, comparison for equality and inequality, and slices, which are discussed separately in Section 6.5.5.

The C-based languages do not provide any array operations, except through the methods of Java, C++, and C#. Perl supports array assignments but does not support comparisons.

Python's arrays are called lists, although they have all the characteristics of dynamic arrays. Because the objects can be of any types, these arrays are heterogeneous. Python provides array assignment, although it is only a reference change. Python also has operations for array catenation (+) and element membership (**in**). It includes two different comparison operators: one that determines whether the two variables reference the same object (**is**) and one that compares all corresponding objects in the referenced objects, regardless of how deeply they are nested, for equality (==).

Like Python, the elements of Ruby's arrays are references to objects. And like Python, when a == operator is used between two arrays, the result is true only if the two arrays have the same length and the corresponding elements are equal. Ruby's arrays can be catenated with an `Array` method.

F# includes many array operators in its `Array` module. Among these are `Array.append`, `Array.copy`, and `Array.length`.

Arrays and their operations are the heart of APL; it is the most powerful array-processing language ever devised. Because of its relative obscurity and its lack of effect on subsequent languages, however, we present here only a glimpse into its array operations.

In APL, the four basic arithmetic operations are defined for vectors (single-dimensioned arrays) and matrices, as well as scalar operands. For example,

```
A + B
```

is a valid expression, whether `A` and `B` are scalar variables, vectors, or matrices.

APL includes a collection of unary operators for vectors and matrices, some of which are as follows (where `V` is a vector and `M` is a matrix):

ϕ`V`	reverses the elements of `V`
ϕ`M`	reverses the columns of `M`
θ`M`	reverses the rows of `M`
⍉`M`	transposes `M` (its rows become its columns and vice versa)
⌹`M`	`inverts M`

APL also includes several special operators that take other operators as operands. One of these is the inner product operator, which is specified with a period (.). It takes two operands, which are binary operators. For example,

```
+.×
```

is a new operator that takes two arguments, either vectors or matrices. It first multiplies the corresponding elements of two arguments, and then it sums the results. For example, if A and B are vectors,

```
A × B
```

is the mathematical inner product of A and B (a vector of the products of the corresponding elements of A and B). The statement

```
A +.× B
```

is the sum of the inner product of A and B. If A and B are matrices, this expression specifies the matrix multiplication of A and B.

The special operators of APL are actually functional forms, which are described in Chapter 15.

6.5.6 Rectangular and Jagged Arrays

A **rectangular array** is a multidimensioned array in which all of the rows have the same number of elements and all of the columns have the same number of elements. Rectangular arrays model rectangular tables exactly.

A **jagged array** is one in which the lengths of the rows need not be the same. For example, a jagged matrix may consist of three rows, one with 5 elements, one with 7 elements, and one with 12 elements. This also applies to the columns and higher dimensions. So, if there is a third dimension (layers), each layer can have a different number of elements. Jagged arrays are made possible when multidimensioned arrays are actually arrays of arrays. For example, a matrix would appear as an array of single-dimensioned arrays.

C, C++, and Java support jagged arrays but not rectangular arrays. In those languages, a reference to an element of a multidimensioned array uses a separate pair of brackets for each dimension. For example,

```
myArray[3][7]
```

C# and F# support both rectangular and jagged arrays. For rectangular arrays, all subscript expressions in references to elements are placed in a single pair of brackets. For example,

```
myArray[3, 7]
```

6.5.7 Slices

A **slice** of an array is some substructure of that array. For example, if `A` is a matrix, then the first row of `A` is one possible slice, as are the last row and the first column. It is important to realize that a slice is not a new data type. Rather, it is a mechanism for referencing part of an array as a unit. If arrays cannot be manipulated as units in a language, that language has no use for slices.

Consider the following Python declarations:

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
```

Recall that the default lower bound for Python arrays is 0. The syntax of a Python slice reference is a pair of numeric expressions separated by a colon. The first is the first subscript of the slice; the second is the first subscript after the last subscript in the slice. Therefore, `vector[3:6]` is a three-element array with the fourth through sixth elements of `vector` (those elements with the subscripts `3`, `4`, and `5`). A row of a matrix is specified by giving just one subscript. For example, `mat[1]` refers to the second row of `mat`; a part of a row can be specified with the same syntax as a part of a single-dimensioned array. For example, `mat[0][0:2]` refers to the first and second element of the first row of mat, which is `[1, 2]`.

Python also supports more complex slices of arrays. For example, `vector[0:7:2]` references every other element of `vector`, up to but not including the element with the subscript 7, starting with the subscript `0`, which is `[2, 6, 10, 14]`.

Perl supports slices of two forms, a list of specific subscripts or a range of subscripts. For example,

```
@list[1..5] = @list2[3, 5, 7, 9, 13];
```

Notice that slice references use array names, not scalar names, because slices are arrays (not scalars).

Ruby supports slices with the `slice` method of its `Array` object, which can take three forms of parameters. A single integer expression parameter is interpreted as a subscript, in which case `slice` returns the element with the given subscript. If `slice` is given two integer expression parameters, the first is interpreted as a beginning subscript and the second is interpreted as the number of elements in the slice. For example, suppose `list` is defined as follows:

```
list = [2, 4, 6, 8, 10]
```

`list.slice(2, 2)` returns `[6, 8]`. The third parameter form for `slice` is a range, which has the form of an integer expression, two periods, and a second integer expression. With a range parameter, `slice` returns an array of the element with the given range of subscripts. For example, `list.slice (1..3)` returns `[4, 6, 8]`.

6.5.8 Evaluation

Arrays have been included in virtually all programming languages. The primary advances since their introduction in Fortran I have been slices and dynamic arrays. As discussed in Section 6.6, the latest advances in arrays have been in associative arrays.

6.5.9 Implementation of Array Types

Implementing arrays requires considerably more compile-time effort than does implementing primitive types. The code to allow accessing of array elements must be generated at compile time. At run time, this code must be executed to produce element addresses. There is no way to precompute the address to be accessed by a reference such as

```
list[k]
```

A single-dimensioned array is implemented as a list of adjacent memory cells. Suppose the array `list` is defined to have a subscript range lower bound of `0`. The access function for `list` is often of the form

address (`list[k]`) = address (`list[0]`) + `k` * element_size

where the first operand of the addition is the constant part of the access function, and the second is the variable part.

If the element type is statically bound and the array is statically bound to storage, then the value of the constant part can be computed before run time. However, the addition and multiplication operations must be done at run time.

The generalization of this access function for an arbitrary lower bound is

address (`list[k]`) = address (`list`[lower_bound]) +
((`k` −lower_bound) * element_size)

The compile-time descriptor for single-dimensioned arrays can have the form shown in Figure 6.4. The descriptor includes information required to construct the access function. If run-time checking of index ranges is not done and the attributes are all static, then only the access function is required during execution; no descriptor is needed. If run-time checking of index ranges is

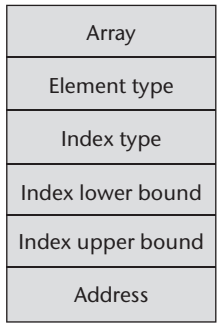


Figure 6.4
Compile-time descriptor for single-dimensioned arrays

done, then those index ranges may need to be stored in a run-time descriptor. If the subscript ranges of a particular array type are static, then the ranges may be incorporated into the code that does the checking, thus eliminating the need for the run-time descriptor. If any of the descriptor entries are dynamically bound, then those parts of the descriptor must be maintained at run time.

True multidimensional arrays, that is, those that are not arrays of arrays, are more complex to implement than single-dimensioned arrays, although the extension to more dimensions is straightforward. Hardware memory is linear—just a simple sequence of bytes. So values of data types that have two or more dimensions must be mapped onto the single-dimensioned memory. There are two ways in which multidimensional arrays can be mapped to one dimension: row major order and column major order (not used in any widely used language). In **row major order**, the elements of the array that have as their first subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the first subscript, and so forth. If the array is a matrix, it is stored by rows. For example, if the matrix had the values

3 4 7
6 2 5
1 3 8

it would be stored in row major order as

3, 4, 7, 6, 2, 5, 1, 3, 8

The access function for a multidimensional array is the mapping of its base address and a set of index values to the address in memory of the element specified by the index values. The access function for two-dimensional arrays stored in row major order can be developed as follows. In general, the address of an element is the base address of the structure plus the element size times the number of elements that precede it in the structure. For a matrix in row major order, the number of elements that precede an element is the number of rows above the element times the size of a row, plus the number of elements to the left of the element in its row. This is illustrated in Figure 6.5, in which we assume that subscript lower bounds are all zero.

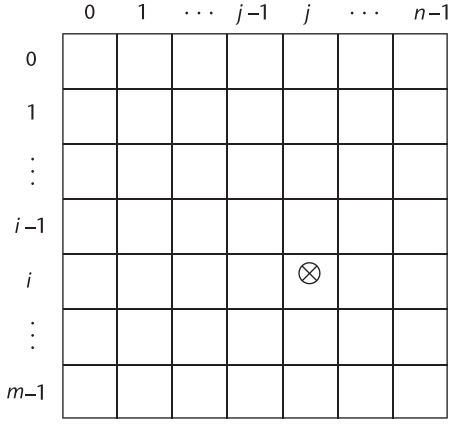


Figure 6.5

The location of the `[i,j]` element in a matrix

To get an actual address value, the number of elements that precede the desired element must be multiplied by the element size. Now, the access function can be written as

location(a[i,j]) = address of a[0, 0]
+ ((((number of rows above the i*th* row) * (size of a row))
+ (number of elements left of the j*th* column)) *
element size)

Because the number of rows above the i*th* row is i and the number of elements to the left of the j*th* column is j, we have

location(a[i, j]) = address of a[0, 0] + (((i * n) + j) *
element_size)

where n is the number of elements per row. The first term is the constant part and the last is the variable part.

The generalization to arbitrary lower bounds results in the following access function:

location(a[i, j]) = address of a[row_lb, col_lb]
= (((i − row_lb) * n) + (j − col_lb)) * element_size

where row_lb is the lower bound of the rows and col_lb is the lower bound of the columns. This can be rearranged to the form

location(a[i, j]) = address of a[row_lb, col_lb]
− (((row_lb * n) + col_lb) * element_size)
+ (((i * n) + j) * element_size)

where the first two terms are the constant part and the last is the variable part. This can be generalized rather easily to an arbitrary number of dimensions.

For each dimension of an array, one add and one multiply instruction are required for the access function. Therefore, accesses to elements of arrays with several subscripts are costly. The compile-time descriptor for a multidimensional array is shown in Figure 6.6.

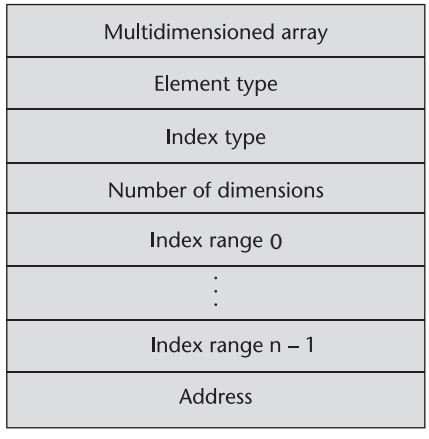


Figure 6.6

A compile-time descriptor for a multidimensional array

6.6 Associative Arrays

An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called **keys**. In the case of non-associative arrays, the indices never need to be stored (because of their regularity). In an associative array, however, the user-defined keys must be stored in the structure. So each element of an associative array is in fact a pair of entities, a key and a value. We use Perl's design of associative arrays to illustrate this data structure. Associative arrays are also supported directly by Python, Ruby, and Swift and by the standard class libraries of Java, C++, C#, and F#.

The only design issue that is specific for associative arrays is the form of references to their elements.

6.6.1 Structure and Operations

In Perl, associative arrays are called **hashes**, because in the implementation their elements are stored and retrieved with hash functions. The namespace for Perl hashes is distinct: Every hash variable name must begin with a percent sign (`%`). Each hash element consists of two parts: a key, which is a string, and a value, which is a scalar (number, string, or reference). Hashes can be set to literal values with the assignment statement, as in

```
%salaries = ("Gary" => 75000, "Perry" => 57000,
             "Mary" => 55750, "Cedric" => 47850);
```

Individual element values are referenced using notation that is similar to that used for Perl arrays. The key value is placed in braces and the hash name is replaced by a scalar variable name that is the same except for the first character. Although hashes are not scalars, the value parts of hash elements are scalars, so references to hash element values use scalar names. Recall that scalar variable names begin with dollar signs (`$`). So, an assignment of `58850` to the element of `%salaries` with the key `"Perry"` would appear as follows:

```
$salaries{"Perry"} = 58850;
```

A new element is added using the same assignment statement form. An element can be removed from the hash with the **`delete`** operator, as in the following:

```
delete $salaries{"Gary"};
```

The entire hash can be emptied by assigning the empty literal to it, as in the following:

```
@salaries = ();
```

The size of a Perl hash is dynamic: It grows when an element is added and shrinks when an element is deleted, and also when it is emptied by assignment

of the empty literal. The `exists` operator returns true or false, depending on whether its operand key is an element in the hash. For example,

```
if (exists $salaries{"Shelly"}) . . .
```

The `keys` operator, when applied to a hash, returns an array of the keys of the hash. The `values` operator does the same for the values of the hash. The `each` operator iterates over the element pairs of a hash.

Python's associative arrays, which are called **dictionaries**, are similar to those of Perl, except the values are all references to objects. The associative arrays supported by Ruby are similar to those of Python, except that the keys can be any object,[6] rather than just strings. There is a progression from Perl's hashes, in which the keys must be strings, to PHP's arrays, in which the keys can be integers or strings, to Ruby's hashes, in which any type object can be a key.

PHP's arrays are both normal arrays and associative arrays. They can be treated as either. The language provides functions that allow both indexed and 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.

Swift's associative arrays are called dictionaries. The keys can be of one specific type, but the values can be of mixed types, in which case they are objects.

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.

---

6. Objects that change do not make good keys, because the changes could change the hash function value. Therefore, arrays and hashes are never used as keys.

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++, C#, and Swift, records are supported with the **`struct`** data type. In C++, structures are a minor variation on classes. In C#, structs also are related to classes, but are quite different from them. 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.

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

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, if `Middle` is a field in the `Employee_Name` record which is embedded in the `Employee_Record` record, it would be referenced with the following:

```
Employee_Record.Employee_Name.Middle
```

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. In the COBOL example above the field reference is 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 therefore it is 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.

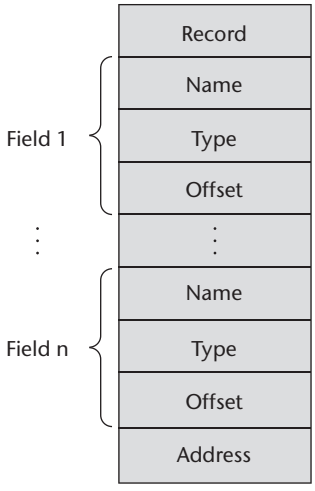
Records and arrays represent thoughtful and efficient methods of fulfilling two separate but related applications of data structures.

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

Figure 6.7

A compile-time descriptor for a record



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.

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. The asterisk can be misleading. It is used to separate the tuple components, indicating a type product, and has nothing to do with arithmetic.

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. In Swift, tuples are passed by value, so they are sometimes used to pass data to a function when the function is not to change that data.

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, is their major design issue.

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 ML, Haskell, and F#.

6.10.3 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:[7]

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

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
```

7. 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.

6.10.4 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 ML, Haskell, and F#.

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

6.10.5 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.

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++). 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 that does not have pointers or dynamic storage. This would require the programmer to provide and maintain a pool of available tree nodes, which would probably be implemented in parallel arrays. Also, 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 also could 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 many 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 6.8.

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

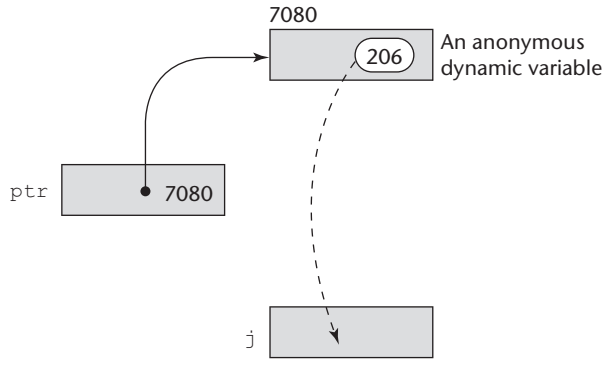


Figure 6.8

The assignment operation `j = *ptr`

refer to that field. The operator ->, when used between a pointer to a struct and a field of that struct, combines dereferencing and field reference. For example, the expression `p -> age` is equivalent to `(*p).age`.

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 to 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 C and C++

In C and C++, pointers can be used in the same ways as addresses are used in assembly languages. This means they are extremely flexible but must be used with great care. This design offers no solutions to the dangling pointer or lost heap-dynamic variable problems. However, the fact that pointer arithmetic is possible in C and C++ makes their pointers more interesting than those of the other programming languages.

C and C++ pointers can point at any variable, regardless of where it is allocated. In fact, they can point anywhere in memory, whether there is a variable there or not, which is one of the dangers of such pointers.

In C and C++, the asterisk (`*`) denotes the dereferencing operation and the ampersand (`&`) denotes the operator for producing the address of a variable. For example, consider the following code:

```
int *ptr;
int count, init;
. . .
ptr = &init;
count = *ptr;
```

The assignment to the variable `ptr` sets it to the address of `init`. The assignment to `count` dereferences `ptr` to produce the value at `init`, which is then assigned to `count`. So, the effect of the two assignment statements is to assign the value of `init` to `count`. Notice that the declaration of a pointer specifies its domain type.

Pointers can be assigned the address value of any variable of the correct domain type, or they can be assigned the constant zero, which is used for `nil`.

Pointer arithmetic is also possible in some restricted forms. For example, if `ptr` is a pointer variable that is declared to point at some variable of some data type, then

```
ptr + index
```

is a legal expression. The semantics of such an expression is as follows. Instead of simply adding the value of `index` to `ptr`, the value of `index` is first scaled by the size of the memory cell (in memory units) to which `ptr` is pointing (its base type). For example, if `ptr` points to a memory cell for a type that is four memory units in size, then `index` is multiplied by 4, and the result is added to `ptr`. The primary purpose of this sort of address arithmetic is array manipulation. The following discussion is related to single-dimensioned arrays only.

In C and C++, all arrays use zero as the lower bound of their subscript ranges, and array names without subscripts always refer to the address of the first element. Consider the following declarations:

```
int list [10];
int *ptr;
```

Now consider the assignment

```
ptr = list;
```

This assigns the address of `list[0]` to `ptr`. Given this assignment, the following are true:

- `*(ptr + 1)` is equivalent to `list[1]`.
- `*(ptr + index)` is equivalent to `list[index]`.
- `ptr[index]` is equivalent to `list[index]`.

It is clear from these statements that the pointer operations include the same scaling that is used in indexing operations. Furthermore, pointers to arrays can be indexed as if they were array names.

Pointers in C and C++ can point to functions. This feature is used to pass functions as parameters to other functions. Pointers are also used for parameter passing, as discussed in Chapter 9.

C and C++ include pointers of type **`void *`**, which can point at values of any type. In effect they are generic pointers. However, type checking is not a problem with **`void *`** pointers, because these languages disallow dereferencing them. One common use of **`void *`** pointers is as the types of parameters of functions that operate on memory. For example, suppose we wanted a function to move a sequence of bytes of data from one place in memory to another. It would be most general if it could be passed two pointers of any type. This would be legal if the corresponding formal parameters in the function were **`void *`** type. The function could then convert them to **`char *`** type and do the operation, regardless of what type pointers were sent as actual parameters.

6.11.5 Reference Types

A **reference type** variable is similar to a pointer, with one important and fundamental difference: A pointer refers to an address in memory, while a reference refers to an object or a value in memory. As a result, although it is natural to perform arithmetic on addresses, it is not sensible to do arithmetic on references.

C++ includes a special kind of reference type that is used primarily for the formal parameters in function definitions. A C++ reference type variable is a constant pointer that is always implicitly dereferenced. Because a C++ reference type variable is a constant, it must be initialized with the address of some variable in its definition, and after initialization a reference type variable can

never be set to reference any other variable. The implicit dereference prevents assignment to the address value of a reference variable.

Reference type variables are specified in definitions by preceding their names with ampersands (&). For example,

```
int result = 0;
 int &ref_result = result;
. . .
ref_result = 100;
```

In this code segment, `result` and `ref_result` are aliases.

When used as formal parameters in function definitions, C++ reference types provide for two-way communication between the caller function and the called function. This is not possible with nonpointer primitive parameter types, because C++ parameters are passed by value. Passing a pointer as a parameter accomplishes the same two-way communication, but pointer formal parameters require explicit dereferencing, making the code less readable and less safe. Reference parameters are referenced in the called function exactly as are other parameters. The calling function need not specify that a parameter whose corresponding formal parameter is a reference type is anything unusual. The compiler passes addresses, rather than values, to reference parameters.

In their quest for increased safety over C++, the designers of Java removed C++-style pointers altogether. Unlike C++ reference variables, Java reference variables can be assigned to refer to different class instances; they are not constants. All Java class instances are referenced by reference variables. That is, in fact, the only use of reference variables in Java. These issues are discussed further in Chapter 12.

In the following, `String` is a standard Java class:

```
String str1;
. . .
str1 = "This is a Java literal string";
```

In this code, `str1` is defined to be a reference to a `String` class instance or object. It is initially set to null. The subsequent assignment sets `str1` to reference the `String` object, `"This is a Java literal string"`.

Because Java class instances are implicitly deallocated (there is no explicit deallocation operator), there cannot be dangling references in Java.

C# includes both the references of Java and the pointers of C++. However, the use of pointers is strongly discouraged. In fact, any subprogram that uses pointers must include the **`unsafe`** modifier. Note that although objects pointed to by references are implicitly deallocated, that is not true for objects pointed to by pointers. Pointers were included in C# primarily to allow C# programs to interoperate with C and C++ code.

All variables in the object-oriented languages Smalltalk, Python, and Ruby are references. They are always implicitly dereferenced. Furthermore, the direct values of these variables cannot be accessed.

6.11.6 Evaluation

The problems of dangling pointers and garbage have already been discussed at length. The problems of heap management are discussed in Section 6.11.7.3.

Pointers have been compared with the goto. The goto statement widens the range of statements that can be executed next. Pointer variables widen the range of memory cells that can be referenced by a variable. Perhaps the most damning statement about pointers was made by Hoare (1973): "Their introduction into high-level languages has been a step backward from which we may never recover."

On the other hand, pointers are essential in some kinds of programming applications. For example, pointers are necessary to write device drivers, in which specific absolute addresses must be accessed.

The references of Java and C# provide some of the flexibility and the capabilities of pointers, without the hazards. It remains to be seen whether programmers will be willing to trade the full power of C and C++ pointers for the greater safety of references. The extent to which C# programs use pointers will be one measure of this.

6.11.7 Implementation of Pointer and Reference Types

In most languages, pointers are used in heap management. The same is true for Java and C# references, as well as the variables in Smalltalk and Ruby, so we cannot treat pointers and references separately. First, we briefly describe how pointers and references are represented internally. We then discuss two possible solutions to the dangling pointer problem. Finally, we describe the major problems with heap-management techniques.

6.11.7.1 Representations of Pointers and References

In most larger computers, pointers and references are single values stored in memory cells. However, in early microcomputers based on Intel microprocessors, addresses have two parts: a segment and an offset. So, pointers and references are implemented in these systems as pairs of 16-bit cells, one for each of the two parts of an address.

6.11.7.2 Solutions to the Dangling-Pointer Problem

There have been several proposed solutions to the dangling-pointer problem. Among these are **tombstones** (Lomet, 1975), in which every heap-dynamic variable includes a special cell, called a tombstone, that is itself a pointer to the heap-dynamic variable. The actual pointer variable points only at tombstones and never to heap-dynamic variables. When a heap-dynamic variable is deallocated, the tombstone remains but is set to nil, indicating that the heap-dynamic variable no longer exists. This approach prevents a pointer from ever pointing to a deallocated variable. Any reference to any pointer that points to a `nil` tombstone can be detected as an error.

Tombstones are costly in both time and space. Because tombstones are never deallocated, their storage is never reclaimed. Every access to a heap-dynamic variable through a tombstone requires one more level of indirection, which requires an additional machine cycle on most computers. Apparently none of the designers of the more popular languages have found the additional safety to be worth this additional cost, because no widely used language uses tombstones.

An alternative to tombstones is the **locks-and-keys approach** used in the implementation of UW-Pascal (Fischer and LeBlanc, 1977, 1980). In this compiler, pointer values are represented as ordered pairs (key, address), where the key is an integer value. Heap-dynamic variables are represented as the storage for the variable plus a header cell that stores an integer lock value. When a heap-dynamic variable is allocated, a lock value is created and placed both in the lock cell of the heap-dynamic variable and in the key cell of the pointer that is specified in the call to **`new`**. Every access to the dereferenced pointer compares the key value of the pointer to the lock value in the heap-dynamic variable. If they match, the access is legal; otherwise the access is treated as a run-time error. Any copies of the pointer value to other pointers must copy the key value. Therefore, any number of pointers can reference a given heap-dynamic variable. When a heap-dynamic variable is deallocated with **`dispose`**, its lock value is cleared to an illegal lock value. Then, if a pointer other than the one specified in the **`dispose`** is dereferenced, its address value will still be intact, but its key value will no longer match the lock, so the access will not be allowed.

Of course, the best solution to the dangling-pointer problem is to take deallocation of heap-dynamic variables out of the hands of programmers. If programs cannot explicitly deallocate heap-dynamic variables, there will be no dangling pointers. To do this, the run-time system must implicitly deallocate heap-dynamic variables when they are no longer useful. Lisp systems have always done this. Both Java and C# also use this approach for their reference variables. Recall that C#'s pointers do not include implicit deallocation.

6.11.7.3 Heap Management

Heap management can be a very complex run-time process. We examine the process in two separate situations: one in which all heap storage is allocated and deallocated in units of a single size, and one in which variable-size segments are allocated and deallocated. Note that for deallocation, we discuss only implicit approaches. Our discussion will be brief and far from comprehensive, since a thorough analysis of these processes and their associated problems is not so much a language design issue as it is an implementation issue.

Single-Size Cells The simplest situation is when all allocation and deallocation is of single-size cells. It is further simplified when every cell already contains a pointer. This is the scenario of many implementations of Lisp, where the problems of dynamic storage allocation were first encountered on a large scale. All Lisp programs and most Lisp data consist of cells in linked lists.

In a single-size allocation heap, all available cells are linked together using the pointers in the cells, forming a list of available space. Allocation is a simple matter of taking the required number of cells from this list when they are needed. Deallocation is a much more complex process. A heap-dynamic variable can be pointed to by more than one pointer, making it difficult to determine when the variable is no longer useful to the program. Simply because one pointer is disconnected from a cell obviously does not make it garbage; there could be several other pointers still pointing to the cell.

In Lisp, several of the most frequent operations in programs create collections of cells that are no longer accessible to the program and therefore should be deallocated (put back on the list of available space). One of the fundamental design goals of Lisp was to ensure that reclamation of unused cells would not be the task of the programmer but rather that of the run-time system. This goal left Lisp implementors with the fundamental design question: When should deallocation be performed?

There are several different approaches to garbage collection. The two most common traditional techniques are in some ways opposite processes. These are named **reference counters**, in which reclamation is incremental and is done when inaccessible cells are created, and **mark-sweep**, in which reclamation occurs only when the list of available space becomes empty. These two methods are sometimes called the **eager approach** and the **lazy approach**, respectively. Many variations of these two approaches have been developed. In this section, however, we discuss only the basic processes.

The reference counter method of storage reclamation accomplishes its goal by maintaining in every cell a counter that stores the number of pointers that are currently pointing at the cell. Embedded in the decrement operation for the reference counters, which occurs when a pointer is disconnected from the cell, is a check for a zero value. If the reference counter reaches zero, it means that no program pointers are pointing at the cell, and it has thus become garbage and can be returned to the list of available space.

There are three distinct problems with the reference counter method. First, if storage cells are relatively small, the space required for the counters is significant. Second, some execution time is obviously required to maintain the counter values. Every time a pointer value is changed, the cell to which it was pointing must have its counter decremented, and the cell to which it is now pointing must have its counter incremented. In a language like Lisp, in which nearly every action involves changing pointers, that can be a significant portion of the total execution time of a program. Of course, if pointer changes are not too frequent, this is not a problem. Some of the inefficiency of reference counters can be eliminated by an approach named **deferred reference counting**, which avoids reference counters for some pointers. The third problem is that complications arise when a collection of cells is connected circularly. The problem here is that each cell in the circular list has a reference counter value of at least 1, which prevents it from being collected and placed back on the list of available space. A solution to this problem can be found in Friedman and Wise (1979).

The advantage of the reference counter approach is that it is intrinsically incremental. Its actions are interleaved with those of the application, so it never causes significant delays in the execution of the application.

The original mark-sweep process of garbage collection operates as follows: The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary, without regard for storage reclamation (allowing garbage to accumulate), until it has allocated all available cells. At this point, a mark-sweep process is begun to gather all the garbage left floating around in the heap. To facilitate the process, every heap cell has an extra indicator bit or field that is used by the collection algorithm.

The mark-sweep process consists of three distinct phases. First, all cells in the heap have their indicators set to indicate they are garbage. This is, of course, a correct assumption for only some of the cells. The second part, called the marking phase, is the most difficult. Every pointer in the program is traced into the heap, and all reachable cells are marked as not being garbage. After this, the third phase, called the sweep phase, is executed: All cells in the heap that have not been specifically marked as still being used are returned to the list of available space.

To illustrate the flavor of algorithms used to mark the cells that are currently in use, we provide the following simple version of a marking algorithm. We assume that all heap-dynamic variables, or heap cells, consist of an information part; a part for the mark, named `marker`; and two pointers named `llink` and `rlink`. These cells are used to build directed graphs with at most two edges leading from any node. The marking algorithm traverses all spanning trees of the graphs, marking all cells that are found. Like other graph traversals, the marking algorithm uses recursion.

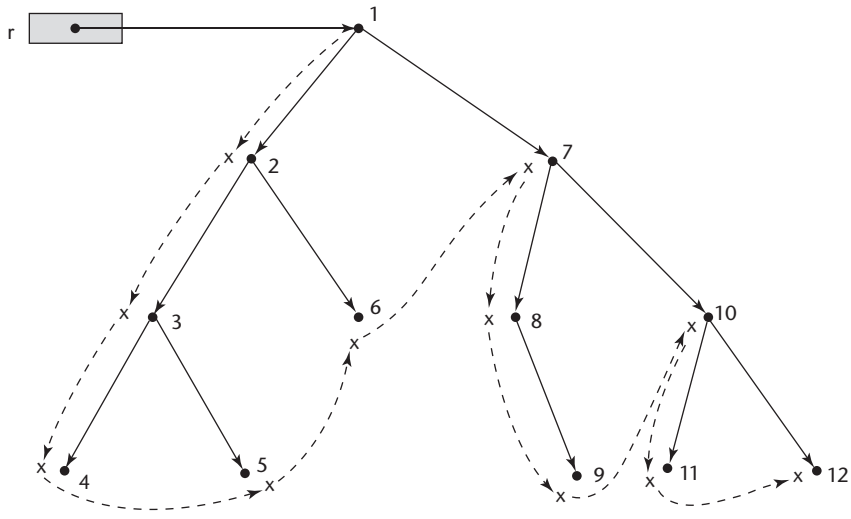
```
for every pointer r  do
    mark(r)

void mark(void  * ptr) {
    if (ptr != 0)
      if (*ptr.marker is not marked) {
        set *ptr.marker
        mark(*ptr.llink)
        mark(*ptr.rlink)
      }
}
```

An example of the actions of this procedure on a given graph is shown in Figure 6.9. This simple marking algorithm requires a great deal of storage (for stack space to support recursion). A marking process that does not require additional stack space was developed by Schorr and Waite (1967). Their method reverses pointers as it traces out linked structures. Then, when the end of a list is reached, the process can follow the pointers back out of the structure.

The most serious problem with the original version of mark-sweep was that it was done too infrequently—only when a program had used all or nearly all of

Figure 6.9

An example of the actions of the marking algorithm



Dashed lines show the order of node_marking

the heap storage. Mark-sweep in that situation takes a good deal of time, because most of the cells must be traced and marked as being currently used. This causes a significant delay in the progress of the application. Furthermore, the process may yield only a small number of cells that can be placed on the list of available space. This problem has been addressed in a variety of improvements. For example, **incremental mark-sweep** garbage collection occurs more frequently, long before memory is exhausted, making the process more effective in terms of the amount of storage that is reclaimed. Also, the time required for each run of the process is obviously shorter, thus reducing the delay in application execution. Another alternative is to perform the mark-sweep process on parts, rather than all of the memory associated with the application, at different times. This provides the same kinds of improvements as incremental mark-sweep.

Both the marking algorithms for the mark-sweep method and the processes required by the reference counter method can be made more efficient by use of the pointer rotation and slide operations that are described by Suzuki (1982).

Variable-Size Cells Managing a heap from which variable-size cells[8] are allocated has all the difficulties of managing one for single-size cells, but also has additional problems. Unfortunately, variable-size cells are required by most programming languages. The additional problems posed by variable-size cell management depend on the method used. If mark-sweep is used, the following additional problems occur:

8. The cells have variable sizes because these are abstract cells, which store the values of variables, regardless of their types. Furthermore, a variable could be a structured type.

- The initial setting of the indicators of all cells in the heap to indicate that they are garbage is difficult. Because the cells are different sizes, scanning them is a problem. One solution is to require each cell to have the cell size as its first field. Then the scanning can be done, although it takes slightly more space and somewhat more time than its counterpart for fixed-size cells.
- The marking process is nontrivial. How can a chain be followed from a pointer if there is no predefined location for the pointer in the pointed-to cell? Cells that do not contain pointers at all are also a problem. Adding an internal pointer to each cell, which is maintained in the background by the run-time system, will work. However, this background maintenance processing adds both space and execution time overhead to the cost of running the program.
- Maintaining the list of available space is another source of overhead. The list can begin with a single cell consisting of all available space. Requests for segments simply reduce the size of this block. Reclaimed cells are added to the list. The problem is that before long, the list becomes a long list of various-size segments, or blocks. This slows allocation because requests cause the list to be searched for sufficiently large blocks. Eventually, the list may consist of a large number of very small blocks, which are not large enough for most requests. At this point, adjacent blocks may need to be collapsed into larger blocks. Alternatives to using the first sufficiently large block on the list can shorten the search but require the list to be ordered by block size. In either case, maintaining the list is additional overhead.

If reference counters are used, the first two problems are avoided, but the available-space list-maintenance problem remains.

For a comprehensive study of memory management problems, see Wilson (2005).

6.12 Optional Types

There are situations in programming when there is a need to be able to indicate that a variable does not currently have a value. Some older languages use zero as a nonvalue for numeric variables. This approach has the disadvantage of not being able to distinguish between when the variable is supposed to have the zero value and when the zero indicates that it has no value. Some newer languages provide types that can have a normal value or a special value to indicate that their variables have no value. Variables that have this capability are called **optional types**. Optional types are now directly supported in C#, F#, and Swift, among others.

C# has two categories of variables, value and reference types. Reference types, which are classes, are optional types by their nature. The `null` value indicates that a reference type has no value. Value types, which are all struct types, can be declared to be optional types, which allows them to have the value `null`. A variable is declared to be an optional type by following its type name with a question mark `(?)`, as in

```
int? x;
```

To determine whether a variable has a normal value, it can be tested against **null**, as in

```
int? x;
. . .
if(x == null)
   Console.WriteLine("x has no value");
else
   Console.WriteLine("The value of x is: {0}", x);
```

Swift's optional types are similar to those of C#, except that the nonvalue is named **nil**, instead of **null**. The Swift version of the above code is:

```
var Int? x;
. . .
if x == nil
  print("x has no value")
else
  print("The value of x is: \(x)")
```

6.13 Type Checking

For our discussion of type checking, the concept of operands and operators is generalized to include subprograms and assignment statements. Subprograms will be thought of as operators whose operands are their parameters. The assignment symbol will be thought of as a binary operator, with its target variable and its expression being the operands.

Type checking is the activity of ensuring that the operands of an operator are of compatible types. A **compatible** type is one that either is legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type. This automatic conversion is called a **coercion**. For example, if an **int** variable and a **float** variable are added in Java, the value of the **int** variable is coerced to **float** and a floating-point add is done.

A **type error** is the application of an operator to an operand of an inappropriate type. For example, in the original version of C, if an **int** value was passed to a function that expected a **float** value, a type error would occur (because compilers for that language did not check the types of parameters).

If all bindings of variables to types are static in a language, then type checking can nearly always be done statically. Dynamic type binding requires type checking at run time, which is called **dynamic type checking**.

Some languages, such as JavaScript and PHP, because of their dynamic type binding, allow only dynamic type checking. It is better to detect errors at compile time than at run time, because the earlier correction is usually less costly. The penalty for static checking is reduced programmer flexibility. Fewer shortcuts and tricks are possible. Such techniques, though, are now generally recognized to be error prone and detrimental to readability.

Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution. Such memory cells can be created with C and C++ unions and the discriminated unions of ML, Haskell, and F#. In these cases, type checking, if done, must be dynamic and requires the run-time system to maintain the type of the current value of such memory cells. So, even though all variables are statically bound to types in languages such as C++, not all type errors can be detected by static type checking.

6.14 Strong Typing

One of the ideas in language design that became prominent in the so-called structured-programming revolution of the 1970s was **strong typing**. Strong typing is widely acknowledged as being a highly valuable language characteristic. Unfortunately, it is often loosely defined, and it is sometimes used in computing literature without being defined at all.

A programming language is **strongly typed** if type errors are always detected. This requires that the types of all operands can be determined, either at compile time or at run time. The importance of strong typing lies in its ability to detect all misuses of variables that result in type errors. A strongly typed language also allows the detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type.

C and C++ are not strongly typed languages because both include union types, which are not type checked.

ML is strongly typed, even though the types of some function parameters may not be known at compile time. F# is strongly typed.

Java and C#, although they are based on C++, are nearly strongly typed. Types can be explicitly cast, which could result in a type error. However, there are no implicit ways type errors can go undetected.

The coercion rules of a language have an important effect on the value of type checking. For example, expressions are strongly typed in Java. However, an arithmetic operator with one floating-point operand and one integer operand is legal. The value of the integer operand is coerced to floating-point, and a floating-point operation takes place. This is what is usually intended by the programmer. However, the coercion also results in a loss of one of the benefits of strong typing—error detection. For example, suppose a program had the `int` variables `a` and `b` and the `float` variable `d`. Now, if a programmer meant to type `a + b`, but mistakenly typed `a + d`, the error would not be detected by the compiler. The value of `a` would simply be coerced to `float`. So, the value of strong typing is weakened by coercion. Languages with a great deal of coercion, like C, and C++, are less reliable than those with no coercion, such as ML and F#. Java and C# have half as many assignment type coercions as C++, so their error detection is better than that of C++, but still not nearly as effective as that of ML and F#. The issue of coercion is examined in detail in Chapter 7.

6.15 Type Equivalence

The idea of type compatibility was defined when the issue of type checking was introduced. The compatibility rules dictate the types of operands that are acceptable for each of the operators and thereby specify the possible type errors of the language.[9] The rules are called compatibility because in some cases the type of an operand can be implicitly converted by the compiler or run-time system to make it acceptable for the operator.

The type compatibility rules are simple and rigid for the predefined scalar types. However, in the cases of structured types, such as arrays and records and some user-defined types, the rules are more complex. Coercion of these types is rare, so the issue is not type compatibility, but type equivalence. That is, two types are **equivalent** if an operand of one type in an expression can be substituted for one of the other type, without coercion. Type equivalence is a strict form of type compatibility—compatibility without coercion. The central issue here is how type equivalence is defined.

The design of the type equivalence rules of a language is important, because it influences the design of the data types and the operations provided for values of those types. With the types discussed here, there are very few predefined operations. Perhaps the most important result of two variables being of equivalent types is that either one can have its value assigned to the other.

There are two approaches to defining type equivalence: name type equivalence and structure type equivalence. **Name type equivalence** means that two variables have equivalent types if they are defined either in the same declaration or in declarations that use the same type name. **Structure type equivalence** means that two variables have equivalent types if their types have identical structures. There are some variations of these two approaches, and many languages use combinations of them.

Name type equivalence is easy to implement but is more restrictive. Under a strict interpretation, a variable whose type is a subrange of the integers would not be equivalent to an integer type variable. For example, supposing Ada used strict name type equivalence, consider the following Ada code:

```
type Indextype is 1..100;
count : Integer;
index : Indextype;
```

The types of the variables `count` and `index` would not be equivalent; `count` could not be assigned to `index` or vice versa.

Another problem with name type equivalence arises when a structured or user-defined type is passed among subprograms through parameters. Such a type must be defined only once, globally. A subprogram cannot state the type

9. Type compatibility is also an issue in the relationship between the actual parameters in a subprogram call and the formal parameters of the subprogram definition. This issue is discussed in Chapter 9.

of such formal parameters in local terms. This was the case with the original version of Pascal.

Note that to use name type equivalence, all types must have names. Most languages allow users to define types that are anonymous—they do not have names. For a language to use name type equivalence, such types must implicitly be given internal names by the compiler.

Structure type equivalence is more flexible than name type equivalence, but it is more difficult to implement. Under name type equivalence, only the two type names must be compared to determine equivalence. Under structure type equivalence, however, the entire structures of the two types must be compared. This comparison is not always simple. (Consider a data structure that refers to its own type, such as a linked list.) Other questions can also arise. For example, are two record (or **`struct`**) types equivalent if they have the same structure but different field names? Are two single-dimensioned array types in a language that allows lower bounds of array subscript ranges to be set in their declarations equivalent if they have the same element type but have subscript ranges of `0..10` and `1..11`? Are two enumeration types equivalent if they have the same number of components but spell the literals differently?

Another difficulty with structure type equivalence is that it disallows differentiating between types with the same structure. For example, consider the following Ada-like declarations:

```
type Celsius = Float;
     Fahrenheit = Float;
```

The types of variables of these two types are considered equivalent under structure type equivalence, allowing them to be mixed in expressions, which is surely undesirable in this case, considering the difference indicated by the type's names. In general, types with different names are likely to be abstractions of different categories of problem values and should not be considered equivalent.

Ada uses a restrictive form of name type equivalence but provides two type constructs, subtypes and derived types, that avoid the problems associated with name type equivalence. A **derived type** is a new type that is based on some previously defined type with which it is not equivalent, although it may have identical structure. Derived types inherit all the properties of their parent types. Consider the following example:

```
type Celsius is new Float;
type Fahrenheit is new Float;
```

The types of variables of these two derived types are not equivalent, although their structures are identical. Furthermore, variables of neither type is type equivalent with any other floating-point type. Literals are exempt from the rule. A literal such as `3.0` has the type universal real and is type equivalent to any floating-point type. Derived types can also include range constraints on the parent type, while still inheriting all of the parent's operations.

An Ada **subtype** is a possibly range-constrained version of an existing type. A subtype is type equivalent with its parent type. For example, consider the following declaration:

```
subtype Small_type is Integer range 0..99;
```

The type `Small_type` is equivalent to the type `Integer`.

Note that Ada's derived types are very different from Ada's subrange types. For example, consider the following type declarations:

```
type Derived_Small_Int is new Integer range 1..100;
subtype Subrange_Small_Int is Integer range 1..100;
```

Variables of both types, `Derived_Small_Int` and `Subrange_Small_Int`, have the same range of legal values and both inherit the operations of `Integer`. However, variables of type `Derived_Small_Int` are not compatible with any `Integer` type. On the other hand, variables of type `Subrange_Small_Int` are compatible with variables and constants of `Integer` type and any subtype of `Integer`.

For variables of an Ada unconstrained array type, structure type equivalence is used. For example, consider the following type declaration and two object declarations:

```
type Vector is array (Integer range <>) of Integer;
Vector_1: Vector (1..10);
Vector_2: Vector (11..20);
```

The types of these two objects are equivalent, even though they have different names and different subscript ranges, because for objects of unconstrained array types, structure type equivalence rather than name type equivalence is used. Because both types have 10 elements and the elements of both are of type `Integer`, they are type equivalent.

For constrained anonymous types, Ada uses a highly restrictive form of name type equivalence. Consider the following Ada declarations of constrained anonymous types:

```
A : array (1..10) of Integer;
```

In this case, `A` has an anonymous but unique type assigned by the compiler and unavailable to the program. If we also had

```
B : array (1..10) of Integer;
```

`A` and `B` would be of anonymous but distinct and not equivalent types, though they are structurally identical. The multiple declaration

```
C, D : array (1..10) of Integer;
```

creates two anonymous types, one for C and one for D, which are not equivalent. This declaration is actually treated as if it were the following two declarations:

```
C : array (1..10) of Integer;
D : array (1..10) of Integer;
```

Note that Ada's form of name type equivalence is more restrictive than the name type equivalence that is defined at the beginning of this section. If we had written instead

```
type List_10 is array (1..10) of Integer;
C, D : List_10;
```

then the types of C and D would be equivalent.

Name type equivalence works well for Ada, in part because all types, except anonymous arrays, are required to have type names (and anonymous types are given internal names by the compiler).

Type equivalence rules for Ada are more rigid than those for languages that have many coercions among types. For example, the two operands of an addition operator in Java can have virtually any combination of numeric types in the language. One of the operands will simply be coerced to the type of the other. But in Ada, there are no coercions of the operands of an arithmetic operator.

C uses both name and structure type equivalence. Every **struct**, **enum**, and **union** declaration creates a new type that is not equivalent to any other type. So, name type equivalence is used for structure, enumeration, and union types. Other nonscalar types use structure type equivalence. Array types are equivalent if they have the same type components. Also, if an array type has a constant size, it is equivalent either to other arrays with the same constant size or to with those without a constant size. Note that **typedef** in C and C++ does not introduce a new type; it simply defines a new name for an existing type. So, any type defined with **typedef** is type equivalent to its parent type. One exception to C using name type equivalence for structures, enumerations, and unions is if two structures, enumerations, or unions are defined in different files, in which case structural type equivalence is used. This is a loophole in the name type equivalence rule to allow equivalence of structures, enumerations, and unions that are defined in different files.

C++ is like C except there is no exception for structures and unions defined in different files.

In languages that do not allow users to define and name types, such as Fortran and COBOL, name equivalence obviously cannot be used.

Object-oriented languages such as Java and C++ bring another kind of type compatibility issue with them. The issue is object compatibility and its relationship to the inheritance hierarchy, which is discussed in Chapter 12.

Type compatibility in expressions is discussed in Chapter 7; type compatibility for subprogram parameters is discussed in Chapter 9.

6.16 Theory and Data Types

Type theory is a broad area of study in mathematics, logic, computer science, and philosophy. It began in mathematics in the early 1900s and later became a standard tool in logic. Any general discussion of type theory is necessarily complex, lengthy, and highly abstract. Even when restricted to computer science, type theory includes such diverse and complex subjects as typed lambda calculus, combinators, the metatheory of bounded quantification, existential types, and higher-order polymorphism. All these topics are far beyond the scope of this book.

In computer science there are two branches of type theory: practical and abstract. The practical branch is concerned with data types in commercial programming languages; the abstract branch primarily focuses on typed lambda calculus, an area of extensive research by theoretical computer scientists over the past half century. This section is restricted to a brief description of some of the mathematical formalisms that underlie data types in programming languages.

A data type defines a set of values and a collection of operations on those values. A **type system** is a set of types and the rules that govern their use in programs. Obviously, every typed programming language defines a type system. The formal model of a type system of a programming language consists of a set of types and a collection of functions that define the type rules of the language, which are used to determine the type of any expression. A formal system that describes the rules of a type system, attribute grammars, is introduced in Chapter 3.

An alternative model to attribute grammars uses a type map and a collection of functions, not associated with grammar rules, that specify the type rules. A type map is similar to the state of a program used in denotational semantics, consisting of a set of ordered pairs, with the first element of each pair being a variable's name and the second element being its type. A type map is constructed using the type declarations in the program. In a static typed language, the type map need only be maintained during compilation, although it changes as the program is analyzed by the compiler. If any type checking is done dynamically, the type map must be maintained during execution. The concrete version of a type map in a compilation system is the symbol table, constructed primarily by the lexical and syntax analyzers. Dynamic types sometimes are maintained with tags attached to values or objects.

As stated previously, a data type is a set of values, although in a data type the elements are often ordered. For example, the elements in all enumeration types are ordered. However, in a mathematical set the elements are not ordered. Despite this difference, set operations can be used on data types to describe new data types. The structured data types of programming languages are defined by type operators, or constructors that correspond to set operations.

These set operations/type constructors are briefly introduced in the following paragraphs.

A finite mapping is a function from a finite set of values, the domain set, onto values in the range set. Finite mappings model two different categories of types in programming languages, functions and arrays, although in some languages functions are not types. All languages include arrays, which are defined in terms of a mapping function that maps indices to elements in the array. For traditional arrays, the mapping is simple—integer values are mapped to the addresses of array elements; for associative arrays, the mapping is defined by a function that describes a hashing operation. The hashing function maps the keys of the associate arrays, usually character strings,[10] to the addresses of the array elements.

A Cartesian, or cross product of n sets, $S_1, S_2, \ldots, S_n$, is a set denoted $S_1 \times S_2 \times \ldots \times S_n$. Each element of the Cartesian product set has one element from each of the constituent sets. So, $S_1 \times S_2 = \{(x, y) \mid x \text{ is in } S_1 \text{ and } y \text{ is in } S_2\}$. For example, if $S_1 = \{1, 2\}$ and $S_2 = \{a, b\}$, $S_1 \times S_2 = \{(1, a), (1, b), (2, a), (2, b)\}$. A Cartesian product defines tuples in mathematics, which appear in Python, ML, Swift, and F# as a data type (see Section 6.5). Cartesian products also model records, or structs, although not exactly. Cartesian products do not have element names, but records require them. For example, consider the following C struct:

```
struct intFloat {
  int myInt;
  float myFloat;
};
```

This struct defines the Cartesian product type **`int` × `float`**. The names of the elements are `myInt` and `myFloat`.

The union of two sets, S_1 and S_2, is defined as $S_1 \cup S_2 = \{x \mid x \text{ is in } S_1 \text{ or } x \text{ is in } S_2\}$. Set union models the union data types, as described in Section 6.10.

Mathematical subsets are defined by providing a rule that elements must follow. These sets model the subtypes of Ada, although not exactly, because subtypes must consist of contiguous elements of their parent sets. Elements of mathematical sets are unordered, so the model is not perfect.

Notice that pointers, defined with type operators, such as the asterisk in C, are not defined in terms of a set operation.

This concludes our discussion of formalisms in data types, as well as our whole discussion of data types.

10. In Ruby, the associative array keys need not be character strings—they can be any type.

SUMMARY

The data types of a language are a large part of what determines that language's style and usefulness. Along with control structures, they form the heart of a language.

The primitive data types of most imperative languages include numeric, character, and Boolean types. The numeric types are often directly supported by hardware.

The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs.

Arrays are part of most programming languages. The relationship between a reference to an array element and the address of that element is given in an access function, which is an implementation of a mapping. Arrays can be either static, as in C++ arrays whose definition includes the **`static`** specifier; fixed stack-dynamic, as in C functions (without the **`static`** specifier); fixed heap-dynamic, as with Java's objects; or heap dynamic, as in Perl's arrays. Most languages allow only a few operations on complete arrays.

Records are now included in most languages. Fields of records are specified in a variety of ways. In the case of COBOL, they can be referenced without naming all of the enclosing records, although this is messy to implement and harmful to readability. In several languages that support object-oriented programming, records are supported with objects.

Tuples are similar to records, but do not have names for their constituent. parts. They are part of Python, ML, and F#.

Lists are staples of the functional programming languages, but are now also included in Python and C#.

Unions are locations that can store different type values at different times. Discriminated unions include a tag to record the current type value. A free union is one without the tag. Most languages with unions do not have safe designs for them, the exceptions being ML, Swift, and F#.

Pointers are used for addressing flexibility and to control dynamic storage management. Pointers have some inherent dangers: Dangling pointers are difficult to avoid, and memory leakage can occur.

Reference types, such as those in Java and C#, provide heap management without the dangers of pointers.

Enumeration and record types are relatively easy to implement. Arrays are also straightforward, although array element access is an expensive process when the array has several subscripts. The access function requires one addition and one multiplication for each subscript.

Pointers are relatively easy to implement, if heap management is not considered. Heap management is easy if all cells have the same size but is complicated for variable-size cell allocation and deallocation.

Optional type variables are variables that allow a nonvalue to be stored. This allows a program to indicate when a variable currently has no value.

Strong typing is the concept of requiring that all type errors be detected. The value of strong typing is increased reliability.

The type equivalence rules of a language determine what operations are legal among the structured types of a language. Name type equivalence and structure type equivalence are the two fundamental approaches to defining type equivalence.

Type theories have been developed in many areas. In computer science, the practical branch of type theory defines the types and type rules of programming languages. Set theory can be used to model most of the structured data types in programming languages.

BIBLIOGRAPHIC NOTES

A wealth of literature exists that is concerned with data type design, use, and implementation. Hoare gives one of the earliest systematic definitions of structured types in Dahl et al. (1972). A general discussion of a wide variety of data types is given in Cleaveland (1986).

Implementing run-time checks on the possible insecurities of Pascal data types is discussed in Fischer and LeBlanc (1980). Most compiler design books, such as Fischer and LeBlanc (1991) and Aho et al. (1986), describe implementation methods for data types, as do the other programming language texts, such as Pratt and Zelkowitz (2001) and Scott (2009). A detailed discussion of the problems of heap management can be found in Tenenbaum et al. (1990). Garbage-collection methods are developed by Schorr and Waite (1967) and Deutsch and Bobrow (1976). A comprehensive discussion of garbage-collection algorithms can be found in Cohen (1981) and Wilson (2005).

REVIEW QUESTIONS

1. What is a descriptor?
2. What are the advantages and disadvantages of decimal data types?
3. What are the design issues for character string types?
4. Describe the three string length options.
5. Define *ordinal*, *enumeration*, and *subrange types*.
6. What are the advantages of user-defined enumeration types?
7. In what ways are the user-defined enumeration types of C# more reliable than those of C++?
8. What are the design issues for arrays?
9. Define *static*, *fixed stack-dynamic*, *fixed heap-dynamic*, and *heap-dynamic arrays*. What are the advantages of each?
10. What happens when a nonexistent element of an array is referenced in Perl?

11. How does JavaScript support sparse arrays?
12. What languages support negative subscripts?
13. What languages support array slices with stepsizes?
14. What is an aggregate constant?
15. Define *row major order* and *column major order*.
16. What is an access function for an array?
17. What are the required entries in a Java array descriptor, and when must they be stored (at compile time or run time)?
18. What is the structure of an associative array?
19. What is the purpose of level numbers in COBOL records?
20. Define *fully qualified* and *elliptical references* to fields in records.
21. What is the primary difference between a record and a tuple?
22. Are the tuples of Python mutable?
23. What is the purpose of an F# tuple pattern?
24. In what primarily imperative language do lists serve as arrays?
25. What is the action of the Scheme function `CAR`?
26. What is the action of the F# function `tl`?
27. In what way does Scheme's `CDR` function modify its parameter?
28. On what are Python's list comprehensions based?
29. Define *union*, *free union*, and *discriminated union*.
30. Are the unions of F# discriminated?
31. What are the design issues for pointer types?
32. What are the two common problems with pointers?
33. Why are the pointers of most languages restricted to pointing at a single type variable?
34. What is a C++ reference type, and what is its common use?
35. Why are reference variables in C++ better than pointers for formal parameters?
36. What advantages do Java and C# reference type variables have over the pointers in other languages?
37. Describe the lazy and eager approaches to reclaiming garbage.
38. Why wouldn't arithmetic on Java and C# references make sense?
39. What is a compatible type?
40. Define type error.
41. Define strongly typed.
42. Why is Java not strongly typed?
43. What is a nonconverting cast?
44. What languages have no type coercions?

45. Why are C and C++ not strongly typed?
46. What is name type equivalence?
47. What is structure type equivalence?
48. What is the primary advantage of name type equivalence?
49. What is the primary disadvantage to structure type equivalence?
50. For what types does C use structure type equivalence?
51. What set operation models C's **`struct`** data type?

PROBLEM SET

1. What are the arguments for and against representing Boolean values as single bits in memory?
2. How does a decimal value waste memory space?
3. VAX minicomputers use a format for floating-point numbers that is not the same as the IEEE standard. What is this format, and why was it chosen by the designers of the VAX computers? A reference for VAX floating-point representations is Sebesta (1991).
4. Compare the tombstone and lock-and-key methods of avoiding dangling pointers, from the points of view of safety and implementation cost.
5. What disadvantages are there in implicit dereferencing of pointers, but only in certain contexts?
6. Explain all of the differences between Ada's subtypes and derived types.
7. What significant justification is there for the `->` operator in C and C++?
8. What are all of the differences between the enumeration types of C++ and those of Java?
9. Multidimensional arrays can be stored in row major order, as in C++, or in column major order, as in Fortran. Develop the access functions for both of these arrangements for three-dimensional arrays.
10. In the Burroughs Extended ALGOL language, matrices are stored as a single-dimensioned array of pointers to the rows of the matrix, which are treated as single-dimensioned arrays of values. What are the advantages and disadvantages of such a scheme?
11. Analyze and write a comparison of C's `malloc` and `free` functions with C++'s **`new`** and **`delete`** operators. Use safety as the primary consideration in the comparison.
12. Analyze and write a comparison of using C++ pointers and Java reference variables to refer to fixed heap-dynamic variables. Use safety and convenience as the primary considerations in the comparison.
13. Write a short discussion of what was lost and what was gained in Java's designers' decision to not include the pointers of C++.

14. What are the arguments for and against Java's implicit heap storage recovery, when compared with the explicit heap storage recovery required in C++? Consider real-time systems.
15. What are the arguments for the inclusion of enumeration types in C#, although they were not in the first few versions of Java?
16. What would you expect to be the level of use of pointers in C#? How often will they be used when it is not absolutely necessary?
17. Make two lists of applications of matrices, one for those that require jagged matrices and one for those that require rectangular matrices. Now, argue whether just jagged, just rectangular, or both should be included in a programming language.
18. Compare the string manipulation capabilities of the class libraries of C++, Java, and C#.
19. Look up the definition of *strongly typed* as given in Gehani (1983) and compare it with the definition given in this chapter. How do they differ?
20. In what way is static type checking better than dynamic type checking?
21. Explain how coercion rules can weaken the beneficial effect of strong typing.

PROGRAMMING EXERCISES

1. Design a set of simple test programs to determine the type compatibility rules of a C compiler to which you have access. Write a report of your findings.
2. Determine whether some C compiler to which you have access implements the `free` function.
3. Write a program that does matrix multiplication in some language that does subscript range checking and for which you can obtain an assembly language or machine language version from the compiler. Determine the number of instructions required for the subscript range checking and compare it with the total number of instructions for the matrix multiplication process.
4. If you have access to a compiler in which the user can specify whether subscript range checking is desired, write a program that does a large number of matrix accesses and time their execution. Run the program with subscript range checking and without it, and compare the times.
5. Write a simple program in C++ to investigate the safety of its enumeration types. Include at least 10 different operations on enumeration types to determine what incorrect or just silly things are

legal. Now, write a C# program that does the same things and run it to determine how many of the incorrect or silly things are legal. Compare your results.

6. Write a program in C++ or C# that includes two different enumeration types and has a significant number of operations using the enumeration types. Also write the same program using only integer variables. Compare the readability and predict the reliability differences between the two programs.
7. Write a C program that does a large number of references to elements of two-dimensioned arrays, using only subscripting. Write a second program that does the same operations but uses pointers and pointer arithmetic for the storage-mapping function to do the array references. Compare the time efficiency of the two programs. Which of the two programs is likely to be more reliable? Why?
8. Write a Perl program that uses a hash and a large number of operations on the hash. For example, the hash could store people's names and their ages. A random-number generator could be used to create three-character names and ages, which could be added to the hash. When a duplicate name was generated, it would cause an access to the hash but not add a new element. Rewrite the same program without using hashes. Compare the execution efficiency of the two. Compare the ease of programming and readability of the two.
9. Write a program in the language of your choice that behaves differently if the language used name equivalence than if it used structural equivalence.
10. For what types of A and B is the simple assignment statement A = B legal in C++ but not Java?