8 Composite Types

8.1.4 Unions (Variant Records, Datatypes)

EXAMPLE 8.82
Nested `structs` and `unions` in traditional C

A variant record provides two or more alternative fields or collections of fields, only one of which is valid at any given time. This notion has its roots in the `equivalence` statement of Fortran I and in the `union` types of Algol 68. Building on the `element` type of Example 8.1, one could specify a variant record as follows in (pre-2011) C:

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
    _Bool naturally_occurring;
    union {
        struct {
            char *source;
                /* textual description of principal commercial source */
            double prevalence;
                /* fraction, by weight, of Earth's crust */
        } natural_info;
        double lifetime;
            /* half-life in seconds of the most stable known isotope */
    } extra_fields;
} copper;
```

Here the programmer presumably intends for the `naturally_occurring` field to indicate which parts of the union are currently valid. A `true` value indicates that the element has at least one naturally occurring stable isotope; in this case fields `source` and `prevalence` are intended to describe how the element may be obtained and how commonly it occurs. A `false` value indicates that the element results only from atomic collisions or the decay of heavier elements; in this case, field `lifetime` is intended to indicate how long atoms so created tend to survive before undergoing radioactive decay. These mutually exclusive sets of fields (`source` and

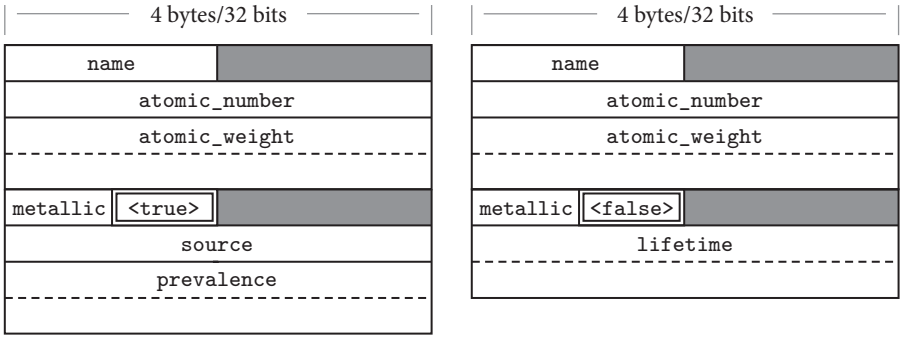

Figure 8.16 Likely memory layouts for `element` variants. The value of the `naturally_occurring` field (shown here with a double border) is intended to indicate which of the interpretations of the remaining space is valid. Field `source` is assumed to point to a string that has been independently allocated.

`prevalence`, on the one hand, or `lifetime` on the other) are sometimes known as *variants*. Either the first or the second variant may be useful, but never both at once. From an implementation perspective, nonoverlapping uses suggest that the variants may share space, as shown in Figure C-8.16.

One significant problem with our nested `struct` and `union` is the need for two extra levels of naming. While the always-present fields can be accessed as, say, `copper.atomic_weight`, fields of the inner `struct` are much less easy to name: `copper.extra_fields.natural_info.source`.

EXAMPLE 8.83
A variant record in Pascal

Pascal's principal contribution to union types was to integrate them with records. In Pascal syntax, our running example might look like this:

```
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean;
    case naturally_occurring : Boolean of
      true : (
        source : string_ptr;
        prevalence : real;
      );
      false : (
        lifetime : real;
      )
end;
```

Here the `naturally_occurring` field is introduced with the keyword `case`, to formalize its role as a *tag* or *discriminant*. Note that the variant fields have no extraneous levels of naming: we can refer directly to `copper.source`.

EXAMPLE 8.84
Anonymous unions in C11 and C++11

Leveraging an extension long supported by `gcc`, C11 and its successors allow a nameless (*anonymous*) struct or union to appear within another struct or union. The members of the anonymous construct are then directly visible in the surrounding context:

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
    _Bool naturally_occurring;
    union {
        struct {
            char *source;
            double prevalence;
        };
        double lifetime;
    };
} copper;
...
copper.source = "various ores";
```

Anonymous nesting makes variants in C11 as convenient as those of Pascal. C++11 even allows anonymous unions in non-struct contexts:

```
void foo() {
    union {
        int a;
        int b;
    };
    ...
    a = 3;
```

Safety

EXAMPLE 8.85
Breaking type safety with unions

A potentially more significant problem with unions in C is the lack of type safety. Mistakes in which the programmer writes to one field of a union and then reads from the other are relatively common:

```
union {
    int i;
    double d;
} u;
...
u.d = 3.0;
...
printf("%d", u.i);
```

Here the `printf` statement, which attempts to output `i` as an integer, will (in most implementations) take its bits from the floating-point representation of 3.0—almost certainly a mistake, but one that the language implementation will not catch. ■

EXAMPLE 8.86
Type-safe unions in OCaml

To avoid these sorts of errors, Algol 68 included features to track the status of unions at run time, and to prevent access to currently invalid fields. Similar features can be found in Ada and in ML-family languages today. Our running `element` example might be written as follows in OCaml:

```
type natural_info = {source : string; prevalence : float};;
type synthesized_info = {lifetime : float};;
type extra_info =
  | Natural of natural_info
  | Synthesized of synthesized_info;;

type element = {
    name : string;
    atomic_number : int;
    atomic_weight : float;
    metallic : bool;
    extra_fields : extra_info};;
```

As in traditional C, the variant portions of a record introduce extra levels of nesting in OCaml. To enforce correct usage, the language implementation maintains a hidden tag in every union object, to indicate which variant is currently valid. Values can be declared only as aggregates that specify the tag and all the fields:

```
let copper = {
    name = "Cu";
    atomic_number = 29;
    atomic_weight = 63.546;
    metallic = true;
    extra_fields = Natural ({
        source = "various ores and native deposits";
        prevalence = 0.00005
    })
};;
```

Individual fields can be read, but only in the context of a `match` expression that verifies the value of the tag:

```
exception Union_error;;
let source (e : element) =
    match e.extra_fields with
      | Natural     n -> n.source
      | Synthesized _ -> raise Union_error;;

let copper_source = source copper;;
```

■

Variant records with mandatory tags (explicit or hidden) are known as *discriminated unions*. Variant records without tags (as in C) are known as *nondiscriminated unions*. Pascal provided both, but in the absence of an analogue of `match`, even the discriminated case was difficult to implement safely (more on this in Exercise C-8.38). Ada, by contrast, combines syntax reminiscent of Pascal with the type safety of ML.

Variants in Ada

EXAMPLE 8.87

Ada variants and tags (discriminants)

Ada variant records must always have a tag (called the *discriminant*). Language rules ensure that this tag can never be changed without simultaneously assigning values to all of the fields of the corresponding variant. The assignment can occur either via whole-record assignment (e.g., `a := b`, where `a` and `b` are variant records), or via assignment of an aggregate (e.g., `p := (polar => true, rho => 1.0, theta => pi/2.0)`). In addition to appearing as a field within the record, the discriminant of a variant record in Ada must also appear in the header of the record's declaration:

```
type Element (Naturally_Occurring : Boolean := True) is record
    Name : String (1..2);
    Atomic_Number : Integer;
    Atomic_Weight : Long_Float;
    Metallic : Boolean;
    case Naturally_Occurring is
        when True =>
            Source : String_Ptr;
            Prevalence : Long_Float;
        when False =>
            Lifetime : Long_Float;
    end case;
end record;
```

Here we have not only declared the discriminant of the record in its header, we have also specified a default value for it. A declaration of a variable of type `Element` has the option of accepting this default value:

```
Copper : Element;
```

or overriding it:

```
Plutonium : Element (False);
Neptunium : Element (Naturally_Occurring => False);
    -- alternative syntax
```

If the type declaration for `Element` did not specify a default value for `Naturally_Occurring`, then all variables of type `Element` would have to provide a value. These rules guarantee that the tag field of a variant record is never uninitialized. ■

An Ada record variable whose declaration specifies a value for the discriminant is said to be *constrained*. Its tag field can never be changed by a subsequent assignment. This immutability means that the compiler can allocate just enough space to hold the specified variant; this space may in some cases be significantly smaller than would be required for other variants. A variable whose declaration does not provide an initial value for the discriminant is said to be *unconstrained*. Its tag will be initialized to the value in the type declaration, but may be changed by later (whole-record) assignments, so the space that the record occupies must be large enough to hold any possible variant.

EXAMPLE 8.88
A discriminated subtype in Ada

An Ada subtype definition can also constrain the discriminant(s) of its parent type:

```
subtype Natural_Element is Element (True);
```

Variables of type `Natural_Element` will all be constrained; their `Naturally_Occurring` field cannot be changed. Because `Natural_Element` is a subtype, rather than a derived type, values of type `Element` and `Natural_Element` are compatible with each other, though a run-time semantic check will usually be required to assign the former into the latter.

EXAMPLE 8.89
Discriminated array in Ada

Ada uses record discriminants not only for variant tags, but in general for any value that affects the size of a record. Here is an example that uses a discriminant to specify the length of an array:

DESIGN & IMPLEMENTATION

8.14 The placement of variant fields

To facilitate space saving in constrained variant records, Ada requires that all variant parts of a record appear at the end. This rule ensures that every field has a constant offset from the beginning of the record, with no holes (in any variant) other than those required for alignment. When a constrained variant record is elaborated, the Ada run-time system need only allocate sufficient space to hold the specified variant, which is never allowed to change. Pascal had a similar rule, designed for a similar purpose. When a variant record was allocated from the heap in Pascal (via the built-in `new` operator), the programmer had the option of specifying `case` labels for the variant portions of the record. A record so allocated was never allowed to change to a different variant, so the implementation could allocate precisely the right amount of space.

Modula-2, which did not provide `new` as a built-in operation, eliminated the ordering restriction on variants. All variables of a variant record type had to be large enough to hold any variant. The usual implementation assigned a fixed offset to every field, with holes following small internal variants as necessary. Similar conventions apply to `unions` and `structs` in modern C.

```
type Element_Array is array (Integer range <>) of Element;
type Alloy (Num_Components : Integer) is record
    Name : String (1..30);
    Components : Element_Array (1..Num_Components);
    Tensile_Strength : Long_Float;
end record;
```

The <> notation in the initial definition of `Element_Array` indicates that the bounds are not statically known. Further discussion of dynamic arrays appears in Section 8.2.2. As with discriminants used for variant tags, the programmer must either specify a default value for the discriminant in the type declaration (we did not do so above), or else every declaration of a variable of the type must specify a value for the discriminant (in which case the variable is constrained, and the discriminant cannot be changed).

The Object-Oriented Alternative

EXAMPLE 8.90
Derived types as an alternative to unions

In dropping variant records from their parent language, the designers of Modula-3 noted [Har92, p. 110] that much of the same effect could be obtained with classes and inheritance. Oberon, similarly, replaced variants with a more general mechanism for *type extension* (sidebar 10.3), and the designers of Java and C# dropped the `unions` of C and C++. In place of the C code of Example C-8.82, a Java programmer might write

```
class Element {
    public String name;
    public int atomicNumber;
    public double atomicWeight;
    public boolean metallic;
}
class NaturalElement extends Element {
    public String source;
    public double prevalence;
}
class SyntheticElement extends Element {
    public double lifetime;
}
```

Like the discriminated subtypes of Ada, this approach constrains each object to a single variant at creation time, but this may not be a problem: while the class of a particular object never changes, class-type variables are references in Java and C#. A variable of type `Element` can easily refer to an object of class `NaturalElement` or `SyntheticElement` at run time.

CHECK YOUR UNDERSTANDING

55. What are *anonymous* unions and structs? What purpose do they serve? How is this related to the integration of variants with records in Pascal and its descendants?

56. What is a *tag* (*discriminant*) in a variant record? In a language like Ada or OCaml, how does it differ from an ordinary field?

57. Discuss the type safety problems that arise with variant records. How can these problems be addressed?

58. Summarize the rules that prevent access to inappropriate fields of variant records in OCaml and Ada.

59. Why might one wish to *constrain* a variable, so that it can hold only one variant of a type?

60. Explain how classes and inheritance can be used to obtain the effect of constrained variant records.

8 Composite Types

8.5.3 Dangling References

Memory access errors—dangling references, memory leaks, out-of-bounds access to arrays—are among the most common program bugs, and among the most difficult to find. Testing and debugging techniques for memory errors vary in when they are performed, how much they cost, and how conservative they are. Several commercial and open-source tools employ binary instrumentation (Section 16.2.3) to track the allocation status of every block in memory and to check every load or store to make sure it refers to an allocated block. These tools have proved to be highly effective, but they can slow a program several-fold, and may generate *false positives*—indications of error in programs that, while arguably poorly written, are technically correct. Many compilers can also be instructed to generate dynamic semantic checks for certain kinds of memory errors. Such checks must generally be fast (much less than 2× slowdown), and must never generate false positives. In this section we consider two candidate implementations of checks for dangling references.

Tombstones

EXAMPLE 8.91
Dangling reference detection with tombstones

Tombstones [Lom75, Lom85] allow a language implementation to catch all dangling references, to objects in both the stack and the heap. The idea is simple: rather than have a pointer refer to an object directly, we introduce an extra level of indirection (Figure C-8.17). When an object is allocated in the heap (or when a pointer is created to an object in the stack), the language run-time system allocates a tombstone. The pointer contains the address of the tombstone; the tombstone contains the address of the object. When the object is reclaimed, the tombstone is modified to contain a value (typically zero) that cannot be a valid address. To avoid special cases in the generated code, tombstones are also created for pointers to static objects. ■

For heap objects, it is easy to invalidate a tombstone when the program calls the deallocation operation. For stack objects, the language implementation must be able to find all tombstones associated with objects in the current stack frame

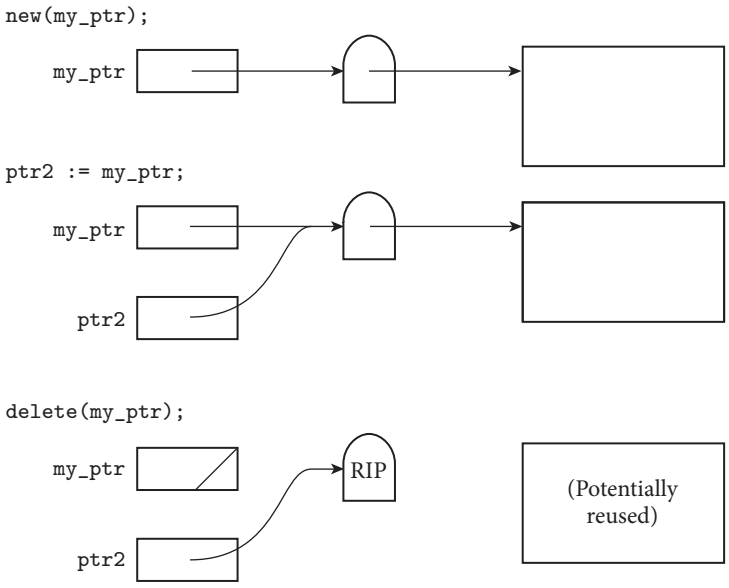

Figure 8.17 **Tombstones.** A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an "expired" tombstone.

when returning from a subroutine. One possible solution is to link all stack-object tombstones together in a list, sorted by the address of the stack frame in which the object lies. When a pointer is created to a local object, the tombstone can simply be added to the beginning of the list. When a pointer is created to a parameter, the run-time system must scan down the list and insert in the middle, to keep it sorted. When a subroutine returns, the epilogue portion of the calling sequence invalidates the tombstones at the head of the list, and removes them from the list.

Tombstones may be allocated from the heap itself or, more commonly, from a separate pool. The latter option avoids fragmentation problems, and makes allocation relatively fast, since the first tombstone on the free list is always the right size.

Tombstones can be expensive, both in time and in space. The time overhead includes (1) creation of tombstones when allocating heap objects or using a "pointer to" operator, (2) checking for validity on every access, and (3) double-indirection. Fortunately, checking for validity can be made essentially free on most machines by arranging for the address in an "invalid" tombstone to lie outside the program's address space. Any attempt to use such an address will result in a hardware interrupt, which the operating system can reflect up into the language run-time system. We can also use our invalid address, in the pointer itself, to represent the constant `nil`. If the compiler arranges to set every pointer to `nil` at elaboration time, then the hardware will catch any use of an uninitialized pointer. (This technique works without tombstones, as well.)

The space overhead for tombstones can be significant. The simplest approach is never to reclaim them. Since a tombstone is usually significantly smaller than the object to which it refers, a program will waste less space by leaving a tombstone around forever than it would waste by never reclaiming the associated object. Even so, any long-running program that continually creates and reclaims objects will eventually run out of space for tombstones. A potential solution, which we consider in Section 8.5.4, is to augment every tombstone with a *reference count*, and reclaim tombstones themselves when the reference count goes to zero.

Tombstones have a valuable side effect. Because of double-indirection, it is easy to change the location of an object in the heap. The run-time system need not locate every pointer that refers to the object; all that is required is to change the address in the tombstone. The principal reason to change heap locations is for *storage compaction*, in which all dynamically allocated blocks are "scooted together" at one end of the heap in order to eliminate external fragmentation. Tombstones are not widely used in language implementations, but the Macintosh operating system (versions 9 and below) used them internally, for references to system objects such as file and window descriptors. They also closely resemble the implementation used for *smart pointers* in the C++ standard library.

Locks and Keys

Locks and *keys* [FL80] are an alternative to tombstones. Their disadvantage is that they provide only probabilistic protection from dangling pointers. Their advantage is that they avoid the need to keep tombstones around forever (or to figure out when to reclaim them). Again the idea is simple: Every pointer is a tuple consisting of an address and a key. Every object in the heap begins with a lock. A pointer to an object in the heap is valid only if the key in the pointer matches the lock in the object (Figure C-8.18). When the run-time system allocates a new heap object, it generates a new key value. These can be as simple as serial numbers, but should avoid "common" values such as zero and one. When an object is reclaimed, its lock is changed to some arbitrary value (e.g., zero) so that the keys in any remaining pointers will not match. If the block is subsequently reused for another purpose, we expect it to be very unlikely that the location that used to contain the lock will be restored to its former value by coincidence. The original implementation of locks and keys in Pascal considered only pointers to heap objects, as outlined above, but in principle a run-time system could also add locks to objects allocated on the stack.

EXAMPLE 8.92

Dangling reference detection with locks and keys

Like tombstones, locks and keys incur significant overhead. They add an extra word of storage to every pointer and to every block in the heap. They increase the cost of copying one pointer into another. Most significantly, they incur the cost of comparing locks and keys on every access (or every access that cannot be proven to be redundant). It is unclear whether the lock and key check is cheaper or more expensive than the tombstone check. A tombstone check may result in two cache misses (one for the tombstone and one for the object); a lock and key check is unlikely to cause more than one. On the other hand, the lock and key check requires a significantly longer instruction sequence on most machines. To

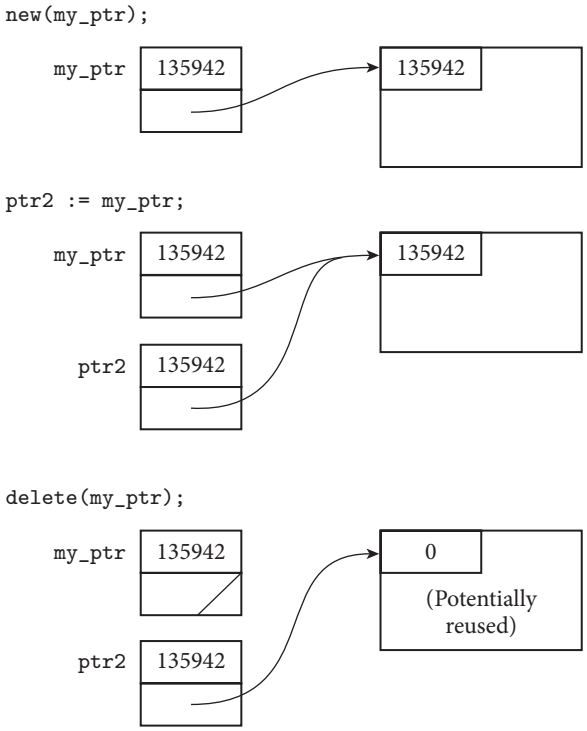

Figure 8.18 **Locks and keys.** A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.

minimize time and space overhead, most compilers do not by default generate code to check for dangling references.

CHECK YOUR UNDERSTANDING

61. What are *tombstones*? What changes do they require in the code to allocate and deallocate memory, and to assign and dereference pointers?
62. Explain how tombstones can be used to support *compaction*.
63. What are *locks* and *keys*? What changes do they require in the code to allocate and deallocate memory, and to assign and dereference pointers?
64. Explain why the protection afforded by locks and keys is only probabilistic.
65. Discuss the comparative advantages of tombstones and locks and keys as a means of catching dangling references.

8 Composite Types

8.7 Files and Input/Output

The first two subsections below are devoted to interactive and file-based I/O, respectively. Section C-8.7.3 then considers the common special case of text files.

8.7.1 Interactive I/O

On a modern machine, interactive I/O usually occurs through a graphical user interface (GUI: "gooey") system, with a mouse, a keyboard, and a bit-mapped screen that in turn support windows, menus, scrollbars, buttons, sliders, and so on. GUI characteristics vary significantly among, say, Microsoft Windows, the Macintosh, and Unix's X11; the differences are one of the principal reasons it is difficult to port applications across platforms.

Within a single platform, the facilities of a GUI system usually take the form of library routines (to create or resize a window, print text, draw a polygon, and so on). Input events (mouse move, button push, keystroke) may be placed in a queue that is accessible to the program, or tied to *event handler* subroutines that are called by the run-time system when the event occurs. Because the handler is triggered from outside, its activities must generally be *synchronized* with those of the main program, to make sure that both parties see a consistent view of any data shared between them. We will discuss events further in Section 9.6, and synchronization in Section 13.3.

A few programming languages—notably Smalltalk—attempt to incorporate a standard set of GUI mechanisms into the language itself. The Smalltalk design team was part of the original group at Xerox's Palo Alto Research Center (PARC) that invented mouse-and-window based interfaces in the early 1970s. Unfortunately, while the Smalltalk GUI is successful within the confines of the language, it tends not to integrate well with the "look and feel" of the host system on which it runs.

Other languages—Java, for example—provide graphics as a standard library package. Java's original GUI facilities (the Abstract Window Toolkit—AWT) had something of a "least common denominator" look to them. The Java routines and their interface have evolved significantly over time; the more recent Swing and JavaFX libraries have "pluggable" look and feel, allowing them to integrate more easily with (and port more easily among) a variety of window systems.

The "parallel execution" of the program and the human user that characterizes interactive systems is difficult to capture in a functional programming model. A functional program that operates in a "batch" mode (taking its input from a file and writing its output to a file) can be modeled as a function from input to output. A program that interacts with the user, however, requires a very concrete notion of program ordering, because later input may depend on earlier output. If both input and output take the form of an ordered sequence of tokens, then interactive I/O can be modeled using lazy data structures, a subject we considered in Section 6.6.2. More general solutions can be based on the notion of *monads*, which use a functional notion of sequencing to model side effects. We will consider these issues again in Sections 11.5 and 11.8.

8.7.2 File-Based I/O

Persistent files are the principal mechanism by which programs that run at different times communicate with each other. A few language proposals (e.g., Argus [LS83] and χ [SH92]) allow ordinary variables to persist from one invocation of a program to the next, and a few experimental operating systems (e.g., Opal [CLFL94] and Hemlock [GSB+93]) provide persistence for variables outside the language proper. In addition, some language-specific programming environments, such as those for Smalltalk and Common Lisp, provide a notion of *workspace* that includes persistent named variables. With coming advances in nonvolatile memory technology, such features may find their way into a larger number of languages. Historically, they have been more the exception than the rule. For the most part, data that need to outlive a particular program invocation have needed to reside in files.

Like interactive I/O, files can be incorporated directly into the language, or provided via library routines. In the latter case, it is still a good idea for the language designers to suggest a standard library interface, to promote portability of programs across platforms. The lack of such a standard in Algol 60 is widely credited with impeding the language's widespread use. One of the principal reasons to incorporate I/O into the language proper is to make use of special syntax. In particular, several languages, notably Fortran and Pascal, provide built-in I/O facilities in order to obtain type-safe "subroutines" that take a variable number of parameters, some of which may be optional.

Depending on the needs of the programmer and the capabilities of the host operating system, data in files may be represented in binary form, much as it is in memory, or as *text*. In a binary file, the number 1066_{10} would be represented by the 32-bit value 10000101010_2. In a text file, it would probably be represented

by the character string "1066". Temporary files are usually kept in binary form for the sake of speed and convenience. Persistent files are commonly kept in both forms. Text files are more easily ported across systems: issues of word size, byte order, alignment, floating-point format, and so on do not arise. Text files also have the advantage of human readability: they can be manipulated by text editors and related tools. Unfortunately, text files tend to be large, particularly when used to hold numeric data. A double-precision floating-point number occupies only eight bytes in binary form, but can require as many as 24 characters in decimal notation (not counting any surrounding white space). Text files also incur the cost of binary to text conversion on output, and text to binary conversion on input. The size problem can be addressed, at least for archival storage, by using data compression. Mechanisms to control text/binary conversion tend to be the most complicated part of I/O; we discuss them in the following subsection.

EXAMPLE 8.93
Files as a built-in type

When I/O is built into a language, files are usually declared using a built-in type constructor, as for example in Pascal:

```
var my_file : file of foo;
```

EXAMPLE 8.94
The open operation

If I/O is provided by library routines, the library usually provides an opaque type to represent a file. In either case, each file variable is generally bound to an external, operating system–supported file by means of an *open* operation. In C, for example, one says

```
my_file = fopen(path_name, mode);
```

The first argument to fopen is a character string that names the file, using the naming conventions of the host operating system. The second argument is a string that indicates whether the file should be readable, writable, or both, whether it should be created if it does not yet exist, and whether it should be overwritten or appended to if it does exist.

EXAMPLE 8.95
The close operation

When a program is done with a file, it can break the association between the file variable and the external object by using a *close* operation:

```
fclose(my_file);
```

In response to a call to close, the operating system may perform certain "finalizing" operations, such as unlocking an exclusive file (so that it may be used by other programs), rewinding a tape drive, or forcing the contents of buffers out to disk.

Most files, both binary and text, are stored as a linear sequence of characters, words, or records. Every open file then has a notion of *current position*: an implicit reference to some element of the sequence. Each read or write operation implicitly advances this reference by one position, so that successive operations access successive elements, automatically. In a *sequential* file, this automatic advance is the only way to change the current position. Sequential files usually correspond to media like printers and tapes, in which the current position has a physical representation (how many pages we've printed; how much tape is on each spool) that is difficult to change.

In other, *random-access* files, the programmer can change the current position to an arbitrary value by issuing a *seek* operation. In a few programming languages (e.g., Cobol and PL/I), random-access files (also called *direct* files) have no notion of current position. Rather, they are *indexed* on some key, and every `read` or `write` operation must specify a key. A file that can be accessed both sequentially *and* by key is said to be *indexed sequential*.

Random-access files usually correspond to media like solid-state flash drives or magnetic or optical disks, in which the current position can be changed with relative ease. Where tape drives (still widely used for archival storage) can take more than a minute to seek to a given position, modern disks take anywhere from 5 to 200 ms, depending on technology. (Note that 5 ms is still a very long time—10 million cycles on a 2 GHz processor.) Seeking on a solid-state device is essentially instantaneous. A few languages—notably Pascal—provide no random-access files, though individual implementations may support random access as a nonstandard language extension.

8.7.3 Text I/O

It is conventional to think of text files as consisting of a sequence of *lines*, each of which in turn consists of characters. In older systems, particularly those designed around the metaphor of punch cards, lines are reflected in the organization of the file itself. A `seek` operation, for example, may take a line number as argument. More commonly, a text file is simply a sequence of characters. Within this sequence, control (nonprinting) characters indicate the boundaries between lines. Unfortunately, end-of-line conventions are not standardized. In Unix and in modern versions of the Mac OS, each line of a text file ends with a *newline* ("control-J") character, ASCII value 10. (On "classic" Macs, each line ended with a *carriage return* ("control-M") character, ASCII value 13.) On Windows machines, each line ends with a carriage return/newline pair. Text files are usually sequential.

Despite the muddied conventions for line breaks, text files are much more portable and readable than binary files.[1] Because they do not mirror the structure of internal data, text files require extensive conversions on input and output. Issues to be considered include the base for integer values (and the representation of nondecimal bases); the representation of floating-point values (number of digits, placement of decimal point, notation for exponent); the representation of enumerations and other nonnumeric, nonstring types; and positioning, if any, within columns (right and left justification, zero or white-space fill, "floating" dollar signs in Cobol). Some of these issues (e.g., the number of digits in a floating-point

1 We are speaking here, of course, of plain-text ASCII or Unicode files. So-called "rich text" files, consisting of formatted text in particular fonts, sizes, and colors, perhaps with embedded graphics, are another matter entirely. Word processors typically represent rich text with a combination of binary and ASCII data, though ASCII-only standards such as Postscript, textual PDF, RTF, and XML can be used to enhance portability.

number) are influenced by the hardware, but most are dictated by the needs of the application and the preferences of the programmer.

In most languages the programmer can take complete control of input and output formatting by writing it all explicitly, using language or library mechanisms to read and write individual characters only. I/O at such a low level is tedious, however, and most languages also provide more high-level operations. These operations vary significantly in syntax and in the degree to which they allow the programmer to specify I/O formats. We illustrate the breadth of possibilities with examples from four imperative languages: Fortran, Ada, C, and C++.

Text I/O in Fortran

EXAMPLE 8.96
Formatted output in Fortran

In Fortran, we could write a character string, an integer, and an array of 10 floating-point numbers as follows:

```
character s*20
integer n
real r (10)
...
write (4, '(A20, I10, 10F8.2)'), s, n, r
```

In the `write` statement, the 4 indicates a *unit number*, which identifies a particular output file. The quoted, parenthesized expression is called a *format*; it specifies how the printed variables are to be represented. In this case, we have requested a 20-column ASCII string, a 10-column integer, and 10 eight-column floating-point numbers (with two columns of each reserved for the fractional part of the value). Fortran provides an extremely rich set of these *edit descriptors* for use inside of formats. Cobol, PL/I, and Perl provide comparable facilities, though with a very different syntax.

EXAMPLE 8.97
Labeled formats

Fortran allows a format to be specified indirectly, so it may be used in more than one input or output statement:

```
write (4, 100), s, n, r             ! 100 is the line number
...                                 ! of the format statement
100 format (A20, I10, 10F8.2)
```

It also allows formats to be created at run time, and stored in strings:

```
character(len=20) :: fmt
...
fmt = "(A20, I10, 10F8.2)"
...
write (4, fmt), s, n, r
```

If the programmer does not know, or does not care about, the precise allocation of columns to fields, the format can be omitted:

```
write (4, *), s, n, r
```

In this case, the run-time system will use default format conventions.

EXAMPLE 8.98
Printing to standard output

To write to the standard output stream (i.e., the terminal or its surrogate), the programmer can use the `print` statement, which resembles a `write` without a unit number:

```
print*, s, n, r          ! * means default format
```

For input, `read` is used both for standard input and for specific files; in the former case, the unit number is omitted, together with the extra set of parentheses:

```
read 100, s, n, r
...
read*, s, n, r           ! * means default format
```

The star may be omitted in Fortran 90.

In the full form of `read`, `write`, and `print`, additional arguments may be provided in the parenthesized list with the unit number and format. These can be used to specify a variety of additional information, including a label to which to jump on end-of-file, a label to which to jump on other errors, a variable into which to place status codes returned by the operating system, a set of labels (a "namelist") to attach to the output values, and a control code to override the usual automatic advance to the next line of the file. Because there are so many of these optional arguments, most of which are usually omitted, they are usually specified using *named* (keyword) parameter notation, a notion we defer to Section 9.3.3.

The variety of shorthand versions of `read`, `write`, and `print`, together with the fact that they operate on a variable number of program variables, makes it very difficult to cast them as "ordinary" subroutines. Fortran 90 provides optional and named parameters, but Fortran 77 does not, and even in Fortran 90 there is no way to define a subroutine with an *arbitrary* number of parameters.

Text I/O in Ada

Ada provides a suite of five standard library packages for I/O. The `Sequential_IO` and `Direct_IO` packages are for binary files. They provide generic file types that can be instantiated for any desired element type. The `IO_Exceptions` and `Low_Level_IO` packages handle error conditions and device control, respectively. The `Text_IO` package provides formatted input and output on sequential files of characters.

EXAMPLE 8.99
Formatted output in Ada

Using `Text_IO`, our original three-variable Fortran output statement would look something like this in Ada:

```
s : array (1..20) of Character;
n : Integer;
r : array (1..10) of Real;
...
```

```
set_output(My_File);
Put(N, 10);
Put(S);
for I in 1..10 loop Put(R(I), 5, 2); end loop;
New_Line;
```

In the `Put` of an element of `R` (within the loop), the second parameter specifies the number of digits before the decimal point, rather than the width of the entire number (including the decimal point), as it did in Fortran. The `Put` of `S` will use the string's natural length. If a different length is desired, the programmer will have to write blanks or `Put` a substring explicitly. If precise output positioning is not desired for the integers and real numbers, the extra parameters in their `Put` calls can be omitted; in this case the run-time system will use standard defaults. The programmer can use additional library routines to change these defaults if desired. A call to `Set_Output` invokes a similar mechanism: it changes the default notion of output file.

EXAMPLE 8.100
Overloaded `Put` routines

There are two overloaded forms of `Put` for every built-in type. One takes a file name as its first argument; the other does not. The last five lines above could have been written

```
Put(My_File, N, 10);
Put(My_File, S);
for I in 1..10 loop Put(My_File, R(I), 5, 2); end loop;
New_Line(My_File);
```

The programmer can of course define additional forms of `Get` and `Put` for arbitrary user-defined types. All of these facilities rely on standard Ada mechanisms; in contrast to Fortran, no support for I/O is built into the language itself.

Text I/O in C

C provides I/O through a library package called `stdio`; as in Ada, no support for I/O is built into the language itself. Many C implementations, however, build knowledge of I/O functions into the compiler, so it can issue warnings when arguments appear to be used incorrectly.

EXAMPLE 8.101
Formatted output in C

Our example output statement would look something like this in C:

```
char s[20];
int n;
double r[10];
...
fprintf(my_file, "%20s%10d", s, n);
for (i = 0; i < 10; i++) fprintf(my_file, "%8.2f", r[i]);
fprintf(my_file, "\n");
```

The arguments to `fprintf` are a file, a format string, and a sequence of expressions. The format string has capabilities similar to the formats of Fortran, though

the syntax is very different. In general, a format string consists of a sequence of characters with embedded "placeholders," each of which begins with a percent sign. The placeholder `%20s` indicates a 20-character string; `%d` indicates an integer in decimal notation; `%8.2f` indicates an 8-character floating-point number, with two digits to the right of the decimal point.

EXAMPLE 8.102
Text in format strings

As in Fortran, formats can be computed and stored in strings, and a single `fprintf` statement can print an arbitrary number of expressions. As in Ada, an explicit `for` loop is needed to print an array. Commonly the format string also contains labeling text and white space:

```
strcpy(s, "four");                      /* copy "four" into s */
n = 20;
char *fmt = "%s score and %d years ago\n";
fprintf(my_file, fmt, s, n);
```

A percent sign can be printed by doubling it:

```
fprintf(my_file, "%d%%\n", 25);     /* prints "25%" */
```

EXAMPLE 8.103
Formatted input in C

Input in C takes a similar form. The `fscanf` routine takes as argument a file, a format string, and a sequence of pointers to variables. In the common case, every argument after the format is a variable name preceded by a "pointer to" operator:

```
fscanf(my_file, "%s %d %lf", s, &n, &r[0]);
```

In this call, the `%s` placeholder will match a string of maximal length that does not include white space. If this string is longer than 20 characters (the length of `s`), then `fscanf` will write beyond the end of the storage for the string. (This weakness in `scanf` is one of the sources of the so-called "buffer overflow" bugs discussed in Sidebar 8.7. It can be avoided in this example by replacing the `%s` specifier with `%19s`, which will cause `fscanf` to move at most 19 bytes, plus a terminating `NUL`.) The three-character `%lf` placeholder informs the library routine that the corresponding argument is a `double`; the 2-character sequence `%f` would read into a `float`.[2] Accidentally using a placeholder for the wrong size variable is a common error in older implementations of C; forgetting the ampersand on a trailing argument is another. While such mistakes will often be caught by a modern C compiler with special-case knowledge of `fscanf`, they would always be caught in a language with type-safe I/O. Note that we have read a single element of `r`; as with `fprintf`, a `for` loop would be needed to read the whole array. Note also that while

2 C's `doubles` are double-precision IEEE floating-point numbers in most implementations; `floats` are usually single precision. The lack of safety for `%s` arguments is only one of several problems with `fscanf`. Others include the inability to "skip over" erroneous input, and undefined behavior when there is insufficient input. Instead of `fscanf`, seasoned C programmers tend to use `fgets`, which reads (length-limited) input into a string, followed by manual parsing using `strtol` (string-to-`long`), `strtod` (string-to-`double`), and so on.

we have not made use of this fact in our example, `fscanf` returns an integer value indicating the number of `&`-identified items that were scanned successfully.

We have noted above that the I/O routines of Fortran and Pascal are built into the language largely to permit them to take a variable number of arguments. We have also noted that moving I/O into a library in Ada forces us to make a separate call to `put` for every output expression. So how do `fprintf` and `fscanf` work? It turns out that C permits functions with a variable number of parameters (we will discuss such functions in more detail in Section 9.3.3). Unfortunately, the types of trailing parameters are unspecified, which makes compile-time type checking of variable-length argument lists impossible in the general case. Moreover, the lack of run-time type descriptors in C precludes run-time checking as well. At the same time, because the C library (including `fprintf` and `fscanf`) is part of the language standard, special knowledge of these routines can be built into the compiler—and often is: while the I/O routines of C are formally defined as "ordinary" functions, they are typically implemented in the same way as their analogues in Fortran and Pascal. As a result, C compilers will often provide good error diagnostics when the arguments to `fprintf` or `fscanf` do not match the format string.

To simplify I/O to and from the standard input and output streams, `stdio` provides routines called `printf` and `scanf` that omit the initial arguments of `fprintf` and `fscanf`. To facilitate the formatting of strings *within* a program, `stdio` also provides routines called `sprintf` and `sscanf`, which replace the initial arguments of `fprintf` and `fscanf` with a pointer to an array of characters. The `sscanf` function "reads" from this array; `sprintf` "writes" to it. Fortran 90 provides similar support for intraprogram formatting through so-called *internal files*.

Text I/O in C++

As a descendant of C, C++ supports the `stdio` library described in the previous subsection. It also supports an I/O library called `iostream` that exploits the object-oriented features of the language. The `iostream` library is more flexible than `stdio`, provides arguably more elegant syntax (though this is a matter of taste), and is completely type safe.

C++ *streams* use operator overloading to co-opt the << and >> symbols normally used for bit-wise shifts. The `iostream` library provides an overloaded version of << and >> for each built-in type, and programmers can define versions for new types. To print a C-style character string in C++, one writes

EXAMPLE 8.104
Formatted output in C++

```
my_stream << s;
```

To output a string and an integer one can write

```
my_stream << s << n;
```

This idiom requires that `my_stream` be an instance of the `ostream` (output stream) class defined in the `iostream` library. The << operator, with a right operand of type

`T`, is then syntactic sugar for either the "operator function" `operator<<(ostream, T)` or the "operator method" `ostream::operator<<(T)`, as described in Section 3.5.2. As it turns out, `iostream` provides an operator function for C-style strings and a member function for integers. Because `<<` associates left-to-right, the statement above is equivalent to

```
(operator<<(my_stream, s)).operator<<(n);
```

The code works because `operator<<` returns a reference to its first argument as its result (as we shall see in Section 9.3.1, C++ supports both a value model and a reference model for variables).

EXAMPLE 8.105
Stream manipulators

As shown so far, output to an `ostream` uses default formatting conventions. To change conventions, one may embed so-called *stream manipulators* in a sequence of `<<` operations. To print `n` in octal notation (rather than the default decimal), we could write

```
my_stream << oct << n;
```

To control the number of columns occupied by `s` and `n`, we could write

```
my_stream << setw(20) << s << setw(10) << n;
```

The `oct` manipulator causes the stream to print all subsequent numeric output in octal. The `setw` manipulator causes it to print its next string or numeric output in a field of a specified minimum width (behavior reverts to the default after a single output).

The `oct` manipulator is declared as a function that takes an `ostream` as a parameter and produces a reference to an `ostream` as its result. Because it is not followed by empty parentheses, the occurrence of `oct` in the output sequence above is *not* a call to `oct`; rather, a reference to `oct` is passed to an overloaded version of `<<` that expects a manipulator function as its right-hand argument. This version of `<<` then calls the function, passing the stream (the left-hand argument of `<<`) as argument.

The `setw` manipulator is even trickier. It is declared as a function that returns a reference to what we might call an "object closure"—an object containing a reference to a function and a set of arguments. In this particular case, `setw(20)` is a call to a *constructor* function that returns a closure containing the number 20 and a pointer to the `setw` manipulator. (We will discuss constructors in detail in Section 10.3, and object closures in Section 3.6.3.) The `iostream` library provides an overloaded version of `<<` that expects an object closure as its right-hand argument. This version of `<<` calls the function inside the closure, passing it as arguments the stream (the left-hand argument of `<<`) and the integer inside the closure.

EXAMPLE 8.106
Array output in C++

The `iostream` library provides a wealth of manipulators to change the formatting behavior of an `ostream`. Because C++ inherits C's handling of pointers and arrays, however, there is no way for an `ostream` to know the length of an array. As a result, our full output example still requires a `for` loop to print the `r` array:

```
char s[20];
int n;
double r[10];
...
my_stream << setw(20) << s << setw(10) << n;
for (i = 0; i < 10; i++)
    my_stream << setiosflags(ios::fixed)
        << setw(8) << setprecision(2) << r[i];
my_stream << "\n";
```

Here the manipulators in the output sequence in the `for` loop specify fixed format (rather than scientific) for floating-point numbers, with a field width of eight, and two digits past the decimal point. The `setiosflags` and `setprecision` manipulators change the default format of the stream; the changes apply to all subsequent output.

EXAMPLE 8.107
Changing default format

To avoid calling stream manipulators repeatedly, we could modify our example as follows:

```
my_stream.flags(my_stream.flags() | ios::fixed);
my_stream.precision(2);
for (i = 0; i < 10; i++) my_stream << setw(8) << r[i];
```

To facilitate the restoration of defaults, the `flags` and `precision` functions return the previous value:

```
ios::fmtflags old_flags = my_stream.flags(my_stream.flags() | ios::fixed);
int old_precision = my_stream.precision(2);
for (i = 0; i < 10; i++) my_stream << setw(8) << r[i];
my_stream.flags(old_flags);
my_stream.precision(old_precision);
```

Formatted input in C++ is analogous to formatted output. It uses `istreams` instead of `ostreams`, and the >> operator instead of <<. It also supports a suite of manipulators comparable to those for output. I/O on the standard input and output streams does not require different functions; the programmer simply begins an input or output sequence with the standard stream name `cin` or `cout`. (In keeping with C tradition, there is also a standard stream `cerr` for error messages.) To support intraprogram formatting of character strings, the `strstream` library provides `istrstream` and `ostrstream` object classes that are derived from `istream` and `ostream`, and that allow a stream variable to be bound to a string instead of to a file.

CHECK YOUR UNDERSTANDING

66. Explain the differences between interactive and file-based I/O, between temporary and persistent files, and between binary and text files. (Some of this information is in the main text.)
67. What are the comparative advantages of *text* and *binary* files?
68. Describe the end-of-line conventions of Unix, Windows, and Macintosh files.
69. What are the advantages and disadvantages of building I/O into a programming language, as opposed to providing it through library routines?
70. Summarize the different approaches to text I/O adopted by Fortran, Ada, C, and C++.
71. Describe some of the weaknesses of C's `scanf` mechanism.
72. What are *stream manipulators*? How are they used in C++?

8 Composite Types

8.9 Exercises

8.35 In Example 6.70 we described a programming idiom in which an iterator takes a "loop body" function as argument, and applies it to every element of a given container or set. Show how to use this idiom in ML to apply a function to every element of the tree in Example 11.39. Write versions of your iterator for preorder, inorder, and postorder traversals.

8.36 Show how unions can be used in C to interpret the bits of a value of one type as if they represented a value of some other type. Explain why the same technique does not work in Ada. After consulting an Ada manual, describe how an `unchecked` pragma can be used to get around the Ada rules.

8.37 Are variant records a form of polymorphism? Why or why not?

8.38 Learn the details of variant records in Pascal.

(a) You may have noticed that the language does not allow you to pass the tag field of a variant record to a subroutine by reference. Why not?

(b) Explain how you might implement dynamic semantic checks to catch references to uninitialized fields of a tagged variant record. Changing the value of the tag field should cause all fields of the variant part of the record to become uninitialized. Suppose you want to avoid adding flag fields within the record itself (e.g., to avoid changing the offsets of fields in a systems program). How much harder is your task?

(c) Explain how you might implement dynamic semantic checks to catch references to uninitialized fields of an *untagged* variant record. Any assignment to a field of a variant should cause all fields of other variants to become uninitialized. Any assignment that changes the record from one variant to another should also cause all other fields of the new variant to be uninitialized. Again, suppose you want to avoid adding flag fields within the untagged record itself. How much harder is your task?

8.39 We noted in Section C-8.1.4 that Ada requires the variant portions of a record to occur at the end, to save space when a particular record is constrained to have a comparatively small variant part. Could a compiler rearrange fields to achieve the same effect, without the restriction on the declaration order of fields? Why or why not?

8.40 Reference counts can be used to reclaim tombstones,. While it is certainly possible to create a circular structure with tombstones, the fact that the programmer is responsible for explicit deallocation of heap objects implies that reference counts will fail to reclaim tombstones only when the programmer has failed to reclaim the objects to which they refer. Explain how to leverage this observation to catch memory leaks at run time. Does your solution work in all cases? Explain.

8.41 For objects allocated in the heap, we have suggested that a "lock" for dangling reference detection be allocated in the object header, at a known offset from the beginning of the object itself. This choice doesn't work well for pointers to static or stack-allocated objects, or in general for pointers created with an "address of" (`&`) operator, since these may refer to fields in the middle of larger objects.

Zhou [ZCH23] has suggested solving this problem by adding an extra field to every pointer, to indicate the offset between the lock and the pointed-to object. A single lock can then protect an entire stack frame, or all the static objects in a module.

Elaborate on this suggestion. Specifically, describe the code that must be executed in subroutine prologues and epilogues—and on pointer assignment and dereference—in order to detect dangling references in a language that permits pointers to non-heap objects.

8.42 In Section 8.5.4 we introduced the notion of *smart pointers* in C++. Learn how these are implemented, and write an explanation. Discuss the relationship to tombstones.

8.43 Rewrite Example C-8.103 using `fgets`, `strtol`, `strtod`, etc. (read the `man` pages), so that it is guaranteed not to result in buffer overflow.

8.44 The output routines of several languages (e.g., `println` in Swift) give special treatment to ends of lines. By contrast, C's `printf` does not; it treats newlines and carriage returns the same as any other character. What are the comparative advantages of these approaches? Which do you prefer? Why?

8 Composite Types

8.10 Explorations

8.54 Research the history of smart pointers (Section 8.5.4) in C++, including the `unique_ptr`, `shared_ptr`, and `weak_ptr` of C++11; the `auto_ptr` of C++98, and the various pointer classes of the popular Boost library. How has the standard set of pointers evolved over time? What accounts for the changes? Do you consider the current mechanisms an adequate replacement for automatic garbage collection? Why or why not?

8.55 Find a Cobol manual and learn about the language's facilities for text I/O. Prepare a written comparison of those facilities to those of the languages described in Section C-8.7.3.

8.56 If you were designing the text I/O facilities for a new programming language, what approach would you take? In particular, do you believe that I/O should be a built-in part of the language, or should it be handled by library routines?