

7

Data Types

7.3.4 Variant Records (Unions)

EXAMPLE 7.122

Variant record in Pascal

A variant record provides two or more alternative fields or collections of fields, only one of which is valid at any given time. Building on the `element` type of Example 7.36, we might write the following in Pascal.

```
type long_string = packed array [1..200] of char;
type string_ptr = ^long_string;
type element = record
  name : two_chars;
  atomic_number : integer;
  atomic_weight : real;
  metallic : Boolean;
  case naturally_occurring : Boolean of
    true : (
      source : string_ptr;
      (* textual description of principal commercial source *)
      prevalence : real;
      (* fraction, by weight, of Earth's crust *)
    );
    false : (
      lifetime : real;
      (* half-life in seconds of most stable known isotope *)
    )
  )
end;
```

Here the `naturally_occurring` field of the record is known as its *tag*, or *discriminant*. It's a field that indicates which variant is valid. In this example, a `true` tag indicates that the element has at least one naturally occurring stable isotope; in this case the record contains two additional fields—`source` and `prevalence`—that describe how the element may be obtained and how commonly it occurs.

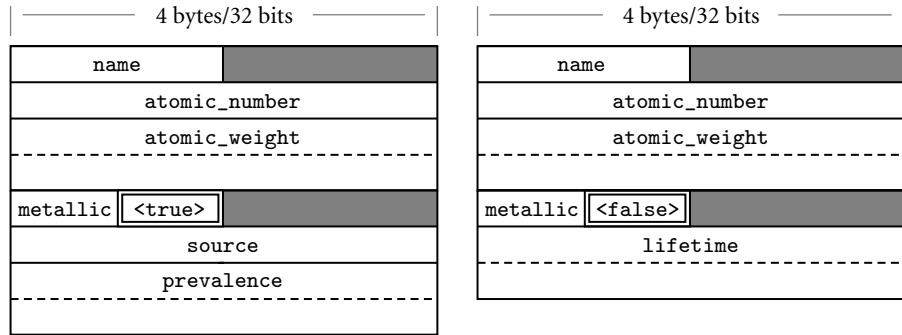


Figure 7.15 Likely memory layouts for `element` variants. The value of the `naturally_`
`occurring` field (shown here with a double border) determines which of the interpretations
of the remaining space is valid. Type `string_ptr` is assumed to be represented by a (4-byte)
pointer to dynamically allocated storage.

A `false` tag indicates that the element results only from atomic collisions or the decay of heavier elements; in this case, the record contains an additional field—`lifetime`—that indicates how long atoms so created tend to survive before undergoing radioactive decay. Each of the parenthesized field lists (one containing `source` and `prevalence`, the other containing `lifetime`) is known as a *variant*. Either the first or the second variant may be useful, but never both at once. From an implementation point of view, these nonoverlapping uses mean that the variants may share space (see Figure 7.15). ■

Variant records have their roots in the equivalence statement of Fortran I and in the union types of Algol 68. The Fortran syntax looks like this:

EXAMPLE 7.123

Fortran equivalence
statement

```
integer i
real r
logical b
equivalence (i, r, b)
```

The equivalence statement informs the compiler that `i`, `r`, and `b` will never be used at the same time, and should share the same space in memory. ■

Pascal's principal contribution to union types (retained by Modula and Ada) was to integrate them with records. This was an important contribution, because the need for alternative types seldom arises anywhere else. In our running example, we use the same field-name syntax to access both the `atomic_weight` and `lifetime` fields of an `element`, despite the fact that the former is present in every `element`, while the latter is present only in those that are not naturally occurring. Without the integration of records and unions, the notation is less convenient. Here's what it looks like in C:

EXAMPLE 7.124

Mixing structs and unions
in C

```

struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
    _Bool naturally_occurring;
    union {
        struct {
            char *source;
            double prevalence;
        } natural_info;
        double lifetime;
    } extra_fields;
} copper;

```

Because the union is not a part of the struct, we have to introduce two extra levels of naming. The third field is still `copper.atomic_weight`, but the `source` field must be accessed as `copper.extra_fields.natural_info.source`. A similar situation occurs in ML, in which datatypes can be used for unions, but the notation is not integrated with records (Exercise ©7.27). ■

Safety

EXAMPLE 7.125

Breaking type safety with equivalence

One of the principal problems with equivalence statements is that they provide no built-in means of determining which of the equivalence-ed objects is currently valid: the program must keep track. Mistakes in which the programmer writes to one object and then reads from the other are relatively common:

```

r = 3.0
...
print '(I10)', i

```

Here the `print` statement, which attempts to output `i` as a 10-digit 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 7.126

Union conformity in Algol 68

Fortran equivalence statements introduce an extreme case of aliases: not only are there two names for the “same thing” (in this case the same block of storage), but the types associated with those names are different. To address this potential source of bugs, the Algol 68 designers required that the language implementation track union-ed types at run time:

```

union (int, real, bool) uirb
    # uirb can be an integer, a floating-point number, or a Boolean #
...
uirb := 1          # uirb is now an integer #
...
uirb := 3.14       # uirb is now a floating-point number #

```

To use the value stored inside a union, the programmer must employ a special form of case statement (called a *conformity clause* in Algol 68) that determines which type is currently valid:

```
case uirb in
  (int i) : print(i),
  (real r) : print(r),
  (bool b) : print(b)
esac
```

The labels on the arms of the case statement provide names for the “deunified” values. A similar `tagcase` construct can be found in Clu. ■

To enforce correct usage of union types in Algol 68, the language implementation must maintain a hidden field for every union object that indicates which type is currently valid. When an object of a union type is assigned a value, the hidden field is also set, to indicate the type of the value just assigned. When execution encounters a conformity clause, the hidden field is inspected to determine which arm to execute.

In effect, the tag field of a Pascal variant record is an explicit representation of the hidden field required in an Algol 68 union. Our integer/floating-point/Boolean example could be written as follows in Pascal.

EXAMPLE 7.127

Tagged variant record in Pascal

```
type tag = (is_int, is_real, is_bool);
var uirb : record
  case which : tag of
    is_int : (i : integer);
    is_real : (r : real);
    is_bool : (b : Boolean)
  end;
end;
```

EXAMPLE 7.128

Breaking type safety with variant records

Unfortunately, while the hidden tag of an Algol 68 union can only be changed implicitly, by assigning a value of a different type to the union as a whole, the tag of a Pascal variant record can be changed by an ordinary assignment statement. The compiler can generate code to verify that a field in variant v is never accessed unless the value of the tag indicates that v is currently valid, but this is not enough to guarantee type safety. It can catch errors of the form

```
uirb.which := is_real;
uirb.r := 3.0;
...
writeln(uirb.i);    (* dynamic semantic error *)
```

but it cannot catch the following:

```
uirb.which := is_real;
uirb.r := 3.0;
uirb.which := is_int;
...                (* no intervening assignment to i *)
writeln(uirb.i);    (* ouch! *)
```

Any Pascal implementation will accept this code, but the output is likely to be erroneous, just as it was in Fortran. ■

Semantically speaking, changing the tag of a Pascal variant record should make the remaining fields of the variant *uninitialized*. It is possible, by adding hidden fields, to flag them as such and generate a semantic error message on any subsequent access, but the code to do so is expensive [FL80], and outlaws programs which, while arguably erroneous, are permitted by the language definition (Exercise ©7.33).

EXAMPLE 7.129

Untagged variants in Pascal

The situation in Pascal is actually worse than our example so far might imply. Additional insecurity stems from the fact that Pascal's tag fields are *optional*. We could drop the *which* field of our *uirb* record:

```
var uirb : record
    case tag of
        is_int : (i : integer);
        is_real : (r : real);
        is_bool : (b : Boolean)
    end;
...
uirb.r := 3.0;
...          (* no intervening assignment to i *)
writeln(uirb.i);  (* ouch! *)
```

Now the language implementation is not required to devote any space to either an explicit or hidden tag, but even the limited form of checking (make sure the tag has an appropriate value when a field of a variant is accessed) is no longer possible (but see Exercise ©7.34). Variant records with tags (explicit or hidden) are known as *discriminated unions*. Variant records without tags are known as *nondiscriminated unions*. ■

The degree of type safety provided is arguably the most important dimension of variation among the variant records and union types of modern languages. Though designed after Algol 68 (and borrowing its union terminology), the union types of C are semantically closer to Fortran's equivalence statements. Their fields share space, but nothing prevents the programmer from using them in inappropriate ways. By contrast, the variant records of Ada are syntactically similar to those of Pascal, but are as type-safe as the unions of Algol 68. Concerned at the lack of type safety in Pascal and Modula-2, and reluctant to introduce the complexity of Ada's rules, the designers of Modula-3 chose to eliminate variant records from the language entirely.

Variants in Ada

EXAMPLE 7.130

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., *A := {which => is_real,*

`r => pi}`;). 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 : real;
  metallic : Boolean;
  case naturally_occurring is
    when true =>
      source : string_ptr;
      prevalence : real;
    when false =>
      lifetime : real;
  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 7.131

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 7.132

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:

```
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 : real;
end record;
```

The `<>` notation in the initial definition of `element_array` indicates that the bounds are not statically known. We will have more to say about dynamic arrays in Section 7.4.2. 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). ■

DESIGN & IMPLEMENTATION**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 has a similar rule, designed for a similar purpose. When a variant record is allocated from the heap in Pascal (via the built-in `new` operator), the programmer has the option of specifying case labels for the variant portions of the record. A record so allocated is never allowed to change to a different variant, so the implementation can allocate precisely the right amount of space.

Modula-2, which does not provide `new` as a built-in operation, eliminates the ordering restriction on variants. All variables of a variant record type must be large enough to hold any variant. The usual implementation assigns a fixed offset to every field, with holes following small internal variants as necessary (see Figure ©7.16 and Exercise ©7.35).

```

TYPE element = RECORD
  name : ARRAY [1..2] OF CHAR;
  metallic : BOOLEAN;
  CASE naturally_occurring : BOOLEAN OF
    TRUE :
      source : string_ptr;
      prevalence : REAL;
    | FALSE :
      lifetime : REAL;
  END;
  atomic_number : INTEGER;
  atomic_weight : REAL;
END;

```

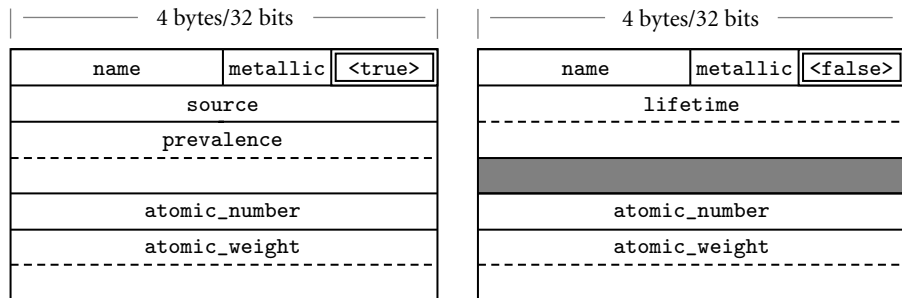


Figure 7.16 Likely memory layout for a variant record in Modula-2. Here the variant portion of the record is not required to lie at the end. Every field has a fixed offset from the beginning of the record, with internal holes as necessary following small-size variants.

The Object-Oriented Alternative

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, replaces variants with a more general mechanism for *type extension* (Section 9.2.4), and the designers of Java and C# dropped the unions of C and C++. In place of the C code of Example 7.124, a Java programmer might write

EXAMPLE 7.133

Derived types as an alternative to unions

```

class Element {
  public String name;
  public int atomic_number;
  public double atomic_weight;
  public boolean metallic;
}
class NaturalElement extends Element {
  public String source;
  public double prevalence;
}

```



```
class SyntheticElement extends Element {  
    public double lifetime;  
}
```

Like the unification of records and variants of Pascal, this approach avoids the artificial `extra_fields` and `natural_info` names of Example ©7.124. Like the discriminated subtypes of Ada, however, it constrains each variable to a single variant at elaboration time; this cannot be changed by subsequent assignment. ■

✓ CHECK YOUR UNDERSTANDING

68. Why is it useful to integrate variants (unions) with records (structs)? Why not leave them as separate mechanisms, as they are in Algol 68 and C?
 69. Discuss the type safety problems that arise with variant records. How can these problems be addressed?
 70. What is a *tag* (*discriminant*)? How does it differ from an ordinary field?
 71. Summarize the rules that prevent access to inappropriate fields of a variant record in Ada.
 72. Why might one wish to *constrain* a variable, so that it can hold only one variant of a type?
 73. Explain how classes and inheritance can be used to obtain the effect of constrained variant records.
-

