10 Polymorphism

Polymorphism means “many-formed”. Here we use the term to refer to facilities that enable the programmer to use a variable, value or subprogram in two or more different ways. Almost by definition polymorphism can be the source of bugs; it is difficult enough to understand a program where each name has one meaning, much less one where a name can have multiple meanings! Nevertheless, in many situations polymorphism is essential and can be safe if used carefully.

Polymorphism can be either static or dynamic. In static polymorphism, the multiple forms are resolved at compile time, and appropriate machine code generated. Examples are:

- Type conversion: a value is converted from one type to another.
- Overloading: the same name is used for two or more different objects or subprograms (including operators).
- Generics: a parameterized template of a subprogram is used to create several instances of a subprogram.

In dynamic polymorphism, the structural uncertainty is not resolved until run-time:

- Variant and unconstrained records: one variable can have values of different types.
- Run-time dispatching: the selection of a subprogram to call is done at run-time.

10.1 Type conversion

Type conversion is the operation of taking a value of one type and converting it to a value of another type. There are two variations: (1) translating a value of one type to a valid value of the other, and (2) transferring the value as an uninterpreted bit string.

Conversion of numeric values, say floating-point to integer, involves executing instructions to rearrange the bits of the floating-point value so that they represent a suitable integer. In effect, type conversion is done by a function receiving a parameter of one type and returning a result of another type; the Ada syntax for type conversion is the same as that of a function:

Ada

```
I: Integer := 5;
F: Float := Float(I);
```

while the C syntax can be confusing, especially when used in a complicated expression:

C

```
int i = 5;
float f = (float) i;
```

C++ retains the C syntax for compatibility but also introduces a functional syntax as in Ada for improved readability.

In addition, both C and C++ include implicit type conversion between types, primarily numerical types:

C

```
int i;
float f = i;
```

Explicit type conversions are safe because they are simply functions: if the predefined type conversion didn't exist, you could always write your own. Implicit type conversions are more problematical, because the reader of the program can never know if the conversion was intended or if it is an unintended oversight. Using integer values in a complicated floating-point expression should cause no problems, but other conversions should be explicitly written out.

The second form of type conversion simply allows the program to use the same bit string in two different ways. Unfortunately, C uses the same syntax for both forms of conversion: if it makes sense to do a type conversion, such as between numeric types or pointer types, a true conversion is done; otherwise, the bit string is transferred as it is.

In Ada, it is possible to do an *unchecked conversion* between any two types; the conversion treats the value as an uninterpreted bit string. Since this is inherently unsafe and renders all the hard-earned type checking invalid, unchecked conversion is discouraged and the syntax in Ada is designed to highlight its use. Uses of unchecked conversion will not be overlooked during a code inspection and the programmer will have to justify its use.

C++, while retaining C type conversion for compatibility, has defined a new set of *cast* operators:

dynamic_cast See Section 15.4.

static_cast An expression of type T1 can be statically cast to type T2, if T1 can be implicitly converted to T2 or conversely. static_cast would be used for type-safe conversions like float to int or conversely.

reinterpret_cast Unsafe type conversions.

const_cast Used to allow assignments to constant objects.

10.2 Overloading

Overloading is the use of the same name to denote different objects in a single scope. The use of the same name for variables in two different procedures (scopes) is not considered to be overloading, because the two variables do not exist simultaneously. The idea for overloading comes from the need to use mathematical and input-output libraries for variables of different types. In C, different names need to be used for the absolute value function on different types:

C

```
int      i      = abs(25);
double   d      = fabs(1.57);
long     l      = labs(−25L);
```

In Ada and in C++, the same name can be used for two or more different subprograms provided that the *parameter signatures* are different. As long as the number and/or the types (but not just the names or modes) of the formal parameters are different, the compiler will be able to resolve any subprogram call by checking the number and types of the actual parameters:

Ada

```
function Sin(X: in Float) return Float;
function Sin(X: in Long_Float) return Long_Float;

F1, F2: Float;
L1, L2: Long_Float;

F1 := Sin(F2);
L1 := Sin(L2);
```

An interesting difference between the two languages is that Ada takes the function result type into account when searching for overloading, while C++ restricts itself to the formal parameters:

C++

```
float    sin(float);
double   sin(double);     // Overloads sin
double   sin(float);      // Error, redefinition in a scope
```

Of particular interest is the possibility of overloading predefined operators like + and * in Ada:

Ada

```
function "+"(V1, V2: Vector) return Vector;
```

Of course you have to supply the function to implement the overloaded operator for the new types. Note that the syntactic properties of the operator, in particular its precedence, are not changed. C++ has a similar facility for overloading:

C++

```
Vector operator+(const Vector &, const Vector &);
```

This is just like a function declaration except for the use of the reserved keyword operator. Operator overloading should only be used for operations that are similar to the predefined meaning, so as not to confuse maintainers of the program.

If used carefully, overloading can reduce the size of the name space and ensure the portability of a program. It can even enhance the clarity of a program because artificial names like fabs are no longer needed. On the other hand, indiscriminate overloading can easily destroy readability by assigning too many meanings to the same name. Overloading should be restricted to subprograms that do the same sort of computation, so that the reader of a program can interpret the meaning just from the subprogram name.

10.3 Generics

Arrays, lists and trees are data structures that can store and retrieve data elements of arbitrary type. If it is necessary to store several types simultaneously, some form of dynamic polymorphism is needed. However, if we are working only with homogeneous data structures such as an array of integers, or a list of floating-point numbers, it is sufficient to use static polymorphism to create instances of a program template at compile time.

For example, consider a subprogram to sort an array. The type of the array element is used only in two places: when comparing elements and when swapping elements. The complex manipulation of indices is the same whatever the array element type:[1]

Ada

```
type Int_Array is array(Integer range <>) of Integer;

procedure Sort(A: Int_Array) is
    Temp, Min: Integer;
begin
    for I in A'First ..A'Last-1 loop
        Min := I;
        for J in I+1 .. A'Last loop
            if A(J) ¡ A(Min) then Min := J; end if;
                                -- Compare elements using "¡"
        end loop;
        Temp := A(I); A(I) := A(Min); A(Min) := Temp;
                                -- Swap elements using ":="
    end loop;
end Sort;
```

In fact, even the index type is irrelevant to the programming of the procedure, as long as a discrete type (such as characters or integers) is used.

To obtain a Sort procedure for some other element type such as Character, we could physically copy the code and make the necessary modifications, but this would introduce the possibility of errors. Furthermore, if we wish to modify the algorithm, we would have to do so in each copy separately. Ada defines a facility called *generics* that allows the programmer to define a template of a subprogram and then to create instances of the template for several types. While C lacks a

[1]This algorithm is called *selection sort*; it is much more efficient and no more difficult than the *bubble sort* algorithm that you may have been taught. The idea is to search for the smallest element remaining in the unsorted part of the array and to move it to the front.

similar facility, its absence is less serious because void pointers, the sizeof operator and pointers to functions can be used to program general, if unsafe, subprograms. Note that the use of generics does not ensure that any of the object code will be common to the instantiations; in fact, an implementation may choose to produce independent object code for each instantiation.

Here is a declaration of a *generic subprogram* with two *generic formal parameters*:

Ada

```
generic
   type Item is (<>);
   type Item_Array is array(Integer range <>) of Item;
procedure Sort(A: Item_Array);
```

This generic declaration does *not* declare a procedure, only a template of a procedure. A procedure body must be supplied; the body will be written in terms of the generic parameters:

Ada

```
procedure Sort(A: Item_Array) is
   Temp, Min: Item;
begin
   . . .                        -- Exactly as before
end Sort;
```

To get a (callable) procedure, you must *instantiate* the generic declaration, that is, create an instance by furnishing generic actual parameters:

Ada

```
type Int_Array is array(Integer range <>) of Integer;
type Char_Array is array(Integer range <>) of Character;

procedure Int_Sort(A: Int_Array) is new Sort(Integer, Int_Array);
procedure Char_Sort(A: Char_Array) is new Sort(Character, Char_Array);
```

These are actual procedure declarations; instead of a body following the is keyword, a new copy of the generic template is requested.

The generic parameters are *compile-time* parameters and are used by the compiler to generate the correct code for the instance. The parameters form a *contract* between the code of the generic procedure and the instantiation. The first parameter, Item, is declared with the notation (<>), which means that the instantiating program promises to supply a discrete type such as Integer or Character, and the code promises to use only operations valid on such types. Since every discrete type has the relational operators defined on its values, the procedure Sort is assured that "¡" is valid. The second generic parameter Item_Array is a clause in the contract that says: whatever type was given for the first parameter, the second parameter must be an integer-indexed array of that type.

The contract model works both ways. An attempt to do an arithmetic operation such as "+" on values of type Item in the generic body is a compilation error, since there are discrete types such as Boolean for which arithmetic is not defined. Conversely, the generic procedure could not be instantiated with a record type because the procedure needs “¡” which is not defined for records.

The motivation for the contract model is to allow programmers to use or reuse generic units with the certainty that they need not know how the generic body is implemented. Once the generic body compiles correctly, an instantiation can fail only if its actual parameters do not fit the contract. An instantiation will *not* cause a compile error in the body.

Templates in C++

In C++, generics are implemented with the *template* facility:

```
template ¡class Item_Array¿ void Sort(Item_Array parm)
{
        . . .
}
```

There is no need for explicit instantiation; a subprogram is created implicitly when the subprogram is used:

```
typedef int I_Array[100];
typedef char C_Array[100];
I_Array a;
C_Array c;

Sort(a);                    // Instantiate for int arrays
Sort(c);                    // Instantiate for char arrays
```

Explicit instantiation is an optional programmer-specified optimization; otherwise, the compiler decides exactly which instantiations need to be done. Templates can only be instantiated with types and values, or more generally with classes (see Chapter 14).

C++ does not use the contract model, so an instantiation can fail by causing a compilation error in the template definition. This makes it more difficult to supply templates as proprietary software components.

* Generic subprogram parameters in Ada

Ada allows generic parameters to be subprograms. The sort example can be written:

```
generic
   type Item is private;
   type Item_Array is array(Integer range <>) of Item;
   with function "<"(X, Y: in Item) return Boolean;
procedure Sort(A: Item_Array);
```

The contract is now extended to require that a Boolean function be supplied for the operator "<". Since the comparison operator is supplied, Item need no longer be restricted to discrete types for which the operator is predefined. The keyword private means that any type upon which assignment and equality are defined can be supplied during instantiation:

```
type Rec is record ... end record;
type Rec_Array is array(Integer range <>) of Rec;
function "<"(R1, R2: in Rec) return Boolean;

procedure Rec_Sort(A: Rec_Array) is new Sort(Rec, Rec_Array, "<");
```

Within the Sort subprogram, assignment is the usual bitwise assignment of the record values, and when two records must be compared the function "¡" is called. This programmer-supplied function will decide if one record value is smaller than another.

The contract model of Ada is very powerful: types, constants, variables, pointers, arrays, subprograms and packages (in Ada 95) can be used as generic parameters.

10.4 Polymorphic data structures in Java

In Ada and C++, we have two ways of constructing polymorphic data structures: generics in Ada and templates in C++ for compile-time polymorphism, and class-wide types in Ada and pointers/references to classes in C++ for run-time polymorphism. The advantage of generics/templates is that the data structure is fixed when it is instantiated at compile-time; this can improve both the efficiency of code generation and the memory that needs to be allocated for the data structure.

Java chooses to implement only run-time polymorphism. As in Smalltalk and Eiffel, every class in Java is considered to be derived from a root class called Object. This means that a value of any non-primitive type[2] can be assigned to an object of type Object. (Of course this works because of the reference semantics.)

To create a linked list, a node class would first be defined as containing (a pointer to) an Object. The list class would then contain methods to insert and retrieve values of type Object:

Java

```
class Node {
    Object data;
    Node next;
}

class List {
    private Node head;
    void Put(Object data) { ... };
    Object Get( ) { ... };
}
```

[2]To allow a primitive type to be considered as derived from Object, "wrapper" classes such as Integer are defined.

If L is an object of type List and a is an object of type Airplane_Data, then L.Put(a) is valid because Airplane_Data is derived from Object. When a value is retrieved from the list, it must be cast to the proper descendant of Object:

Java

```
a = (Airplane_Data) List.Get( );
```

Of course, if the value returned is not of type Airplane_Data (or descended from the type), an exception will be raised.

The advantage of this paradigm is that it is very easy to write generalized data structures in Java, but compared with generics/templates, there are two disadvantages: (1) the additional overhead of the reference semantics (even for a list of integers!), and (2) the danger that an object placed on the wrong queue will cause a run-time error when retrieved.

10.5 Variant records

Variant records are used when it is necessary to interpret a value in several different ways at run-time. Common examples are:

- Messages in a communications system and parameter blocks in operating system calls. Usually, the first field of the record is a code whose value determines the number and types of the remaining fields in the record.
- Heterogeneous data structures, such as a tree which may contain nodes of various types.

To solve these types of problems, programming languages introduce a new category of types called *variant records* which have *alternative* lists of fields. Thus a variable may initially contain a value of one variant and later be assigned a value of another variant with a completely different set of fields. In addition to the alternative fields, there may be fields which are common to all records of this type; such fields usually include a code which is used by the program to determine which variant is actually being used. For example, suppose that we wish to create a variant record whose fields may be either an array or a record:

C

```
typedef int Arr[10];
typedef struct {
    float  f1;
    int    i1;
} Rec;
```

Let us first define a type that encodes the variant:

C

```
typedef enum {Record_Code, Array_Code} Codes;
```

Now a *union type* in C can be used to create a variant record which can itself be embedded into a structure that includes the common tag field:

C

```
typedef struct {
    Codes code;             /* Common tag field */
    union {                 /* Union with alternative fields */
        Arr a;              /* Array alternative */
        Rec r;              /* Record alternative */
    } data;
} S_Type;

S_Type s;
```

From a syntactical point of view, this is just ordinary nesting of records and arrays within other records. The difference is in the implementation: the field data is allocated enough memory to contain the larger of the array field a or the record field r (Figure 10.1). Since enough memory is

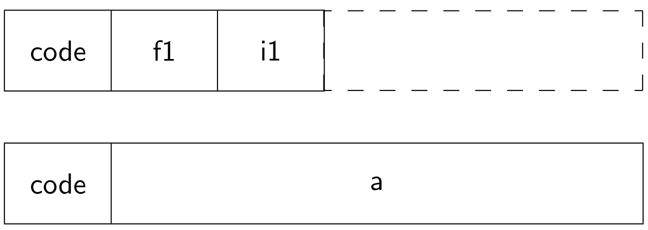

Figure 10.1: Variant records

allocated to accommodate the largest possible field, variant records can be extremely wasteful of memory if one alternative is very large and the others small:

C

```
union {
    int a[1000];
    float f;
    char c;
};
```

At the cost of complicating the programming, the best solution in this case is to use a pointer to the long fields.

The assumption underlying variant records is that exactly one of the fields of the union is meaningful at any one time, unlike an ordinary record where all fields exist simultaneously:

C

```
if (s.code == Array_Code)
    i = s.data.a[5];        /* Access first variant */
else
    i = s.data.r.i1;        /* Access second variant */
```

The main problem with variant records is that they can potentially cause serious bugs. It is possible to treat a value of one type as if it were a value of any other type (say to access a floating-point number as an integer), since the union construction enables the program to access the same bit string in different ways. In fact, Pascal programmers use variant records to do type conversion which is not directly supported in the language. In the above example, the situation is even worse because it is possible to access memory locations which have no meaning at all: the field s.data.r might be 8 bytes long to accommodate the two numbers, while the field s.data.a might be 20 bytes long to accommodate ten integers. If a valid record is stored in s.data.r, s.data.a[5] is meaningless.

Ada does not allow variant records to be used to break type checking. The "code" field that we used in the example is now a required field called the *discriminant*, and accesses to the variant fields are checked against the value of the discriminant. The discriminant appears as a "parameter" to the type:

Ada

```
type Codes is (Record_Code, Array_Code);

type S_Type(Code: Codes) is
   record
      case Code is
         when Record_Code => R: Rec;
         when Array_Code => A: Arr;
      end case;
   end record;
```

and a record must be declared with a specific discriminant so that the compiler can know exactly how much memory to allocate:

Ada

```
S1: S_Type(Record_Code);
S2: S_Type(Array_Code);
```

Alternatively, a pointer to the variant record can be declared and the discriminant set at run-time:

Ada

```
type Ptr is access S_Type;

P: Ptr := new S_Type(Record_Code);
I := P.R.I1;                    -- OK
I := P.A(5);                    -- Error
```

The first of the assignment statements is correct because the discriminant of the record P.all is Record_Code which ensures that the field R exists, while the second causes a run-time exception because the discriminant does not match the requested field.

The basic rule concerning discriminants in Ada is that they can be read but not written, so that the type checking cannot be bypassed. This also means that memory can be allocated exactly according to the variant chosen, rather than always allocating according to the largest variant.

* Unconstrained records in Ada

In addition to the constrained records whose variant is fixed when the variable is created, Ada allows the declaration of *unconstrained records* which allow a type-safe way of assigning values of different variants to a variable during the execution of a program:

```
S1, S2: S_Type;              -- Unconstrained records

S1 := (Record_Code, 4.5);
S2 := (Array_Code, 1..10 => 17);
S1 := S2;                    -- Assign a different variant to S1
                             -- S2 is larger than S1 !
```

Two rules ensure that the type checking is not broken:

- A default value must be given for the discriminant to ensure that initially the record has a valid discriminant:

  ```
  type S_Type(Code: Codes := Record_Code) is . . .
  ```

- The discriminant field cannot be changed by itself. Only assignment of a valid value to the entire record is acceptable as shown in the example.

There are two possible implementations of unconstrained records. It is possible to create every variable with the maximum size needed to hold any variant. Alternatively, implicit heap allocation can be used so that when it is required to assign a large value to a variable, memory can be freed and reallocated. Most implementations choose the first method: it is both simpler and does not use implicit heap allocation which is undesirable in many applications.

10.6 Dynamic dispatching

Suppose that each variant of the record S_Type is to be processed by its own subprogram. A case-statement must be used to *dispatch* to the appropriate subprogram:

Ada

```
procedure Dispatch(S: S_Type) is
begin
   case S.Code is
      when Record_Code => Process_Rec(S);
      when Array_Code => Process_Array(S);
   end case;
end Dispatch;
```

Suppose now further that during modification of the program, it is necessary to add an additional variant to the record. The modifications to the program are not difficult: add a code to the type

Codes, add an alternative to the case-statement in Dispatch, and add the new processing subprogram. As easy as these modifications are, they can be problematic in large systems because they require that the *source* code of existing, well-tested program components be modified and recompiled. Additionally, retesting is probably necessary to ensure that the modification of the global enumeration type does not have unintended side-effects.

The solution is to arrange for "dispatching" code to be part of the run-time system of the language, rather than explicitly programmed as shown above. This is called dynamic polymorphism, since it is now possible to call a general routine Process(S), and the binding of the call to a specific routine can be delayed until run-time when the current tag of S can be evaluated. This polymorphism is supported by *virtual functions* in C++ and by subprograms with *class-wide parameters* in Ada 95 (see Chapter 14).

10.7 Exercises

1. Why doesn't C++ use the result type to distinguish overloaded functions?
2. What problems does overloading pose for the linker?
3. In C++, the operators "++" and "−−" can be both prefix and postfix. What are the implications for overloading and how does C++ cope with these operators?
4. Neither Ada nor C++ allows overloading to modify the precedence or associativity of an operator; why?
5. Write a template sort program in C++.
6. Write a generic sort program in Ada and use it to sort an array of records.
7. The first generic sort program defined the item type as (<>). Can Long_Integer be used in an instantiation of this procedure? How about Float?
8. Write a program that maintains a heterogeneous queue, that is, a queue whose nodes can contain values of more than one type. Each node will be a variant record with alternative fields for Boolean, integer and character values.