14 Object-Oriented Programming

14.1 Object-oriented design

The previous chapter discussed language support for structuring programs, but it did not attempt to answer the question: how should programs be decomposed into modules? Normally, this subject would be studied in a course on software engineering, but one method of decomposition called *object-oriented programming* (OOP) is so important that modern programming languages directly support this method. The next two chapters will be devoted to the topic of language support for OOP.

When designing a program, the natural approach is to examine the requirements in terms of functions or operations, that is to ask: what should the program *do*? For example, software to support the tasks of an airline reservations clerk would have its requirements stated as follows:

1. Accept the customer's destination and departure date from the clerk.
2. Display the available flights to the clerk.
3. Accept the reservation request for a specific flight from the clerk.
4. Confirm the reservation and print the ticket.

This translates naturally into the design shown in Figure 14.1 with a module for each function and a "main" module to call the others.

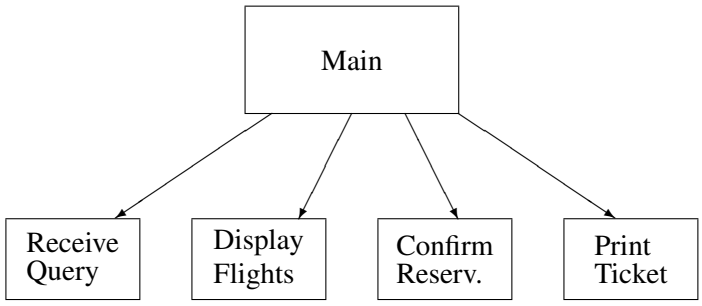

Figure 14.1: Functional decomposition

Unfortunately, this design is not *robust*; even minor modifications in the requirements can require extensive modifications of the software. For example, suppose that the airline improves working

conditions by replacing obsolete display terminals. The new terminals are likely to require modifications to all four modules; similarly, a new code-sharing arrangement with another airline will require extensive modifications.

But we all know that software modification is extremely error-prone; a design which is not robust will thus cause the delivered software system to be unreliable and unstable. You might object that people should refrain from modifying software, but the whole justification for software is that it is "soft" and thus "inexpensive" to change; otherwise, all applications would be efficiently wired-in like the program of a pocket calculator.

Software can be made much more robust and reliable by changing the focus of our design criteria. The correct question to ask is: *on what* is the software working? The emphasis is changed from a focus on functionality to a focus on external devices, internal data structures, and real-world models, collectively called *objects*. A module should be constructed for every "object", and should contain all data and operations needed to implement the object. In the example we can identify several objects as shown in Figure 14.2.

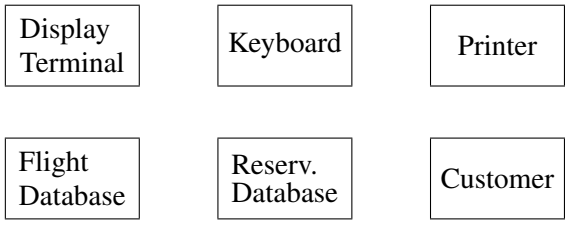

Figure 14.2: Object-oriented design

External devices such as the display terminal and the printer are identified as objects, as are the databases of flight information and reservations. In addition, we have designed an object Customer, whose purpose is to model an imaginary form on which the clerk enters data until the flight is confirmed and the ticket issued. This design is robust to modifications:

- Modifications to use a different terminal can be limited to the Terminal object. The programming of this object maps the customer data into actual display and keyboard commands, so the Customer object need not be modified, just the mapping onto the new hardware.
- Code-sharing may certainly require a total reorganization of the database, but as far as the rest of the program is concerned, one two-letter airline code is the same as another.

Object-oriented design is not limited to modelling real-world objects; it can also be used to create reusable software components. This is directly linked with one of the programming language concepts we have emphasized: abstraction. Modules implementing data structures can be designed and programmed as objects which are instances of an abstract data type together with the operations to manipulate the data. Abstraction is achieved by hiding the representation of the data type within the object.

In fact, the main difference between object-oriented design and "ordinary" programming is that in ordinary programming we are limited to predefined abstractions, while in object-oriented design

we can define our own abstractions. For example, floating-point numbers (Chapter 9) are nothing more than a useful abstraction of a difficult manipulation of data in the computer. It would be helpful if all programming languages contained predefined abstractions for every object we would ever need (complex numbers, rational numbers, vectors, matrices, etc., etc.), but there is no end to the useful abstractions. Eventually a programming language has to stop and leave the work to the programmer.

How can a programmer create new abstractions? One way is to use coding conventions and documentation ("the first element of the array is the real part and the second is the imaginary part"). Alternatively, the language can supply a construct like Ada private types which enable the programmer to explicitly define new abstractions; these abstractions will be compiled and checked just like predefined abstractions. OOP can (and should) be done in ordinary languages, but like other ideas in programming, it works best when done in languages that directly support the concept. The basic construct used in languages that support OOP is the abstract data type that was discussed in the previous chapter, but it is important to realize that object-oriented design is more general and extends to abstracting external devices, models of the real world, and so on.

Object-oriented design is extremely difficult. The reason is that it requires a great deal of experience and judgement to decide if something deserves to become an object or not. First-time users of object-oriented design tend to become over-enthusiastic and make everything an object; this leads to programs so dense and tedious that all the advantages of the method are lost. The best rule-of-thumb to use is based on the concept of *information hiding*:

> Every object should hide one major design decision.

It helps to ask yourself: "is the decision likely to change over the lifetime of the program?".

The specific display terminals and printers chosen for a reservation system are clearly subject to upgrading. Similarly, decisions on database organization are likely to change, if only to improve performance as the system grows. On the other hand, it could be argued that the customer data form is unlikely to change, and need not be a separate object. Even if you don't agree with our design decision to create a Customer object, you should agree that object-oriented design is a good framework in which to discuss design issues and to argue the merits of one design over another.

The following sections will discuss language support for OOP using two languages: C++ and Ada 95. First we look at C++ which was designed by adding a single integrated construct for OOP on top of C which does not even have support for modules. Then we will see how full object-oriented programming is defined in Ada 95 by adding a few small constructs to Ada 83 which already had many features that partially supported OOP.

14.2 Object-oriented programming in C++

A programming language is said to support OOP if it includes constructs for:

- Encapsulation and data abstraction
- Inheritance

- Dynamic polymorphism

Let us recall the discussion of encapsulation and data abstraction from the previous chapter.

Modules such as Ada packages encapsulate computational resources, exposing only an interface specification. Data abstraction can be achieved by specifying the data representation in the private part which cannot be accessed from other units. The unit of encapsulation and abstraction in C++ is the *class* which contains data type and subprogram declarations. Actual objects, called *instances*, are created from the class. An example of a class in C++ is:

```
class Airplane_Data {
public:
    char *get_id(char *s) const {return id;}
    void set_id(char *s)       {strcpy(id, s);}
    int get_speed( ) const     {return speed;}
    void set_speed(int i)      {speed = i;}
    int get_altitude( ) const  {return altitude;}
    void set_altitude(int i)   {altitude = i;}
private:
    char id[80];
    int speed;
    int altitude;
};
```

This example extends the example of the previous chapter by creating a separate class for the data on each airplane. The class can now be *used* by another class, for example by one that defines a data structure for storing many airplanes:

```
class Airplanes {
public:
    void New_Airplane(Airplane_Data, int &);
    void Get_Airplane(int, Airplane_Data &) const;
private:
    Airplane_Data database[100];
    int current_airplanes;
    int find_empty_entry( );
};
```

Each class is designed to encapsulate a set of data declarations. The data declarations in the private part can be modified without modifying programs that use the class (called *clients* of the class), though you will need to recompile. A class will have a set of interface functions which will retrieve and update data values that are internal to the class.

You may question why Airplane_Data should be made a separate class, rather than just being declared as an ordinary public record. This is a debatable design decision: data should be hidden in a class if you believe that the internal representation is likely to change. For example, you

may know that one customer prefers to measure altitude in English units (feet) while another prefers metric units (meters). By defining a separate class for Airplane_Data, you can use the same software for both customers and only change the implementation of the access functions.

There is a price to pay for this flexibility; *every* access to a data value requires a subprogram call:

```
Aircraft_Data a;              // Instance of the class
int alt;

alt = a.get_altitude( );      // Get value hidden in instance
alt = (alt * 2) + 1000;
a.set_altitude(alt);          // Return value to instance
```

instead of a simple assignment statement if a were a public record:

```
a.alt = (a.alt * 2) + 1000;
```

Programming can be very tedious and the resulting code hard to read and understand, because the access functions obscure the intended processing. Thus classes should be defined only when there is a clear advantage to be obtained from hiding implementation details of an abstract data type.

However, there need not be any significant run-time overhead for encapsulation. As shown in the example, the body of the interface function can be written within the class declaration; in this case the function is an *inline* function,[1] which means that the mechanism of a subprogram call and return (see Chapter 7) is not used. Instead, the code for the subprogram body is inserted directly within the sequence of code at the point of call. Since inlining trades space for time, it should be restricted to very small subprograms (no more than two or three instructions). Another factor to consider before inlining a subprogram is that it introduces additional compilation conditions. If you modify an inlined subprogram, all clients must be recompiled.

14.3 Inheritance

In Section 4.6 we showed how in Ada one type can be derived from another so that the derived type has a copy of the values and a copy of the operations that were defined for the parent type. Given the parent type:

Ada

```
package Airplane_Package is
   type Airplane_Data is
      record
         ID: String(1..80);
         Speed: Integer range 0..1000;
         Altitude: Integer range 0..100;
      end record;
```

[1] In C++ a function can be inline even if the body is written separately and not in the class declaration.

```
    procedure New_Airplane(Data: in Airplane_Data; I: out Integer);
    procedure Get_Airplane(I: in Integer; Data: out Airplane_Data);
end Airplane_Package;
```

a derived type can be declared in another package:

Ada

```
type New_Airplane_Data is
    new Airplane_Package.Airplane_Data;
```

You can declare new subprograms that operate on the derived type and you can replace a subprogram of the parent type with a new one:

Ada

```
procedure Display_Airplane(Data: in New_Airplane_Data);
        -- Additional subprogram
procedure Get_Airplane(Data: in New_Airplane_Data; I: out Integer);
        -- Replaced subprogram
        -- Subprogram New_Airplane copied from Airplane_Data
```

Derived types form a family of types and a value of any type in the family can be converted to that of another type in the family:

Ada

```
A1: Airplane_Data;
A2: New_Airplane_Data := New_Airplane_Data(A1);
A3: Airplane_Data := Airplane_Data(A2);
```

Furthermore, you can even derive from a private type, though of course all subprograms for the derived type must be defined in terms of public subprograms of the parent type.

The problem with derived types in Ada is that only the *operations* can be extended, not the data components that form the type. For example, suppose that the air-traffic control system must be modified so that for supersonic aircraft the Mach number[2] is stored in addition to the existing data. One possibility is simply to modify the existing record to include the extra field. This solution is acceptable if the modification is made during the initial development of the program. However, if the system has already been tested and installed, it would be better to find a solution that does not require all the existing source code to be recompiled and checked.

The solution is to use *inheritance*, which is a way of extending an existing type, not just by adding and modifying operations, but also by adding data to the type. In C++ this is done by deriving one class from another:

C++

```
class SST_Data: public Airplane_Data {
private:
    float mach;
public:
```

[2]The speed in terms of the speed of sound.

```
    float get_mach( ) const {return mach;};
    void set_mach(float m) {mach = m;};
};
```

The derived class SST_Data is derived from an existing class Airplane_Data. This means that every data element and subprogram that are defined for the *base class*[3] are also available in the derived class. In addition, values of the derived class SST_Data will each have an additional data component mach, and there are two new subprograms that can be applied to values of the derived type.

The derived class is an ordinary class in the sense that instances can be declared and subprograms invoked:

C++

```
SST_Data s;

s.set_speed(1400);          // Inherited subprogram
s.set_mach(2.4);            // New subprogram
```

The subprogram called for set_mach is the one which is declared within class SST_Data, while the subprogram called for set_speed is the one inherited from the base. Note that the derived class can be compiled and linked without modifying and recompiling the base class; thus existing code need not be affected by the extension.

14.4 Dynamic polymorphism in C++

When one class is derived from another class, you can *override* inherited subprograms in the derived class by redefining the subprogram:

```
class SST_Data: public Airplane_Data {
public:
    int get_speed( ) const;   // Override
    void set_speed(int);      // Override
};
```

Given a call:

```
obj.set_speed(100);
```

the decision as to which of the subprograms to call—the subprogram inherited from Airplane_Data or the new one in SST_Data—is made at compile-time based on the class of the object obj. This is called *static binding* or *early binding*, since the decision is made before the program is run, and each call is always to the same subprogram.

[3]Parent type in Ada is roughly the same as base class in C++.

However, the whole point of inheritance is to create a group of classes with similar properties, and it is reasonable to expect that it should be possible to assign a value belonging to any of these classes to a variable. What should happen when a subprogram is called for such a variable? The decision as to which subprogram to call must be made at run-time because the value held in the variable is not known until then; in fact, the variable may hold values of different classes at different times during the program. The terms used to denote the ability to select subprograms at run-time are *dynamic polymorphism*, *dynamic binding*, *late binding*, and *run-time dispatching*.

In C++, *virtual functions* are used to denote those subprograms for which dynamic binding is performed:

```
class Airplane_Data {
private:
   ...
public:
   virtual int get_speed( ) const;
   virtual void set_speed(int);
   ...
};
```

A subprogram in the derived class with the same name and parameter signature as that of a virtual subprogram in the parent class is also considered virtual. You are not required to repeat the virtual specifier, but it is good practice to do so for clarity:

```
class SST_Data : public Airplane_Data {
private:
   float mach;
public:
   float get_mach( ) const;       // New subprogram
   void set_mach(float m);        // New subprogram
   virtual int get_speed( ) const;// Override virtual subprogram
   virtual void set_speed(int);   // Override virtual subprogram
   ...
};
```

Consider now the procedure update which takes a reference parameter to the base class:

```
void update(Airplane_Data & d, int spd, int alt)
{
   d.set_speed(spd);         // What type does d point to ??
   d.set_altitude(alt);      // What type does d point to ??
}

Airplane_Data a;
SST_Data s;
```

```
void proc( )
{
   update(a, 500, 5000);   // Call with Airplane_Data
   update(s, 800, 6000);   // Call with SST_Data
}
```

The idea of derived classes is that a derived value *is* a base value (perhaps with additional fields), so update can also be called with the value s of the derived class SST_Data. Within update, the compiler has no way of knowing what d points to: a value of Airplane_Data or a value of SST_Data. So it has no way of compiling an unambiguous call to set_speed which is defined differently for the two types. Therefore, the compiler must create code to *dispatch* the call to the correct subprogram at run-time, depending on the class of the value pointed to by d. The first call in proc passes a value of type Airplane_Data to d, so the call to set_speed will be dispatched to the subprogram defined in the class Airplane_Data while the second call will be dispatched to the subprogram defined in SST_Data.

Let us stress the advantages of dynamic polymorphism: you can write large portions of a program in a completely general fashion using calls to virtual subprograms. The specialization of the processing to a specific class in a family of derived classes is made only at run-time, by dispatching on the virtual subprograms. Furthermore, if you ever need to add derived classes to the family, none of the existing code need be modified or recompiled, because any change in the existing computation is solely limited to the new implementations of the virtual subprograms. For example, if we derive another class:

```
class Space_Plane_Data : public SST_Data {
   virtual void set_speed(int);   // Override virtual subprogram
private:
   int reentry_speed;
};

Space_Plane_Data sp;
update(sp, 2000, 30000);
```

the file containing the definition of update need not be recompiled, even though (i) a new subprogram has overridden set_speed, and (ii) the value of the formal parameter d of update contains an additional field reentry_speed.

When is dynamic polymorphism used?

Let us declare a base class with a virtual subprogram and an ordinary non-virtual subprogram, and let us derive a class that adds an additional field and supplies new declarations for both subprograms:

```
class Base_Class {
```

```
private:
    int Base_Field;
public:
    virtual void virtual_proc( );
    void ordinary_proc( );
};
class Derived_Class : public Base_Class {
private:
    int Derived_Field;
public:
    virtual void virtual_proc( );
    void ordinary_proc( );
};
```

Next let us declare instances of the classes as variables. Assignment of a value of the derived class to a variable of the base class is permitted:[4]

```
Base_Class    Base_Object;
Derived_Class Derived_Object;
if (...) Base_Object = Derived_Object;
```

because the derived object *is* a base object (plus extra information), and the extra information can be ignored in the assignment (Figure 14.3).

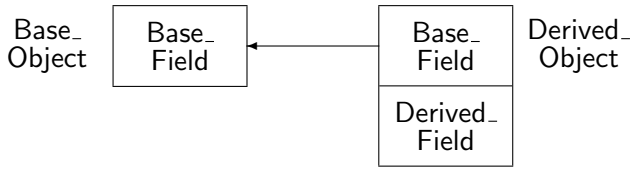

Figure 14.3: Direct assignment of a derived object

Furthermore, a call to a subprogram (whether virtual or not) is unambiguous and the compiler can use static binding:

```
Base_Object.virtual_proc( );
Base_Object.ordinary_proc( );
Derived_Object.virtual_proc( );
Derived_Object.ordinary_proc( );
```

Suppose, however, that indirect allocation is used and a pointer to a derived class is assigned to a pointer to the base class:

```
Base_Class*    Base_Ptr = new Base_Class;
Derived_Class* Derived_Ptr = new Derived_Class;
if (...) Base_Ptr = Derived_Ptr;
```

[4]See section 15.4 on assignment of objects from base to derived classes.

In this case the semantics are different since the base pointer points to the *complete* derived object and no truncation is done (Figure 14.4). There is no implementation problem because we assume

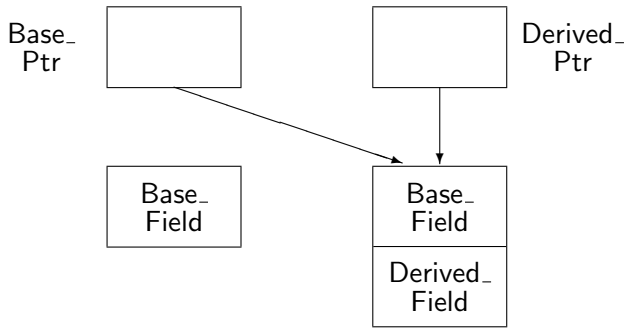

Figure 14.4: Indirect assignment of a derived object

that all pointers are represented identically regardless of the designated type.

What is important to note is that following the pointer assignment, the *compiler* no longer has any information as to the type of the designated object. Thus it has no way of binding a call:

```
Base_Ptr->virtual_proc( );
```

to the correct subprogram and dynamic dispatching must be done. A similar situation occurs when a reference parameter is used as was shown above.

This situation is potentially confusing since programmers usually don't distinguish between a variable and a designated object. After the following statements:

```
int i1 = 1;
int i2 = 2;
int *p1 = &i1;          // p1 points to i1
int *p2 = &i2;          // p2 points to i2
p1 = p2;                // p1 also points to i2
i1 = i2;                // i1 has the same value as i2
```

you expect that i1==i2 and that *p1==*p2; this is certainly true as long as the types are exactly the same, but it is not true for assignment from a derived to a base class because of the truncation. When using inheritance you have to remember that the designated object may not be of the same type as the designed type in the pointer declaration.

There is a pitfall in the semantics of dynamic polymorphism in C++: if you look carefully, you will note that the discussion was about dispatching to an overridden *virtual* subprogram. But there may also be ordinary subprograms in the class which are overridden:

```
Base_Ptr = Derived_Ptr;
Base_Ptr->virtual_proc( );// Dispatches on designated type
Base_Ptr->ordinary_proc( );// Statically bound to base type !!
```

There is a semantic difference between the two calls: the call to the virtual subprogram is dispatched at run-time according to the *type of the designated object*, in this case Derived_Class; the call to the ordinary subprogram is bound at compile-time according to the *type of the pointer*, in this case Base_Class. This difference is dangerous because a modification that changes a non-virtual subprogram to a virtual subprogram, or conversely, can cause bugs in the entire family of classes derived from the base class.

> Dynamic dispatching in C++ is done on calls to virtual subprograms made through a pointer or reference.

Implementation

Earlier we noted that if a subprogram is not found in a derived class, a search is made in ancestor classes until a definition of the subprogram is found. In the case of static binding, the search can be made at compile-time: the compiler looks at the base class of the derived class, then at its base class, and so on, until an appropriate subprogram binding is found. Then an ordinary procedure call can be compiled to that subprogram.

If virtual subprograms are used, the situation is more complicated because the actual subprogram to call is not known until run-time. Note that if a virtual subprogram is called with an object of a specific type, as opposed to a reference or pointer, static binding can still be used. Otherwise, deciding which subprogram to call is based on (1) the name of the subprogram, and (2) the class of the object. But (1) is known at compile-time, so all we need to do is to simulate a case-statement on the class.

The usual implementation is slightly different; for each class with virtual subprograms, a dispatch table is maintained (Figure 14.5). Each value of a class must "drag" with it an *index* into the

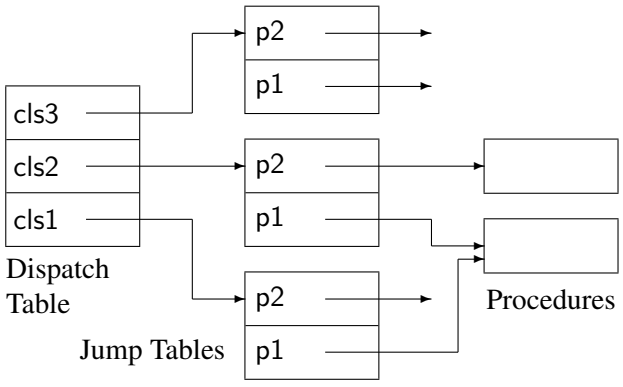

Figure 14.5: Implementation of dynamic polymorphism

dispatch table for the derivation family in which it is defined. The dispatch table entries are pointers to jump tables; in each jump table there is an entry for each virtual subprogram. Note that two jump table entries can point to the same procedure; this will happen when a class does not override a virtual subprogram. In the Figure, cls3 is derived from cls2 which in turn is derived from the base class cls1. cls2 has overridden p2 but not p1, while cls3 has overridden both subprograms.

When a dispatching subprogram call ptr−>p1( ) is encountered, code such as the following is executed, where we assume that the implicit index is the first field of the designated object:

```
load    R0,ptr          Get address of object
load    R1,(R0)         Get index of designated object
load    R2,&dispatch    Get address of dispatch table
add     R2,R1           Compute address of jump table
load    R3,(R2)         Get address of jump table
load    R4,p1(R3)       Get address of procedure
call    (R4)            Call procedure, address is in R4
```

Even without further optimization, run-time overhead is relatively small, and more importantly *fixed*, so in most applications there is no need to refrain from using dynamic polymorphism. Even so the overhead does exist and run-time dispatching should be used after careful analysis. Avoid either extreme: excessive use of dynamic polymorphism just because it is a "good idea", or refraining from its use because it is "inefficient".

Note that the fixed overhead is obtained because the dynamic polymorphism is limited to the fixed set of classes derived from a base class (so a fixed-size dispatch table can be used), and to the fixed set of virtual functions that can be overridden (so the size of each jump table is also fixed). It was a significant achievement of C++ to show that dynamic polymorphism can be implemented without an unbounded run-time search.

14.5 Object-oriented programming in Ada 95

Ada 83 fully supports encapsulation with the package construct and partially supports inheritance through derived types. Derived types do not support full inheritance because when you derive a new type you can only add new operations but not new data components. In addition, the only polymorphism is the static polymorphism of variant records. In Ada 95, full inheritance is supported by enabling the programmer to *extend* a record when deriving a type. To denote that a parent record type is eligible for inheritance, it must be declared as a *tagged* record type:

```
package Airplane_Package is
   type Airplane_Data is tagged
      record
         ID: String(1..80);
         Speed: Integer range 0..1000;
         Altitude: Integer range 0..100;
      end record;
end Airplane_Package;
```

The tag is similar to the Pascal tag and the Ada discriminant in variant records in that it is used to distinguish between the different types that are derived from each other. Unlike those constructs, a

tag of a tagged record is implicit and the programmer does not need to explicitly access it.[5] Looking ahead, this implicit tag will be used to dispatch subprogram calls for dynamic polymorphism.

To create an abstract data type, the type should be declared as private and the full type declaration given in the private part:

```
package Airplane_Package is
   type Airplane_Data is tagged private;
   procedure Set_ID(A: in out Airplane_Data; S: in String);
   function Get_ID(A: Airplane_Data) return String;
   procedure Set_Speed(A: in out Airplane_Data; I: in Integer);
   function Get_Speed(A: Airplane_Data) return Integer;
   procedure Set_Altitude(A: in out Airplane_Data; I: in Integer);
   function Get_Altitude(A: Airplane_Data) return Integer;
private
   type Airplane_Data is tagged
      record
         ID: String(1..80);
         Speed: Integer range 0..1000;
         Altitude: Integer range 0..100;
      end record;
end Airplane_Package;
```

The subprograms defined *within* a package specification containing the declaration of a tagged type (along with predefined operations on the type) are called *primitive operations*, and are the subprograms that are inherited upon derivation. Inheritance is done by *extending* a tagged type:

```
with Airplane_Package; use Airplane_Package;
package SST_Package is
   type SST_Data is new Airplane_Data with
      record
         Mach: Float;
      end record;
   procedure Set_Speed(A: in out SST_Data; I: in Integer);
   function Get_Speed(A: SST_Data) return Integer;
end SST_Package;
```

The values of this derived type are a copy of those of the parent type Airplane_Data together with the additional record field Mach. The operations defined for the type are a copy of the primitive operations; these operations may be overridden. Of course other unrelated subprograms may be declared for the derived type.

Ada does not have a special syntax for calling primitive operations:

```
A: Airplane_Data;
```

[5] If necessary, the tag can be read by using an attribute on an object of a tagged type.

```
Set_Speed(A, 100);
```

The object A is syntactically an ordinary parameter and from its type the compiler can deduce which subprogram to call. The parameter is called a *controlling parameter* because it controls which subprogram is chosen. The controlling parameter need not be the first parameter and there may be more than one (provided that they are all of the same type). Contrast this with C++ which uses a special syntax to call a subprogram declared in a class:

C++

```
Airplane_Data a;
a.set_speed(100);
```

The object a is the *distinguished receiver* of the message set_speed. The distinguished receiver is an implicit parameter, in this case denoting that the speed will be set for object a.

Dynamic polymorphism

Before discussing dynamic polymorphism in Ada 95, we have to deal with a difference in terminology between Ada and other object-oriented languages. C++ uses the term class to represent a data type which is used to create instances of the type. Ada 95 continues to use the terms types and objects, even for tagged types and objects which are known in other languages as classes and instances. The word *class* is used to denote the set of all types that are derived from a common ancestor, what we have called a family of classes in C++. The following discussion is best done in the correct Ada 95 terminology; be careful not to confuse the new use of the word class with its use in C++.

With every tagged type T is associated a type called a *class-wide type*, denoted T'Class. All types derived from a type T are *covered* by T'Class. A class-wide type is unconstrained, and an unconstrained object of the type cannot be declared, any more than we can declare an object of an unconstrained array:

```
type Vector is array(Integer range <>) of Float;
V1: Vector;                          -- Illegal, no constraint

type Airplane_Data is tagged record ... end record;
A1: Airplane_Data'Class;             -- Illegal, no constraint
```

However, an object of class-wide type can be declared if it is given an initial value:

```
V2: Vector := (1..20=>0.0);          -- OK, constrained

X2: Airplane_Data;                   -- OK, specific type
X3: SST_Data;                        -- OK, specific type
A2: Airplane_Data'Class := X2;       -- OK, constrained
A3: Airplane_Data'Class := X3;       -- OK, constrained
```

Like an array, once a class-wide object is constrained, its constraint cannot be changed. Class-wide types can be used in the declaration of local variables in a subprogram that takes a parameter of class-wide type, again just like unconstrained arrays:

```
procedure P(S: String; C: in Airplane_Data'Class) is
   Local_String: String := S;
   Local_Airplane: Airplane_Data'Class := C;
begin
   ...
end P;
```

Dynamic polymorphism occurs when an *actual* parameter is of a class-wide type, while the *formal* parameter is of a specific type belonging to the class:

```
with Airplane_Package; use Airplane_Package;
with SST_Package; use SST_Package;
procedure Main is
   procedure Proc(C: in out Airplane_Data'Class; I: in Integer) is
   begin
      Set_Speed(C, I);      -- What type is C ??
   end Proc;

   A: Airplane_Data;
   S: SST_Data;
begin     -- Main
   Proc(A, 500);          -- Call with Airplane_Data
   Proc(S, 1000);         -- Call with SST_Data
end Main;
```

The *actual* parameter C in the call to Set_Speed is of the class-wide type, but there are two versions of Set_Speed whose *formal* parameter is either of the parent type or of the derived type. At run-time the type of C will change from one call to another, so dynamic dispatching is needed to disambiguate the call.

Figure 14.6 will help you understand the role of the formal and actual parameters in dispatching. The call to Set_Speed at the top of the figure has an *actual* parameter which is of class-wide type. This means that only when the subprogram is called, do we know whether the type of the actual parameter is Airplane_Data or SST_Data. However, the procedure declarations shown at the bottom of the figure each have a formal parameter of a specific type. As shown by the arrows, the call must be dispatched according to the type of the actual parameter.

Note that dispatching is only done if needed; if the compiler can resolve the call statically, it will do so. The following calls need no dispatching because the call is with an *actual* parameter of a specific type and not a class-wide type:

```
Set_Speed(A, 500);
Set_Speed(S, 1000);
```

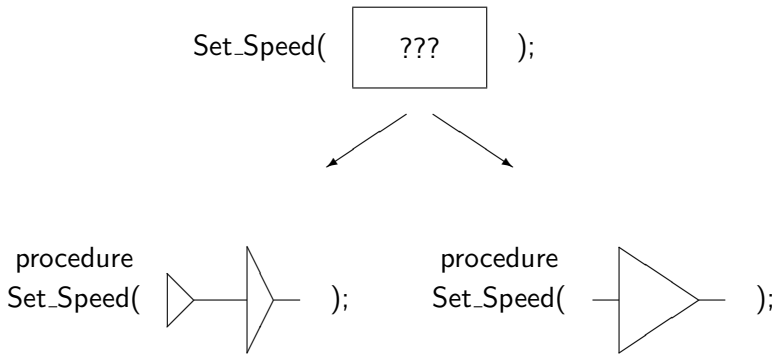

Figure 14.6: Dynamic dispatching in Ada 95

Similarly, if the *formal* parameter is of class-wide type then no dispatching is needed. The calls to Proc are calls to a single unambiguous procedure; the formal parameter is of a class-wide type which matches an actual parameter of any type that is covered by the class. Referring to Figure 14.7, if the declaration of Set_Speed were:

```
procedure Set_Speed(A: in out Airplane'Class: I: in Integer);
```

then any actual parameter of the class will "fit into" the class-wide formal parameter. No dispatching is needed, because every call is to the same subprogram.

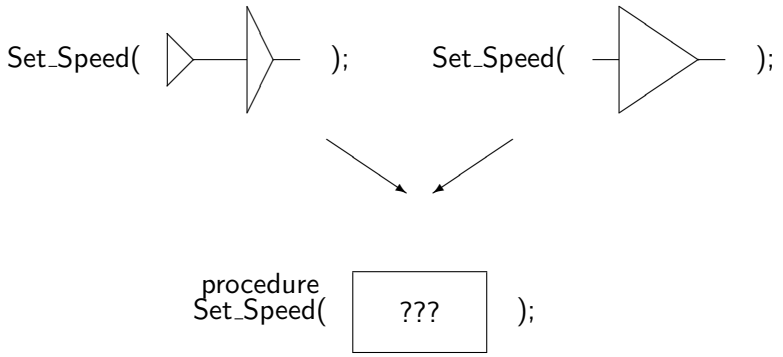

Figure 14.7: Class-wide formal parameter

The designated type of an access type can be a class-wide type. A pointer of this type can designate any object whose type is covered by the class-wide type and dispatching can be done by simply dereferencing the pointer:[6]

```
type Class_Ptr is access Airplane_Data'Class;
Ptr: Class_Ptr := new Airplane_Data;

if (...) then Ptr := new SST_Data; end if;
```

[6] Ada 95 also defines an anonymous access type that can dispatch without dereferencing.

```
Set_Speed(Ptr.all);          -- What type does Ptr point to ??
```

> Dynamic polymorphism in Ada 95 occurs when the actual parameter is of a class-wide type and the formal parameter is of a specific type.

While the run-time implementation of dispatching is similar in Ada 95 and C++, the *conditions* for dispatching are quite different:

- In C++, a subprogram must be declared virtual for dispatching to be done. All indirect calls to the virtual subprogram are dispatching.[7]
- In Ada 95, any inherited subprogram can be overridden and implicitly becomes dispatching. The dispatching is only done as needed if required by a specific call.

The main advantage of the Ada approach is that it is not necessary to specify in advance whether dynamic polymorphism is to be used. This means that the semantic difference between the call to a virtual and to a non-virtual subprogram does not exist. Suppose that Airplane_Data has been defined as tagged but that no derivations have been done. In that case an entire system can be built in which all calls are statically resolved. Later, if derived types are declared, they can use dispatching without the need to modify or recompile the existing code.

14.6 Exercises

1. A software development method called *top-down* programming advocates writing the program in terms of high-level abstract operations and then progressively refining the operations until programming language statements are reached. Compare this method with object-oriented programming.
2. Would you make Aircraft_Data an abstract data type or would you expose the fields of the class?
3. Check that you can inherit from a C++ class or Ada 95 package with a tagged type without recompiling existing code.
4. Write a heterogeneous queue in Ada 95: declare a tagged type Item, specify the queue in terms of Item and then derive from Item for each of Boolean, Integer and Character.
5. Write a heterogeneous queue in C++.
6. Check that in C++ dispatching occurs for a reference parameter but not for an ordinary parameter.
7. In Ada 95, a tagged type can be extended privately:

[7] The compiler may optimize away dynamic dispatching. For example, if the actual parameter is a variable rather than a reference or a pointer, the call can be statically resolved as in Ada 95.

```
with Airplane_Package; use Airplane_Package;
package SST_Package is
   type SST_Data is new Airplane_Data with private;
   procedure Set_Speed(A: in out SST_Data; I: in Integer);
   function Get_Speed(A: SST_Data) return Integer;
private
   ...
end SST_Package;
```

What are the advantages and disadvantages of private extension?

8. Study the machine instructions generated by your Ada 95 or C++ compiler for dynamic polymorphism.