13 Program Decomposition

Both beginning programmers and project managers frequently extrapolate from the ease with which one person can write a program, and assume that developing a software system is just as easy. All you have to do is to assemble a team and let them write the program. However, there is a wide gulf between writing (small) programs and creating (large) software systems, and many systems are delivered late, full of bugs and cost several times the original estimate.[1] The topic called *software engineering* deals with methods for organizing and controlling software development teams, and with notations and tools that support phases of the development process aside from programming. These include requirements specification, software design and software testing.

In this chapter and the next two, we will study programming language constructs that are designed to support the creation of large software systems. There is a clear trade-off: the less support the language offers for the development of large systems, the greater is the need for methods, conventions and notations that are external to the language itself. Since a programming language is of course required, it seems reasonable to integrate support for large systems within the language, and to expect the compiler to automate as much of the process as possible. We software engineers are always willing to automate someone else's job, but we often hesitate before adopting automation into programming languages.

The basic problem is: how should a large software system be decomposed into easily manageable components, that can be developed separately and then assembled into a system in which all the components cooperate with each other as planned? The discussion will begin with elementary "mechanical" methods for decomposing a program, and proceed to modern concepts, such as abstract data types and object-oriented programming, which guide the system designer in creating semantically meaningful components.

Before commencing the discussion, a word to readers who are just beginning their study of programming. The concepts will be presented using small examples that fit into a textbook, and your natural reaction may be that these concepts are just unnecessary "bureaucracy". Be assured that generations of programmers have discovered the hard way that such bureaucracy is essential; the only question is whether it is carefully defined and implemented within a standard language, or invented and imposed by management for each new project.

[1]The first frustrated project manager to write about these problems was Frederick P. Brooks, Jr., in his delightful and instructive book: *The Mythical Man-Month*, Addison-Wesley, 1972.

13.1 Separate compilation

Originally, decomposition of programs was done solely to enable the programmer to compile components separately. Given the power of modern computers and the efficiency of compilers, this reason is less important than it used to be, but it is important to study separate compilation because the features that support it are often the same ones used for decomposition of a program into logical components. Even in very large systems which cannot be constructed without separate compilation, the decomposition into components is determined during program design without reference to compilation time. Since such components are usually relative small, the limiting factor in the time it takes to modify a program is usually link time rather than compile time.

Separate compilation in Fortran

When Fortran was developed, programs were fed into a computer using punched cards and there were no disks or program libraries as we know them today.[2] The unit of compilation in Fortran is identical with the unit of execution, namely the subprogram, called *subroutine*. Each subroutine is compiled, not only separately, but *independently*; no information is stored from one compilation to use in a subsequent one.

This means that absolutely no checking is done between the formal parameters and the actual parameters. You can supply a floating-point value for an integer parameter. Even worse, an array is passed as a pointer to the first component and the called subroutine has no way of knowing the size of the array or even the type of the component. It is even possible for a subroutine to attempt to access a non-existent actual parameter. In other words, matching formal and actual parameters is the task of the programmer; he/she must supply correct declarations of the parameter types and sizes both in the calling and called subroutines.

Since each subroutine is independently compiled, there is no way to directly share global data declarations. Instead, *common blocks* are defined:

```
subroutine S1
common /block1/ distance(100), speed(100), time(100)
real distance, speed, time
. . .
end
```

This declaration requests the allocation of 300 memory locations for floating-point values. All declarations for the same block are allocated to the same memory locations, so if another subroutine declares:

```
subroutine S2
common /block1/ speed(200), time(200), distance(200)
integer speed, time, distance
. . .
```

[2]When I wrote my first program, I was handed a deck of *binary* punched cards and told that it was the plotter driver!

```
end
```

the two subroutines are using different names and different types to access the same memory! Mapping of common blocks is done by memory *location* and not by variable *name*. If a real variable is allocated twice the memory as an integer variable, speed(80) in the subroutine S2 is accessing the same memory as half of the variable distance(40) in S1. The effect is similar to the undisciplined use of union types in C or variant records in Pascal.

Independent compilation and common blocks are unlikely to cause problems for a single programmer writing a small program, but are extremely likely to cause problems in a ten-person team; meetings or inspections must be held to ensure that interfaces are correctly implemented. A partial solution is to use *include*-files especially for common blocks, but you still have to check that you are using the latest version of the include file, and to make sure that some clever programmer doesn't ignore the declarations in the file.

Separate compilation in C

C is unusual among programming languages in that the concept of a source code *file* appears in the language definition and is significant in terms of scope and visibility of identifiers. C encourages separate compilation to such an extent that by default every subprogram and every global variable can be accessed from anywhere in the program.

First some terminology: a *declaration* introduces a name into a program:

```
void proc(void);
```

A name may have many (identical) declarations, but exactly one of them will also be a *definition* which creates an entity of this name: allocates memory for variables or gives the implementation of a subprogram.

The following file contains a main program, as well as a global variable *definition* and a function *declaration*, both of which have external linkage by default:

```
    /* File main.c */
int global;                 /* External by default */
int func(int);              /* External by default */

int main(void)
{
    global = 4;
    return func(global);
}
```

In a separate file, the definition (implementation) of the function is given; the variable global is declared again to enable the function to access it:

```
    /* File func.c */
extern int global;          /* External, declaration only */

int func(int parm)
{
    return parm + global;
}
```

Note that another declaration of func is not needed, because the definition of the function in this file also serves as a declaration and by default it has external linkage. However, in order for func to access the global variable a declaration is needed, and the specifier extern must be used. If extern is not used, the declaration of global will be taken as a second definition of the variable. A linker error will occur since it is illegal to have two definitions of the same global variable in a program.

Compilation in C is independent in the sense that the result of one compilation is not stored for use by another. If a member of your team accidentally writes:

```
    /* File func.c */
extern float global;        /* External, declaration only */

int func(int parm)          /* External by default */
{
    return parm + global;
}
```

the program can still be compiled and linked, and it is only at run-time that an error occurs. On my computer, the integer value 4 assigned to global in main appears in the file func.c to be a very small floating-point number; when converted back to an integer, it becomes zero and the function returns 4 rather than 8.

As in Fortran a partial solution is to use include files so that the same declarations are accessed in all files. Both an extern declaration for a function or variable and a definition can appear within the same computation. Thus we put all the external declarations in one or more include files, while at most one ".c" file will contain the single definition of each function or variable:

```
    /* File main.h */
extern int global;          /* Declaration only */

    /* File func.h */
extern int func(int parm);  /* Declaration only */

    /* File main.c */
#include "main.h"
#include "func.h"
int global;                 /* Definition */
int main(void)
```

```
{
    return func(global) + 7;
}

    /* File func.c */
#include "main.h"
#include "func.h"
int func(int parm)           /* Definition */
{
    return parm + global;
}
```

static specifier

Anticipating the discussion in the next section, we now show how features of the C decomposition can be used to emulate the module construction of other languages. In a file containing dozens of global variables and subprogram definitions, usually only a few of them need to be accessed from outside the file. Each definition which is *not* used externally should be preceded by the specifier static which indicates to the compiler that the declared variable or subprogram is known only within the file:

```
static int g1;                /* Global only in this file */
       int g2;                /* Global to all files */
static int f1(int i) { ...};  /* Global only in this file */
int    f2(int i) { ...};      /* Global to all files */
```

This is known as *file scope* and is used where other languages would have module scope. It would be better of course if the default specifier were static rather than extern; however, it is not difficult to get into the habit of prefixing every global declaration with static.

A source of confusion in C is the fact that static has another meaning, namely that it specifies that the lifetime of a variable is the entire execution of the program. As we discussed in Section 7.4, local variables within a procedure have a lifetime limited to a single invocation of the procedure. Global variables, however, have *static* lifetime, that is they are allocated when the program begins and are not deallocated until the program terminates. Static lifetime is the normal behavior we expect from global variables; in fact, global variables declared with extern also have static lifetime!

The static specifier can also be used on local variables to request static lifetime:

```
void proc(void)
{
    static bool first_time = true;
    if (first_time) {
        /* Statements executed the first time that proc is called */
        first_time = false;
        }
        ...
    }
```

To summarize: all global variables and subprograms in a file should be declared `static`, unless it is explicitly required that they be accessible outside the file. In this case, they should be defined in one file without any specifier, and exported by declaring them in an include file with the `extern` specifier.

13.2 Why are modules needed?

In the previous section we approached the decomposition of programs from a purely mechanical standpoint, namely the desire to separately edit and compile parts of the program in separate files. Starting in this section, we discuss decomposition of programs into components that are created to express the design of the program, and only incidentally to allow separate compilation. But first let us ask why is such decomposition necessary?

You may have been taught that the human brain is only capable of dealing with a small amount of material at any one time. In terms of programming, this is usually expressed by requiring that a single subprogram be no longer than one "page".[3] The subprogram is considered to be a conceptual unit: a sequence of statements performing a certain function. If the subprogram is small enough, say 25 to 100 lines, we will be able to completely and easily understand the relationships among the statements that compose the subprogram.

But to understand an entire program, we must understand the relationships among the subprograms that compose it. By analogy, it ought to be possible to understand programs composed of 25 to 100 subprograms, which totals 625 to 10,000 lines. This size program is relatively small compared to industrial and commercial software systems that contain 100,000, if not one million, lines. Experience shows that 10,000 lines is probably an upper limit on the size of a monolithic program, and that a new structuring mechanism is needed in order to build and maintain larger software systems.

The standard term for the mechanism used to structure large programs is *module*, though the two languages that we are concentrating on use other terms: *packages* in Ada and *classes* in C++. Standard Pascal does not define any method for separate compilation or for decomposing programs. For example, the first Pascal compiler was a *single* program containing over 8,000 lines of Pascal code. Rather than modify Pascal, Wirth developed a new (though similar) language called Modula which takes its name from the central concept of modules. Unfortunately, many vendors extended Pascal with incompatible module constructs, so Pascal is not suitable for writing portable software. Because modules are so important to software development, we choose to focus the discussion on Ada which has an elegant module construct called *packages*.

[3] This of course dates from the days when programs could be read only if printed out on paper, and a single page could hold about 50 lines. Today we would restrict a subprogram to about half that size so that it could be completely displayed on the screen (unless you have a workstation with a large screen!).

13.3 Packages in Ada

The basic concept underlying modules in general and Ada packages in particular is that computational resources like data and subprograms should be *encapsulated*[4] into a unit. Access to components of the unit is permitted only according to an explicit interface specification. Figure 13.1 shows a graphical notation (called *Booch-Buhr diagrams*) used in Ada designs. The large rectan-

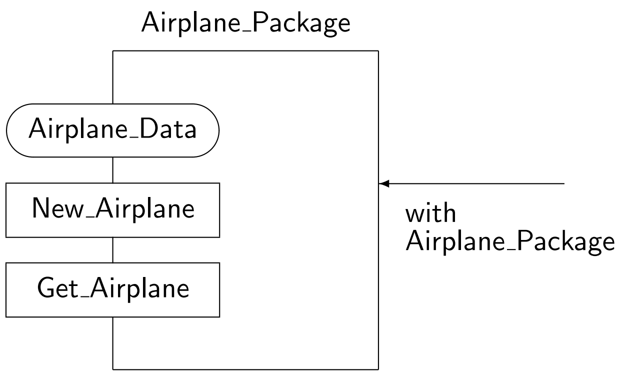

Figure 13.1: Diagram of a package

gle denotes the package Airplane_Package containing hidden computational resources, while the small rectangles are meant to look like windows that allow the user of the package access to the hidden resources. The oval rectangle denotes that a type is exported; the other two are subprograms. From each unit that uses the resources of the package, an arrow is drawn which points to the package.

Declaring a package

A package is composed of two parts: a *specification* and a *body*. The body encapsulates the computational resources, while the specification defines the interface to these resources. The package in the following example is intended to represent a component of an air-traffic control system[5] that stores a description of all the airplanes in the airspace being controlled. The package specification declares a type and two interface subprograms:

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

[4]As in the word “capsule”.

[5]TRACON, from Wesson International, is a simulation of such a system on a PC and will give you an idea of the type of program we are discussing.

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

The package specification contains only procedure declarations—terminated by a semicolon, not procedure bodies—which are denoted by the reserved word is and a block. The declaration serves only as a specification of a computational resource that the package provides.

In the package body, all resources that were promised must be supplied. In particular, for every subprogram declaration, a subprogram body must exist with exactly the same declaration:

```
package body Airplane_Package is
    Airplanes: array(1..1000) of Airplane_Data;
    Current_Airplanes: Integer range 0..Airplanes'Last;

    function Find_Empty_Entry return Integer is
    begin
        . . .
    end Find_Empty_Entry;

    procedure New_Airplane(Data: in Airplane_Data; I: out Integer) is
        Index: Integer := Find_Empty_Entry;
    begin
        Airplanes(Index) := Data;
        I := Index;
    end New_Airplane;

    procedure Get_Airplane(I: in Integer; Data: out Airplane_Data) is
    begin
        Data := Airplanes(I);
    end Get_Airplane;
end Airplane_Package;
```

What has this accomplished? The data structure used to hold the airplanes, here a fixed-size array, is encapsulated within the package body. The rule in Ada is that a change in a package body only, does not require modification either of the package specification or of any other program component that uses the package. Furthermore, it is not even necessary to recompile the specification or any component that uses the package. For example, if you subsequently need to replace the array by a linked list, no other component of the system need be modified *provided* that the interface, as listed in the package specification, is not modified:

```
package body Airplane_Package is
    type Node;
    type Ptr is access Node;
    type Node is
```

```
        record
            Info: Airplane_Data;
            Next: Ptr;
        end record;

    Head: Ptr;              -- Head of linked list

    procedure New_Airplane(Data: in Airplane_Data; I: out Integer) is
    begin
        . . .               -- New implementation
    end New_Airplane;

    procedure Get_Airplane(I: in Integer; Data: out Airplane_Data) is
    begin
        . . .               -- New implementation
    end Get_Airplane;
end Airplane_Package;
```

Encapsulation is not just for convenience but also for reliability. The user of the package is not allowed to directly access data or internal subprograms (like Find_Empty_Entry) of the package body. Thus no other programmer on the team can accidentally (or maliciously) modify the data structure in a manner that was not intended. A bug in the implementation of a package is necessarily within the code of the package body, and not the result of some code written by a member of the team not responsible for the package.

Both the package specification and the package body are distinct units and can be separately compiled. However, they are considered to be a single scope in terms of declarations, so for example, the type Airplane_Data is known within the body. This means of course that the specification must be compiled before the body. Unlike C, there is no concept of "file", and declarations in Ada only exist within a unit such as a subprogram or package. Several compilation units can be in the same file, though it is usually more convenient to write each one in a separate file.

The convention for writing C programs suggested in the previous section attempts to imitate the encapsulation supplied by Ada packages. The include files containing the external declarations correspond to the package specifications, and by writing static on all global variables and subprograms in a file the effect of a package body is achieved. Of course this is just "bureaucracy" and can easily be circumvented, but it is a good way of structuring C programs.

Using a package

An Ada program (or another package) can access the computational resources of a package by writing a *context clause* before the first line of the program:

```
with Airplane_Package;
procedure Air_Traffic_Control is
```

```
    A: Airplane_Package.Airplane_Data;
    Index: Integer;
begin
    while ... loop
        A := ...;               -- Create record
        Airplane_Package.New_Airplane(A, Index);
                                -- Store in data structure
    end loop;
end Air_Traffic_Control;
```

The with-clause tells the compiler that this program is to be compiled in an environment that includes all the declarations of Airplane_Package. The syntax for naming components of a package is like the syntax for selecting record components. Since each package must have a distinct name, components of a package can have identical names and no conflict will arise. This means that the management of the *name space*, the set of names, in a software project is simplified, and need only be controlled at the level of package names. Compare this with C where an identifier that is exported from a file is visible to all other files, so it is not sufficient just to ensure that file names are distinct.

The with-clause adds the expanded names to the name space of a compilation; it is also possible to include a use-clause[6] to "open" the name space and allow direct naming of components appearing in a specification:

```
with Airplane_Package;
use Airplane_Package;
procedure Air_Traffic_Control is
    A: Airplane_Data;                -- Directly visible
    Index: Integer; begin
    New_Airplane(A, Index);          -- Directly visible
end Air_Traffic_Control;
```

One difficulty with use-clauses is that you may encounter ambiguity if use-clauses for two packages expose the same name, or if a local declaration exists with the same name as that in a package. The language rules specify what the compiler's response must be in cases of ambiguity.

More importantly, however, a unit that with's and then use's a lot of packages can become practically impossible to read. A name like Put_Element could come from almost any package, while the location of the definition of Airplane_Package.Put_Element is obvious. The situation is similar to that of a C program with many include files: you simply have no easy way of locating declarations, and the only solution is an external software tool or bureaucratic naming conventions.

Ada programmers should take advantage of the self-documenting aspect of with'ed units, and employ use-clauses only in small segments of a program where the intent is clear and the full

[6]The use-clause in Ada is similar to the with-statement in Pascal which adds the field names of a record to the name space. Note also that Turbo Pascal (a trademark of Borland International, Inc.) contains uses-clauses which access declarations from other units similarly to the Ada with-clause. Don't get confused!

notation would be extremely tedious. Fortunately, it is possible to put a use-clause within a local procedure:

```
procedure Check_for_Collision is
   use Airplane_Package;
   A1: Airplane_Data;
begin
   Get_Airplane(1, A1);
end Check_for_Collision;
```

In most programming languages the *importing* unit automatically obtains all the public resources of the imported unit. Some languages like Modula allow the importing unit to specify exactly which resources it needs. This method can avoid the name space overload caused by the inclusive nature of use-clauses in Ada.

Compilation order

with-clauses impose a natural order on compilation: a package specification must be compiled *before* its body, and *before* any unit that with's it. However, the ordering is partial, that is, there is no required compilation order between the package body and the units that use the package. You can fix a bug either in the package body or in the using unit without recompiling the other, but modifying a package specification requires recompilation both of the body and of all using units. In a very large project, modifications of package specifications should be avoided because they can trigger an avalanche of recompilations: P1 is used by P2 which is used by P3, and so on.

The fact that the compilation of one unit requires the results of the compilation of another means that an Ada compiler must contain a *library* to store compilations. The library may simply be a directory containing internally generated files, or it may be a sophisticated database. Regardless of the method used, a library manager is a central component of an Ada implementation, not just an optional software tool. The Ada library manager enforces the rule that changing a package specification requires the recompilation of the body and the using units. Thus an Ada compiler already includes a make tool that in other programming environments is an optional utility rather than part of the language support.

13.4 Abstract data types in Ada

Airplane_Package is an *abstract data object*. It is *abstract* because the user of the package does not know if the database of airplanes is implemented as an array, a list or a tree. The only access to the database is through the interface procedures in the package specification, which enable the user to abstractly create and retrieve a value of type Airplane_Data without knowledge of how it is stored.

The package is a *data object* because it actually contains data: the array and any other variables declared in the package body. It is correct to look upon Airplane_Package as a glorified variable:

memory must be allocated for it and there are certain operations which can modify its value. It is not a first-class object because it does not have all the privileges of an ordinary variable: you cannot assign to a package or pass a package as a parameter.

Suppose now that we need *two* such databases; perhaps one for the simulated air-traffic control console, and one for a simulation scenario manager that creates and initializes new airplanes. It would be possible to write two packages, each with slightly different names, or to write a generic package and instantiate it twice, but these are very limited solutions. What we would really like to do is to declare as many such objects as we need, just like we declare integers. In other words, we want to be able to construct an *abstract data type (ADT)* which is just like an abstract data object, except that it does not contain any “variables”. Instead, like other types an ADT will specify a set of values and a set of operations on these values, and we leave it to other program components to actually declare variables of the type.

An ADT in Ada is a package which contains only constant, type and subprogram declarations. The package specification includes a type declaration so that other units can declare one or more Airplanes:

```
package Airplane_Package is
    type Airplane_Data is ... end record;
    type Airplanes is
        record
            Database: array(1..1000) of Airplane_Data;
            Current_Airplanes: Integer 0..Database'Last;
        end record;
    procedure New_Airplane(
        A: in out Airplanes; Data: in Airplane_Data; I: out Integer);
    procedure Get_Airplane(
        A: in out Airplanes; I: in Integer; Data: out Airplane_Data);
end Airplane_Package;
```

The package body is as before except that there are no global variables in the body:

```
package body Airplane_Package is
    function Find_Empty_Entry ... ;
    procedure New_Airplane ...;
    procedure Get_Airplane ...;
end Airplane_Package;
```

The program that uses the package can now declare one or more variables of the type supplied by the package. In fact the type is an ordinary type and can be used in further type definitions and as the type of a parameter:

```
with Airplane_Package;
procedure Air_Traffic_Control is
    Airplane: Airplane_Package.Airplanes;
```

```
            -- Variable of the ADT
        type Ptr is access Airplane_Package.Airplanes;
            -- Type with component of the ADT
        procedure Display(Parm: in Airplane_Package.Airplanes);
            -- Parameter of the ADT
        A: Airplane_Package.Airplane_Data;
        Index: Integer;
    begin
        A := . . . ;
        Airplane_Package.New_Airplane(Airplane, A, Index);
        Display(Airplane);
    end Air_Traffic_Control;
```

There is a price to be paid for using ADT's instead of abstract data objects: since there is no longer *one* implicit object in the package body, every interface procedure must contain an extra parameter that explicitly informs the subprogram which object to process.

Now you are supposed to complain: what about "abstract"? Since the type Airplanes is now declared in the package specification, we have lost all the abstraction; it is no longer possible to change the data structure without potentially invalidating every other unit that uses the package. Furthermore, some team member can secretly ignore the interface procedures and write a "better" interface. We have to find a solution where the name of the type is in the specification so that it can be used, but the details of the implementation are encapsulated—something like the following:

```
    package Airplane_Package is
        type Airplane_Data is . . . end record;
        type Airplanes;          -- Incomplete type declaration
        . . .
    end Airplane_Package;

    package body Airplane_Package is
        type Airplanes is        -- Complete type declaration
            record
                Database: array(1..1000) of Airplane_Data;
                Current_Airplanes: Integer 0..Database'Last;
            end record;
        . . .
    end Airplane_Package;
```

Take a few minutes to analyze this proposal yourself before continuing.

As far as the package is concerned, there is no problem with these declarations, because the specification and the body form a single declarative region. The problems begin when we try to use the package:

```
    with Airplane_Package;
```

```
procedure Air_Traffic_Control is
    Airplane_1: Airplane_Package.Airplanes;
    Airplane_2: Airplane_Package.Airplanes;
    ...
end Air_Traffic_Control;
```

Ada is designed so that compilation of a package specification is sufficient to enable compilation of any unit using the package. If fact, the package body need not even exist when the using unit is compiled. But to compile the above program, the compiler must know how much memory to allocate for Airplane_1 and Airplane_2; similarly, if this variable is used in an expression or passed as a parameter, the compiler must know the size of the variable. Thus if the representation of an ADT is encapsulated in the package body, it will be impossible to compile the program.

Private types

Since we are dealing with practical programming languages which have to be compiled, there is no choice but to return the complete type specification to the package specification. To achieve abstraction, a combination of self-delusion and a language rule is used:

```
package Airplane_Package is
    type Airplane_Data is ... end record;
    type Airplanes is private;
                             -- Details to be supplied later
    procedure New_Airplane(Data: in Airplane_Data; I: out Integer);
    procedure Get_Airplane(I: in Integer; Data: out Airplane_Data);
private
    type Airplanes is        -- Complete type declaration
        record
            Database: array(1..1000) of Airplane_Data;
            Current_Airplanes: Integer 0..Database'Last;
        end record;
end Airplane_Package;
```

The type itself is initially declared as a *private type*, while the complete type definition is written in a special section of the package specification that is introduced by the keyword private. The data type is abstract because the compiler enforces a rule that units with'ing the package are not allowed to access information written in the private part. They are only permitted to access the private data type through the interface subprograms in the public part of the specification; these subprograms are implemented in the body which can access the private part. Since the code of the using units does not depend upon the private part, it is possible to modify declarations in the private part without invalidating the using units; of course, recompilation will be necessary because a change in the private part could require a different amount of memory to be allocated.

Since you cannot make explicit use of the information in the private part, you should "pretend" that you cannot even see it. For example, you should not make a special effort to write algorithms that are extremely efficient if a private type is implemented as an array but not as a list, because the project team leader may eventually modify the implementation.

Limited types

An object (variable or constant) of a private type can be declared, and the operations of assignment and equality checking can be performed on objects of private type, since these operations are done bitwise without knowledge of the internal structure. There is, however, a conceptual problem with allowing assignment and equality. Suppose that an implementation is changed from an array to a pointer:

```
package Airplane_Package is
    type Airplanes is private;
    . . .
private
    type Airplanes_Info is
        record
            Database: array(1..1000) of Airplane_Data;
            Current_Airplanes: Integer 0..Database'Last;
        end record;
    type Airplanes is access Airplanes_Info;
end Airplane_Package;
```

We promised that modifying the private part does not require a modification of the using units, but this is not true here because assignment is that of the pointers and not the designated objects:

```
with Airplane_Package;
procedure Air_Traffic_Control is
    Airplane_1: Airplane_Package.Airplanes;
    Airplane_2: Airplane_Package.Airplanes;
begin
    Airplane_1 := Airplane_2;- Pointer assignment
end Air_Traffic_Control;
```

If assignment and equality are not meaningful (for example comparing two arrays that implement databases), Ada allows you to further declare a private type as limited. Objects of limited types cannot be assigned or compared, but you can explicitly write your own versions of these operations. This will solve the problem just described; in the transformation between the two implementations, a modification in the package body of the explicit code for assignment and equality can be done to ensure that these operations are still meaningful. Non-limited private types should be restricted to "small" objects that are not likely to undergo modifications other than adding or changing a field in a record.

Note that if a private type is implemented by a pointer, it doesn't really matter what the designated type is under the assumption that all pointers are represented the same way. Ada does in fact make this assumption and thus the designated type can be defined in the package body. Now, changing the data structure does not even require recompilation of the with'ing units, at the cost of indirect access to each object of the type:

```
package Airplane_Package is
   type Airplanes is private;
   ...
private
   type Airplanes_Info;      -- Incomplete type declaration
   type Airplanes is access Airplanes_Info;
end Airplane_Package;

package body Airplane_Package is
   type Airplanes_Info is   -- Completion in the body
      record
         Database: array(1..1000) of Airplane_Data;
         Current_Airplanes: Integer 0..Database'Last;
      end record;
end Airplane_Package;
```

ADT's are a powerful method of structuring programs because of the clear separation between specification and implementation:

- Using ADT's, it becomes possible to make major modifications of individual program components reliably, without causing bugs in unrelated parts of the program.
- ADT's can be used as a management tool: the project architect designs the interfaces and each team member is given one or more ADT's to implement.
- It is possible to perform testing and partial integration by supplying degenerate implementations of missing package bodies.

In Chapter 14 we will expand on the role of ADT's as the basis of object-oriented programming.

13.5 How to write modules in C++

C++ is an extension of C and thus the concept of file as a unit for structuring programs still exists. The most important extension relative to C is the introduction of *classes* which implement abstract data types directly, unlike Ada which uses a combination of the two concepts: package and private data type. In the next chapter we will discuss object-oriented programming which is based on classes; in this section we will explain the basic concepts of classes and show how they can be used to define modules.

A class is like a package specification that declares one or more private types:

```
class Airplanes {
public:
   struct Airplane_Data {
      char id[80];
      int speed;
      int altitude;
   };
   void new_airplane(const Airplane_Data & a, int & i);
   void get_airplane(int i, Airplane_Data & a) const;

private:
   Airplane_Data database[1000];
   int current_airplanes;
   int find_empty_entry( );
};
```

Note that the name of the class, that is the name of the type, also serves as the name of the unit of encapsulation; there is no separate module name. A class has a public part and a private part. By default, components of a class are private so the public specifier is needed before the public part. In fact, by using public and private specifiers, the public and private parts can be interspersed, unlike Ada which requires that there be exactly one list of declarations for each:

```
class C {
public:
   ...
private:
   ...
public:
   ...
private:
   ...
};
```

The declarations in the public part are accessible by any unit using this class, while the declarations in the private part are only accessible within the class. As a further means of control, the const specifier on get_airplane means that the subprogram will not modify any of the data within an object of the class. Such subprograms are called *inspectors*.

Since a class is a type, objects (constants and variables) of the class, called *instances*, can be declared:

```
Airplanes Airplane;            // Instance of class Airplanes

int index;
Airplanes::Airplane_Data a;
Airplane.new_airplane(a, index);// Call a subprogram on an instance
```

Similarly, a parameter type can be a class. Each instance will be allocated memory for all variables declared in the class, just as a variable of record type is allocated memory for all fields.

The syntax of the subprogram call is different from the Ada syntax because of a difference in the basic concept. The Ada call:

```
Airplane_Package.New_Airplane(Airplane, A, Index);
```

looks upon the package as supplying a resource, the procedure New_Airplane, which must be supplied with a specific object Airplane. C++ considers the object Airplane to be an instance of the class Airplanes, and if you send the object the *message* new_airplane, the corresponding procedure will be executed for this object.

Note that even subprograms like find_empty_entry which are only needed internally to implement the class, are declared in the class definition. C++ has nothing similar to a package body, which is a unit that encapsulates the implementation of the interface and other subprograms. Of course, the internal subprogram is not available for access by other units because it is declared within the private part. The problem in C++ is that if it is necessary to modify the declaration of find_empty_entry or to add another private subprogram, it will be necessary to recompile all units of the program which use this class; in Ada, a modification to the package body does not affect the rest of the program. To achieve true separation of interface and implementation of C++, you must declare the interface as an abstract class and then derive a concrete class which contains the implementation (see Section 15.1).

Where are the subprograms of a class implemented? The answer is that they can be implemented anywhere, in particular, in a separate file which accesses the class definition through an include file. A *scope resolution operator* "::" identifies each subprogram as belonging to a specific class:

```
   // Some file
#include "Airplanes.h"      // Contains class declaration

void Airplanes::new_airplane(const Airplane_Data & a, int & i)
{
   ...
}
void Airplanes::get_airplane(int i, Airplane_Data & a) const
{
   ...
}
int Airplanes::find_empty_entry( )
{
   ...
}
```

Note that the internal subprogram find_empty_entry must be declared within the (private part) of the class so that it can access private data.

Namespaces

One of the last additions to the definition of C++ was the *namespace* construct which enables the programmer to limit the scope of otherwise global entities in a manner similar to that of an Ada package. A clause similar to the Ada use-clause opens the namespace:

```
namespace N1{
   void proc( );              // Procedure in namespace
};
namespace N2{
   void proc( );              // Different procedure
};

N1::proc( );                  // Scope resolution operator to access

using namespace N1;
proc( );                      // OK
using namespace N2;
proc( );                      // Now ambiguous
```

Unfortunately, the C++ language does not define a library mechanism: class declarations must still be shared through include files. A development team must establish procedures for updating include files, preferably using software tools to warn the team members if two compilations are not using the same version of an include file.

13.6 Exercises

1. Write a main program in C that calls an external function f with an int parameter; in another file, write a function f that has a float parameter which it prints. Compile, link and run the program. What does it print? Try to compile, link and run the *same* program in C++.

2. Write a program to implement an abstract data type for a queue, and a main program that declares and uses several queues. The queue should be implemented as an array that is declared in the private part of an Ada package or C++ class. Then change the implementation to a linked list; the main program should be run unaltered.

3. What happens if you try to assign one queue to another? Solve the problem by using a limited private type in Ada, or a *copy-constructor* in C++.

4. In C and C++, a subprogram declaration need not have parameter names:

   ```
   int func(int, float, char*);
   ```

   C

   Why is this so? Should you use parameter names anyway? Why does Ada require parameter names in a package specification?

5. Ada has a construct for separate compilation that is independent of the package construct:

Ada

```
procedure Main is
    Global: Integer;
    procedure R is separate;-- Separately compiled procedure
end Main;

separate(Main)              --Another file
procedure R is
begin
    Global := 4;            -- Ordinary scope rules
end R;
```

The fact that a local package or procedure body is separately compiled does not change its scope and visibility. How can this be implemented? Does a change in a separate unit require recompilation of the parent unit? Why? Conversely, how does a change in the parent unit affect the separate unit?

6. Since a separate unit is a compilation unit, it can have its own context clause:

Ada

```
with Text_IO;
separate(Main)
procedure R is
    ...
end R;
```

Why would this be done?

7. The following Ada program does not compile; why?

Ada

```
package P is
    type T is (A, B, C, D);
end P;

with P;
procedure Main is
    X: P.T;
begin
    if X = P.A then ... end if;
end Main;
```

There are four ways to solve the problem; what are the advantages and disadvantages of each: (a) use-clause, (b) prefix notation, (c) renames, (d) use type-clause in Ada 95?