Introduction to Systematic Programming
Unit 21 - Blocks, Renaming, Scope, Visibility, Overloading

21.1 Blocks

In this course we have referred to the **life-time** variables and constants. Life-time is a dynamic concept, that is a variable or constant is created at the point of execution of its declaration, and ceases to exist at some later point in the execution of the program. So far we have seen variables declared as:

a) Main program variables, defined at the head of the main program, whose life-time is for the duration of execution of the whole program.
b) Local variables, defined in subprograms (procedures or functions), whose life-time is for the duration of execution of the subprogram.
c) Variables defined in packages (in either the specification or body), whose life-time is for the duration of execution of the whole program.

There are sometimes circumstances where we need a temporary variable in a particular sequence of program steps, but not elsewhere in the program. Ada provides a mechanism to achieve our intended effect through the use of a **block** step, which has the basic structure:

```
DECLARE
   Declarations;
BEGIN
   Step(s)_to_be_performed;
END;
```

This means that any variables declared in the *Declarations* part will have a life-time which is only for the duration of execution of the body of the block, i.e. the *Step(s)_to_be_performed* part. The variables are created just before execution of the *Step(s)_to_be_performed*, and then after the `END`, such variables cease to exist. There are no restrictions on what may be defined in the *Declaration* part of a block step and we could include the definition of constants and (very rarely) types and subtypes or even subprograms.

For example, if in a particular sequence of program steps it is necessary to interchange the values of the two `Integer` variables `First` and `Second`, we may write:

```
IF First < Second THEN
   DECLARE
      Temp : Integer := First;
   BEGIN
      First := Second;
      Second := Temp;
   END;
END IF;
```

where the temporary variable `Temp`, is only defined to exist for the duration of execution of the block within which it is declared, i.e. for the fragment of program where the values of `First` and `Second` are interchanged. This also has the advantage that `Temp` is declared near to the place it is used rather than at the head of the (sub)program.

The use of a block to localise the declaration of variables to the fragment of program where they are used may not always improve the clarity of the resulting program; in fact, the excessive use of blocks is likely to make a program more obscure as variable declarations are scattered throughout the code. If you write programs with a high degree of procedurisation, so that variables can be declared local to the subprograms where they are used, there will be relatively few circumstances where program clarity is improved by use of a block rather than a subprogram. Just occasionally when a relatively short piece of code is closely coupled to its surroundings, the use of a block may be preferable to a subprogram as the latter would require a large number of parameters - something we have previously suggested is poor programming style.

21.2 Renaming

It is possible to give a new name to certain Ada entities using a **renaming declaration**. The following entities can be renamed:

variables	array elements	record fields	array slices
constants	packages	procedures	functions

A renaming declaration may placed in the declarative region of a main program, a package or subprogram along with other declarations and definitions.

In Unit 16 we defined a rather complex record type for holding information about a new-born baby:

```
SUBTYPE NameString IS String(1 .. 15);

TYPE FullName IS RECORD
                    Forename   : NameString;
                    SecondName : NameString;
                    Surname    : NameString;
                 END RECORD;

TYPE BabyRecord IS RECORD
                       Gender      : GenderType;
                       TimeOfBirth : ClockTime;
                       DateOfBirth : DateType;
                       BirthWeight : WeightType;
                       MothersName : FullName;
                    END RECORD;

Baby : BabyRecord;
```

Then to refer to the mother's surname and initial we could write

```
Baby.MothersName.Surname
```
and
```
Baby.MothersName.Forename(1)
```

However these names are rather long and this could be inconvenient if these quantities are going to be used frequently in a particular section of program code. To overcome this problem we could write (in an appropriate declaration section):

```
LastName : NameString RENAMES Baby.MothersName.Surname;
Initial : Character RENAMES Baby.MothersName.Forename(1);
```

Now we could use the new abbreviated names for the mother's surname and initial, for example:

```
Put(Initial);            -- output Baby.MothersName.Forename(1)
Put(". ");
Put_Line(LastName);      -- output Baby.MothersName.Surname
```

Note that in the renaming declaration we give the new name and its type (just as if we were declaring a new variable) followed by the keyword `RENAMES` and then the entity being renamed. The type specified in the renaming must, of course, be consistent with that of the entity being renamed.

When renaming array elements it is often necessary to introduce a block. Consider the following example in which an array of records (of the type `BabyRecord` discussed above) is searched for a certain surname and then the contents of that record are output:

```
TYPE BabyList IS ARRAY(1 ..200) OF BabyRecord;
Deliveries : BabyList;
Index : Positive := 1;
Found : Boolean := False;
Name : NameString;
.......
WHILE Index <= BabyList'Last AND NOT Found LOOP
   DECLARE
      Baby : BabyRecord RENAMES Deliveries(Index);
      LastName : NameString RENAMES Baby.MothersName.Surname;
   BEGIN
      IF LastName = Name THEN
         Found := True;
         -- output details of this baby
         Put("Baby ");
         Put_Line(LastName);
         -- output weight
         Put(BirthWeight.Pounds);
         Put(BirthWeight.Ounces);
         .......  -- output remaining details
      ELSE
         Index := Index + 1;
      END IF;
```

```
    END;
  END LOOP;
```

Note that placing the renaming declarations at the end of the main declarative region would not work as then `Baby` would always refer to the record `Deliveries(1)` (as `Index` has the value `1` at the time the declaration is made). The renaming must be placed inside the loop so that the renaming refers to a different record each time through the loop; hence a block is necessary.

Another fairly common use of renaming is to introduce shorter name for a quantity imported from a package to use in place of its fully qualified name. If used judiciously this technique provides the convenience of using short names for a few items exported by a package without the need to make all the items exported by the package directly visible with a `USE` context clause.

```
PACKAGE ElemFns RENAMES Ada.Numerics.Elementary_Functions;
```

Now we could use short fully qualified names such as `ElemFns.Sin` for the facilities provided by the package instead of monstrosities like `Ada.Numerics.Elementary_Functions.Sin`.

In **subprogram renamings** the number, order, mode and type of the parameters (and in the case of a function the type of the value returned) in the renaming must agree with those of the renamed subprogram. This is called **mode conformance** of the new and renamed subprograms. For example if we wanted to make extensive use of the function `Log` (natural logarithm) from the package `Ada.Numerics.Elementary_Functions` (but did not require any other features of the package) we might write

```
FUNCTION Ln(X : Float) RETURN Float
         RENAMES Ada.Numerics.Elementary_Functions.Log;
```

Of course, if we used the above renamings then at the head of our program we would need the context clause:

```
WITH Ada.Numerics.Elementary_Functions;
```

When renaming subprograms none of the following are features are normally taken into account[1]: the names of the formal parameters, their subtypes and the default values (if any) of the parameters. Any subtype restrictions on the parameters indicated in the renaming of the subprogram are ignored; those on the original subprogram still apply. However as a matter of style and to avoid possible confusion the subtypes of any parameters (and for a function that of the return value) in a subprogram renaming should normally coincide with those of the original subprogram. The ability to be able to change the names of the formal parameters and to add, modify or remove default values for `IN` parameters in a subprogram renaming can sometimes be quite useful. For example in systems programming applications we usually want to output integers in hexadecimal format and so we might write

```
PROCEDURE Dump(Number : IN Integer;
               Width  : IN Integer := 1;
               Base   : IN Integer := 16) RENAMES CS_Int_IO.Put;
```

so that the default value of the `Base` parameter for `Dump` becomes 16 rather than 10 as it is for `CS_Int_IO.Put`. The name of the first formal parameter has also been changed from `Item` to `Number` (as it is perhaps more meaningful). Then we could write

```
Dump(255);                          -- output 255 in hexadecimal
Dump(Number => 255, Base => 8);     -- now output it in octal
```

It is not possible to rename types and subtypes in a renaming declaration. However the introduction of a new `SUBTYPE` (without imposing any additional constraints) has virtually the same effect:

```
SUBTYPE FileHandle IS Ada.Text_IO.File_Type;
SUBTYPE Cardinal IS Natural;
```

---

[1] except when a subprogram is declared in a package specification (declaration), and then a body for the subprogram is supplied by a renaming in the package body; in this case the subtypes of parameters and of the return value (if any) must match exactly . This is known as **subtype conformance**.

21.3 The scope and visibility of identifiers

We have now seen several **declarative regions** in a program where identifiers representing variables, constants, types and subtypes may be declared. To summarise, these are:

a) The main program, where identifiers for variables, constants, types and subtypes are declared.
b) A subprogram, where local identifiers for variables, constants, types and subtypes are declared, and identifiers for formal parameters are defined.
c) A block, where identifiers for variables, constants, types and subtypes which are local to the block are declared.
d) A package specification, where identifiers for variables, constants, types and subtypes are declared, which are exported from the package.
e) A package body, where identifiers for variables, constants, types and subtypes are declared, which remain private to the package body.

It is important to distinguish the concept of the life-time of a variable and its **scope**. The concept of scope applies to all identifiers, whether used to represent variables, constants, subprograms, types or subtypes and refers to the section of program text where the entity may be used:

a) For a main program identifier, declared in the declaration section of the main program, its scope is from the point where it is declared to the `END` of the main program.
b) For a local identifier, defined in a subprogram, its scope is from the point where it is declared to the `END` of the body of the subprogram.
c) For an identifier representing a formal parameter, its scope is from the subprogram heading to the `END` of the body of the subprogram.
d) For a local identifier defined in a block, its scope is from the point where it is declared to the `END` of the block.
e) For an identifier defined in a package declaration (specification), its scope is from the point where it is declared to the `END` of the package declaration. Its scope also extends to the complete package body, and to any client program which imports it using `WITH`.
f) For an identifier defined in a package body, its scope is from the point where it is declared to the `END` of the package body.

It is not permitted to declare two variables, constants, types or subtypes with the same identifier within the same declarative region, since otherwise later uses of the identifier would be ambiguous. How would the computer 'know' which of the two entities was being referenced?

A related concept to scope is **visibility**; this refers to the places in a program where the identifier may be used to refer to the associated variable, constant, type or subtype. We may state the following **visibility rules** for variables, constants, types or subtypes:

i) An entity is never visible outside its scope.

ii) If an identifier is declared in one declarative region, and the same identifier is declared in a second declarative region nested inside the first, then within the inner region it is the entity associated with the identifier declared in the inner region that is **directly visible**, and the entity associated with the identifier declared in the outer region is not directly visible within the inner region and is said to be **hidden**. Within the inner region any hidden entities can be still be used but only by giving their fully qualified names. The fully qualified name of an entity `A` declared in a main program `Main` (say) is `Main.A` whilst `Main.P.B` is the fully qualified name of a local entity `B` declared in a subprogram `P` defined in `Main`.

iii) If an identifier is declared in a package specification it is directly visible in the package specification and in the package body (unless it is hidden as described in (ii) above). It is also directly visible in any client program that imports the package using `WITH` and `USE` context clauses. If the client imports the package using only `WITH`, then the identifier is not directly visible, but the underlying entity can be accessed by using its fully qualified name: `Package_Name.Identifier`.

iv) An identifier is never visible in its own declaration. Thus the following is illegal:

```
UserID : String(1 .. 8) := (UserID'Range =>' ');
```

Instead one would need to repeat the range `1 .. 8` explicitly or (better) write:

```
UserID : String := (1 .. 8 =>' ');
```

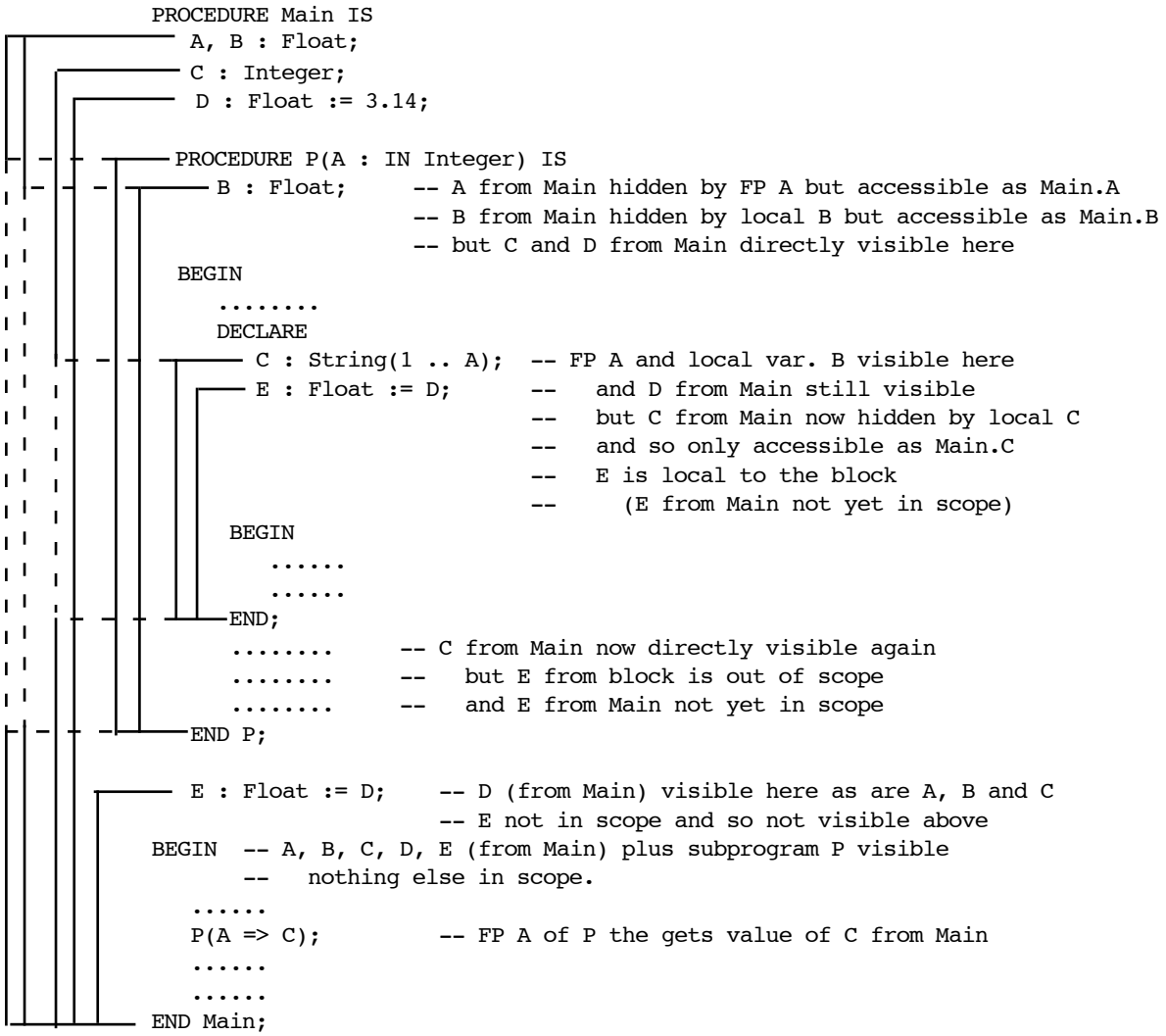
v) A variable, constant, type or subtype is hidden by the declaration of a new entity with the same identifier from the start of that declaration. Thus the following declaration (which intends to use the value of a variable `B` declared in a main program `Main` as the initial value of the local variable `B` of procedure `P`) is illegal:

```
B : Float := B;          -- !!?? illegal in Ada
```

Instead the fully qualified name must be used for the initial value:

```
B : Float := Main.B;     -- OK
```

As an extreme example to illustrate scope and visibility rules, consider the following program outline in which a vertical line denotes the region where the variable is in scope; where the line is solid the variable is directly visible and where the line is dashed, the variable is hidden by a variable of the same name in an inner scope.



In the above program outline several main program variables were referenced directly by the subprogram `P` -- very poor programming style. This was done to illustrate the scope and visibility rules rather than to illustrate good programming style! Main program variables declared before subprograms (procedures and functions) are directly visible in those subprograms. In this course we have recommended that main program variables are placed after all subprogram definitions so that their scope does not include the subprograms. The aim of this recommendation was to enforce good programming style. It is considered extremely poor programming practice to reference main program variables directly in subprograms as those subprograms are no longer self-contained. Instead, all values that need to be passed to and from subprograms should be passed via parameters. Experience has shown that large programs which violate these guidelines are difficult to understand, debug and maintain.

However it is regarded as acceptable programming style to declare types, subtypes and constants **before** subprograms so that they may be used in those subprograms. Indeed it is essential to

declare types and subtypes before subprograms if those subprograms are going to use parameters of those types and subtypes.

Even though the visibility rules describe how the Ada system resolves the situation where there are two (or more) objects referred to by the same identifier, that does not mean that we should make extensive use of them by deliberately defining the same identifier in overlapping declarative regions in our programs. Such practices can be very confusing and so are generally considered to be poor programming practice. However, it is perfectly acceptable style to use the same identifier in non-overlapping declarative regions, for example for parameters and local variables in separate subprogram definitions (provided, of course, the identifiers refer to items with similar meanings). The scope and visibility rules do, however, deal effectively with the situation in a large program (perhaps written by several programmers) where the same identifier is unwittingly used to refer to different entities. This is most common where a client program uses the same identifiers as those imported from a package.

Throughout this section we have referred to identifiers used to represent variables, constants, types and subtypes. It should be noted that identifiers are also used to name subprograms (procedures and functions), and similar scope rules to those given above also apply to subprogram identifiers. For example it is possible to nest one subprogram definition inside another (or even inside a block) and so define a local subprogram. However we have not adopted this approach in these units as such nesting of subprogram definitions is rarely necessary or justified in practice[2].

21.4 Overloading of subprograms

As we saw above if a constant, variable, type or subtype is declared with the same name as an existing one, then the original declaration is **hidden** by the new one so that the original constant, type or variable is no longer directly visible. The visibility rules for subprograms with the same name are different; we now consider what happens if a subprogram (procedure or function) is defined with the same name as an existing one. The new definition of the subprogram does not necessarily hide the original definition, both may be directly visible at the same time provided that their definitions are sufficiently different from each other. The name of the subprogram is then said to be **overloaded** as it has more than one definition.

We need to be more precise about the circumstances in which **overloading** can take place. A new subprogram definition hides an existing definition only if the number, order and base types of its formal parameters and (for a function the base type of its return value) are the same as those of the existing subprogram; subprograms which match in these respects are said to be **type conformant**. Otherwise, the new definition is associated with the name in addition to the existing one, and both definitions are directly visible at the same time. It is possible and quite common in practice to define two (or more) subprograms with the same name in the same declarative region provided they are not type conformant. Note, however, that none of the following features of formal parameters are taken into account when determining type conformance: name, mode, subtype and default value (if any).

When the Ada compiler encounters a program step in which an overloaded subprogram is called, it examines the call step to determine the base types of all the APs supplied and (for functions) the base type of the return value. It then chooses the matching definition from amongst all the definitions of the overloaded subprogram and calls that version of the subprogram. As one might expect, if no matching definition is found or if two or more matching definitions are found, a compilation error occurs.

For example in Unit 18 on sorting we defined two versions of the procedure `SISort`; one for sorting arrays of type `IntArray` (an unconstrained array type with `Integer` elements) and one for sorting arrays of type `IDArray` (an unconstrained array type with elements of the subtype `String(1..8)`). Both these procedures could be exported by a package `SIS_Pack` (say):

```
PACKAGE SIS_Pack IS
   TYPE IntArray IS ARRAY(Positive RANGE <>) OF Integer;
   SUBTYPE UnixUserID IS String(1 .. 8);
   TYPE IDArray IS ARRAY(Positive RANGE <>) OF UnixUserID;
   PROCEDURE SISort(ToSort : IN OUT IntArray);
   PROCEDURE SISort(ToSort : IN OUT IDArray);
END SIS_Pack;
```

and they could be used as follows:

---

[2] except, of course, the nesting of subprograms inside the main program procedure.

```
WITH SIS_Pack;
.....
Class     : SIS_Pack.IDArray(1..120);
ExamMarks : SIS_Pack.IntArray(1..120);
.....
SIS_Pack.SISort(Class);
SIS_Pack.SISort(ExamMarks);
```

in the first call step to `SISort` the compiler 'knows' that the AP `Class` is a subtype of the `IDArray` and so calls the version of the procedure with the matching heading, namely

```
PROCEDURE SISort(ToSort : IN OUT IDArray);
```

whereas in the second call step the AP `ExamMarks` is known to be a subtype of `IntArray` and so this time the procedure with the heading:

```
PROCEDURE SISort(ToSort : IN OUT IntArray);
```

is called. Other versions of `SISort` could be added to the package `SIS_Pack`, for example to sort an array of `Float` values and this would further overload the procedure name `SISort`.

Note it is not an error to import two type conformant subprograms with the same name `SomeProc` (say) from different packages `Pack1` and `Pack2` (say) by means of the context clauses:

```
WITH Pack1; USE Pack1;
WITH Pack2; USE Pack2;
```

However, neither version will be directly visible and so it is an error to attempt to call `SomeProc` using its unqualified name (as the compiler cannot determine which of the two subprograms should be invoked). The solution to this dilemma, of course is to use fully qualified names `Pack1.SomeProc` and `Pack2.SomeProc` to resolve the overloading. Alternatively if simple names were required they could be introduced by the renamings:

```
PROCEDURE SomeProc1(Item : IN Integer) RENAMES Pack1.SomeProc;
PROCEDURE SomeProc2(Item : IN Integer) RENAMES Pack2.SomeProc;
```

where we have assumed the original procedures had a single `IN` mode parameter of type `Integer`.

21.4.1 Overloading of standard I/O procedures

In fact we have been using overloaded procedure and functions (without, perhaps, realising it) for input and output ever since Unit 3. The package `Ada.Text_IO` provides several versions of `Get` with the following abbreviated headings:

`Get(Item : OUT Character)`	for character input from the default input source
`Get(Item : OUT String)`	for string input from the default input source
`Get(File : IN File_Type; Item :OUT Character)`	for character input from a specified file (see Unit 20)
`Get(File : IN File_Type; Item : OUT String)`	for string input from a specified file

Thus even the fully qualified name `Ada.Text_IO.Get` is overloaded, having four different definitions associated with it. In addition `CS_Int_IO` and `CS_Flt_IO` each provide two more versions of `Get`:

`Get(Item : OUT Integer)`	for integer input from the default input source
`Get(File : IN File_Type; Item : OUT Integer)`	for integer input from a specified file
`Get(Item : OUT Float)`	for floating point input from the default input source
`Get(File : IN File_Type; Item : OUT Float)`	for floating point input from a specified file

Thus the fully qualified names `CS_Int_IO.Get` and `CS_Flt_IO.Get` are each overloaded, both having two definitions associated with them. If, as is customary, we make the unqualified name `Get` available by writing the following context clauses at the head of our program:

```
USE Ada.Text_IO; USE CS_Int_IO; USE CS_Flt_IO;
```

then the name Get becomes even more heavily overloaded, having no less than 8 different interpretations. Furthermore, if we create a new package for I/O of an enumeration type (as we did in Unit 13 with the package DayOfWeek_IO for the type DayOfWeek), this associates two extra definitions with Get and so overloads it even more heavily (and so on for each new package created for enumeration I/O):

`Get(Item : OUT DayOfWeek)`	for DayOfWeek input from the default input source
`Get(File : IN File_Type; Item: OUT DayOfWeek)`	for DayOfWeek input from a specified file

Nevertheless, the compiler has no difficulty in determining which of the versions of Get is required by examining the number and base types of the APs supplied in any call of Get.

21.4.2 Overloading and programming style

As we have seen it is possible to define many different versions of a subprogram with the same name. However, there are dangers inherent in overloading particularly if a name is heavily overloaded. Firstly it could be extremely confusing to a human reader of a program if two or more subprograms with the same name performed tasks which had little or nothing in common. For this reason overloading should only used when the different versions of the subprogram perform logically the same or, at least, very similar operations. The instances of overloading discussed above fulfil this criterion, for example all the versions of Get perform basically the same action: namely the input of a single data value, only the type or the source of the data varies. Similarly the various versions of SISort all perform a straight insertion sort and differ only in the type of the array upon which they act.

A second danger of overloading is that it reduces the ability of the Ada compiler to detect programming errors as the following simple example shows. Suppose that, in a program, we have two variables (both with meaningful names):

```
Day    : DayOfWeek;      -- an enumeration type (Mon, Tues, ..., Sun)
DayNum : DayInMonth      -- an integer subrange type (range 1 .. 31)
```

We intend to input an Integer value and store it in the variable DayNum, but we accidentally type

`Get(Day);` instead of `Get(DayNum);`

The compiler will use the version of Get from DayOfWeek_IO (which, we suppose, has previously made available for use elsewhere in the program) instead of the version from CS_Int_IO that we intended to call. The program will compile without error, but a Data_Error exception will be generated at run-time when the Get step is executed and it encounters numeric data instead of the enumeration literal that it 'expects'. Such an error may prove difficult to track down in a large program, particularly if a good debugger is not available. The dangers of this sort of error can be reduced by using fully qualified names which reduce the degree of overloading. If we had written

`CS_Int_IO.Get(Day);` instead of `CS_Int_IO.Get(DayNum);`

then our error would have been detected by the compiler. It is for this reason (among others) that some books, including the recommended text for this course (Feldman and Koffman), advocate the use of fully qualified names. The counter-argument is that overly long fully qualified names make programs difficult to read.

21.4.3 Overloading enumeration literals

It is also possible to overload enumeration literals[3]. For example the literal Orange in

```
TYPE Colour IS (Red, Orange, Yellow, Green, Blue, Indigo, Violet);
TYPE Fruit IS (Apple, Lemon, Lime, Orange, Pear, Plum);
```

Usually the compiler will be able to deduce from the context which of the two possibilities is meant,

```
A : Colour;  I : Natural;
A := Orange;    -- Orange of type Colour used as A is of this type
I := Fruit'Pos(Orange);  -- Orange of type Fruit used as Fruit'Pos
                         -- expects an parameter of type Fruit
```

but if necessary literals can be distinguished by qualifying them with the type to which they belong:

`Colour'(Orange)` and `Fruit'(Orange)`

---

[3] Technically an enumeration literal is a function with no parameters returning a value of the required type.

21.5 Programming example

It is required to develop simple I/O packages for the types `Integer` and `Float` for use in an Ada 95 programming course taken by people with no previous programming experience. In the practical examples for the course, the emphasis is on simple data processing rather than computing in a scientific or engineering context. The packages are to be called `CS_Int_IO` and `CS_Flt_IO` respectively.

The 'traditional' way of doing numerical I/O in Ada programs is to **instantiate** suitable packages in the declaration section of the main program using the facilities provided in the package `Ada.Text_IO`:

```
PACKAGE My_Int_IO IS NEW Integer_IO(Integer);
PACKAGE My_Float_IO IS NEW Float_IO(Float);
USE My_Int_IO;  USE My_Float_IO;
```

These two packages provide much the same facilities as `CS_Int_IO` and `CS_Flt_IO` respectively. However this method is rather complicated for novice programmers[4]. In Ada 95 things are a little easier as two **pre-instantiated** packages are provided for numerical I/O namely `Ada.Integer_Text_IO` and `Ada.Float_Text_IO`. Thus we can simply import these packages as and when required with the context clauses (instead of the corresponding clauses for `CS_Int_IO` and `CS_Flt_IO`):

```
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
WITH Ada.Float_Text_IO;   USE Ada.Float_Text_IO;
```

However the `Put` procedures in the packages `My_Float_IO` and `Ada.Float_Text_IO` all have the disadvantage that the default values of the `Aft` parameter and `Exp` parameters are `6` and `3` respectively so that for example `0.25` is output as `2.500000E-1`. Also the default value of the `Width` parameter of the `Put` procedures in the the packages `Ada.Integer_Text_IO` and `My_Int_IO` is `11` which means that, by default, small integers are output with a number of leading blanks which often results in ugly looking output.

The solution to this problem is actually very easy: we introduce a package which uses the procedures for numerical I/O provided by the standard pre-instantiated packages and use renaming to alter the default values of the `Width`, `Fore`, `Aft` and `Exp` parameters of the exported versions of `Put`.

Complete `CS_Flt_IO` package

```
WITH Ada.Text_IO;  WITH Ada.Float_Text_IO;

PACKAGE CS_Flt_IO IS

  PROCEDURE Get(Item  : OUT Float;
                Width : IN Ada.Text_IO.Field := 0)
     RENAMES Ada.Float_Text_IO.Get;

  PROCEDURE Get(File  : IN Ada.Text_IO.File_Type;
                Item  : OUT Float;
                Width : IN Ada.Text_IO.Field := 0)
     RENAMES Ada.Float_Text_IO.Get;

  PROCEDURE Put(Item  : IN Float;
                Fore  : IN Ada.Text_IO.Field := 1;
                Aft   : IN Ada.Text_IO.Field := 2;
                Exp   : IN Ada.Text_IO.Field := 0)
     RENAMES Ada.Float_Text_IO.Put;

  PROCEDURE Put(File  : IN Ada.Text_IO.File_Type;
                Item  : IN Float;
                Fore  : IN Ada.Text_IO.Field := 1;
                Aft   : IN Ada.Text_IO.Field := 2;
                Exp   : IN Ada.Text_IO.Field := 0)
     RENAMES Ada.Float_Text_IO.Put;

END CS_Flt_IO;
```

[4] This mechanism is similar to the way of instantiating packages for the I/O of enumeration types which was discussed in Unit 13. It works in both Ada 83 and Ada 95.

Notes

i) The package `CS_Int_IO` is implemented in a similar way and can be found on Aston UNIX systems in the file `~barnesa/CSLib/cs_int_io.ads`

ii) The two packages `CS_Int_IO` and `CS_Flt_IO` are unusual in that they do not need package bodies (as the bodies of the procedures that they provide are actually those of the corresponding procedures in `Ada.Integer_Text_IO` and `Ada.Float_Text_IO`).

iii) The formatting parameters `Width`, `Fore`, `Aft` and `Exp` are declared as type `Field`. This type is 'exported' by `Ada.Text_IO` and in fact, it is a subrange of the standard subrange `Natural`.

iv) The `Get` procedures in `CS_Int_IO` and `CS_Flt_IO` have an extra formal parameter `Width` with a default value of 0 which has not been discussed so far in these Units. If a positive value `N` is supplied for `Width`, then input continues until either the end of the number is reached[5] or until `N` non-blank characters have been input whichever is the sooner. Using the default value of zero for `Width` means "continue input until the end of the number is reached".

For example if we have a number of 4-digit 24-hour clock times of the form `2115` (representing the time 9.15 p.m.), we may read the hours and minutes separately as follows:

```
Get(Item => Hrs, Width =>2);
Get(Item => Mins, Width =>2);
```

whereas the call

```
Get(Item => Time);
```

would input all four digits of the time in one 'go' and so require the hours and minute values to be separated by using division and remaindering by 100.

v) An alternative approach (necessary in Ada 83) would have been to instantiate a new package `Standard_Float_IO` (say) inside `CS_Flt_IO` and then replace every occurrence of `Ada.Float_Text_IO` with `Standard_Float_IO`. Thus the package would become

```
WITH Ada.Text_IO;

PACKAGE CS_Flt_IO IS

   PACKAGE Standard_Float_IO IS NEW Ada.Text_IO.Float_IO(Float);

   PROCEDURE Get(Item : OUT Float; Width : IN Ada.Text_IO.Field := 0)
         RENAMES Standard_Float_IO.Get;
   -- and so on for the other procedures
END CS_Flt_IO;
```

vi) Using renaming is neater and more efficient than the following approach which declares the exported procedures in the package specification in the normal way (that is without any renaming) and then defines the `Get` and `Put` procedures in the package body as follows:

```
WITH Ada.Float_Text_IO;

PACKAGE BODY CS_Flt_IO IS

   PROCEDURE Put(Item  : IN Float;
                 Fore  : IN Ada.Text_IO.Field := 1;
                 Aft   : IN Ada.Text_IO.Field := 2;
                 Exp   : IN Ada.Text_IO.Field := 0) IS
   BEGIN
      Ada.Float_Text_IO.Put(Item, Fore, Aft, Exp);
   END Put;
   -- and so on for the other procedures
END CS_Flt_IO;
```

as the latter approach requires both a package specification and a package body. Furthermore, at run-time a call to each I/O procedure will involve two procedure calls: one to the procedure in `CS_Flt_IO` and then one to the corresponding procedure in `Ada.Float_Text_IO` to do the real work. With renaming, the renamed procedure in `Ada.Float_Text_IO` is called directly at run-time (after inserting any modified default parameter values as necessary).

---

[5] that is a blank (or other non-numeric) character is encountered or end-of-line is reached.