Introduction to Systematic Programming
Unit 16 - Records.

16.1 Introduction

We have already looked at one means of collecting related items together, namely the use of arrays. Since array elements are always all of the same type, and can be accessed numerically, arrays are convenient whenever we have a (potentially large) number of related values of the same type to deal with. However, it often happens (particularly in data processing applications) that a fairly small number of values, often of different types, form a logically connected unit. For clarity and convenience, such a group of values should be given an overall name. This can be done by storing the values in an appropriately defined **record** variable. The following example shows the form of a record designed to hold the information that a company might keep for each order that it has:

Customer	PartNo	NoRequired	ItemPrice	InStock	(1)
"RoverΔΔΔΔΔ"	311260	1000	11.25	True	
NameString	RefNum	Positive	Float	Boolean	(2)

Each component of a record is known as a **field**; each field must be given a different (preferably meaningful) name; some suitable field identifiers are shown in line (1). The type associated with each field must also be specified; appropriate types for this example are shown in line (2), where the following declarations are assumed:

```
MaxNameLen : CONSTANT Positive := 10;
SUBTYPE NameString IS String(1 .. MaxNameLen);
SUBTYPE RefNum IS Positive RANGE 100000 .. 999999;
```

16.2 Record types

In Ada in order to declare a record variable we must first give a name to a type describing the complete form of the record we wish to use. For the above example, we might use the record type definition:

```
TYPE OrderType IS RECORD
                     Customer     : NameString;
                     PartNo       : RefNum;
                     NoRequired   : Positive;
                     ItemPrice    : Float;
                     InStock      : Boolean;
                   END RECORD;
```

Such a type definition encloses, between the keywords `RECORD` and `END RECORD`, a sequence of pairs of the form:

Field_name : *Type_name*;

Several successive field specifications involving the same type can be abbreviated in the usual way, for example (in a modified version of the above record type declaration) we might have:

```
ItemPrice, DiscountRate : Float;
```

We can now declare record variables, with the structure shown above, simply by specifying the type `OrderType` in a declaration. For example:

```
NewOrder : OrderType;
```

declares a variable `NewOrder` which has five fields named `Customer`, `PartNo`, `NoRequired`, `ItemPrice` and `InStock`, which are of types `NameString`, `RefNum`, `Positive`, `Float`, and `Boolean` respectively.

It is possible to have procedures and functions which have (one or more) record parameters (of a specified type) and, in the case of functions, it is also possible to specify that the function will return a record as its value. The parameters and return type are specified in the procedure or function heading in the standard way.

16.3 Accessing the fields of a record

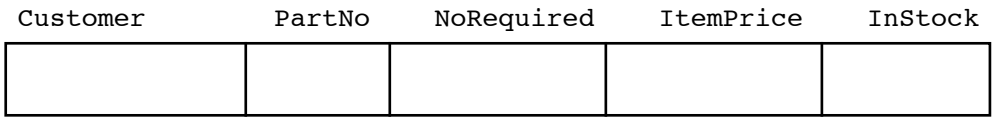
The individual elements of an array behave just like ordinary variables, and in an analogous manner, the individual fields of a record variable also behave like ordinary variables. However the notation for accessing a particular field of a record is rather different; the general form is:

```
Record_variable_name . Field_name
```

Thus `NewOrder.PartNo` refers to the particular field designated by `PartNo`, which, in this example, is a storage location capable of holding a `Positive` integer value. Some possible references to the fields of a variable `NewOrder` of type `OrderType` are:

```
NewOrder.ItemPrice := NewOrder.ItemPrice * 1.175;    -- Add VAT
```
or
```
IF NewOrder.NoRequired > 500 THEN
    NewOrder.ItemPrice := NewOrder.ItemPrice * 0.90; -- 10% discount
END IF;
```
or
```
IF NewOrder.InStock THEN
   Print delivery instructions
ELSE
   Reorder NewOrder.PartNo from manufacturer
END IF;
```
or
```
Put(NewOrder.Customer);   -- Output produced is: RoverΔΔΔΔΔ
```

Note that `NewOrder` itself refers to the whole of this five-field object:



16.4 Whole record assignment and comparison

We saw in Units 11 and 15 how it is possible to assign the entire contents of one array variable to another provided that the types of the arrays are identical. It is possible to perform an analogous operation with record variables again provided that the variables are of the same record type. For example, assuming the declarations:

```
OldOrder, NewOrder : OrderType;
```

and assuming that the variable `NewOrder` has already had values assigned to its fields, we may write:

```
OldOrder := NewOrder;
```

to copy the contents of each field of the `NewOrder` record into the corresponding fields of `OldOrder`. This is clearly far more convenient than the equivalent steps:

```
OldOrder.Customer   := NewOrder.Customer;
OldOrder.PartNo     := NewOrder.PartNo;
OldOrder.NoRequired := NewOrder.NoRequired;
OldOrder.ItemPrice  := NewOrder.ItemPrice;
OldOrder.InStock    := NewOrder.InStock;
```

It is also possible to test two records of the *same type* for equality and inequality using the operators = and /=. As might be expected two records are equal if (and only if) all their corresponding fields have the same values. For example we might write:

```
IF OldOrder /= NewOrder THEN
    Put_Line("Order changed");
END IF;
```

16.5 Record aggregates

In Units 11 and 15 we discussed array aggregates and their use in assigning values to array variables and constants. In Ada it is also possible to have **record aggregates**. The notation for these is very similar to that for arrays: a record aggregate consists of an association of fields and values written in parentheses. Aggregates may be used to assign values to record variables (or constants) by whole record assignment. For example, the assignment:

```
NewOrder := (Customer => "Vauxhall  ", PartNo => 234091,
             NoRequired => 200, ItemPrice => 9.99, InStock => True);
```

is equivalent to the five assignments:

```
NewOrder.Customer := "Vauxhall  ";
NewOrder.PartNo := 234091;
NewOrder.NoRequired := 2000;
NewOrder.ItemPrice := 9.99;
NewOrder.InStock := True;
```

but the former is obviously far more convenient.

Note that *all* the fields of the record must be given values in an aggregate, otherwise an error will occur when the whole record assignment is attempted. The most frequent uses of record aggregates is in initialised record variable declarations and constant declarations, for example (including the declarations of types `DayOfWeek` and `MonthOfYear` as in Unit 15):

```
TYPE DayOfWeek IS (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);
TYPE MonthOfYear IS (Jan, Feb, Mar, Apr, May, Jun,
                     Jul, Aug, Sep, Oct, Nov, Dec);
SUBTYPE DayOfMonth IS Positive RANGE 1 .. 31;

TYPE DateType IS RECORD
                    Day       : DayOfWeek;
                    DayNumber : DayOfMonth;
                    Month     : MonthOfYear;
                    Year      : Positive;
                 END RECORD;

D_Day : CONSTANT DateType := (Day => Tues, DayNumber => 6,
                              Month => Jun, Year => 1944);

EasterDay : DateType := (Day => Sun, DayNumber => 3,
                         Month => Apr, Year => 1994);
```

As named association is being used in these aggregates, the values for the fields may be specified in any order. Thus, for example, we could also write (as an American programmer almost certainly would!):

```
D_Day : CONSTANT DateType := (Day => Tues, Month => Jun,
                              DayNumber => 6, Year => 1944);
```

16.6 Default values for record fields

It is possible to specify default values for one or more record fields in a record type declaration and then, whenever a record variable of this type is established, the field is initialised with this default value *unless* an explicit initial value for the field is specified by means of a record aggregate. The way that a default field value is specified is very similar to the way a default value is given to a procedure formal parameter:

```
Field_name : Type_name := Default_value;
```

For example, we could define the following record type for holding a time in hours and minutes and assign the default value of zero to both fields so that any record variable of this type (such as `ElapsedTime` in the declarations which follow) would automatically be initialised to zero as it was declared:

```
SUBTYPE HourRange IS Natural RANGE 0 .. 23;
SUBTYPE MinRange  IS Natural RANGE 0 .. 59;
TYPE ClockTime IS RECORD
                     Hours   : HourRange := 0;
                     Minutes : MinRange  := 0;
                  END RECORD;
ElapsedTime : ClockTime;
```

16.7 More complex records

The examples of records that we have seen so far have been relatively straightforward having fields of the simple types `Integer`, `Float`, `String` etc. However it is possible for the fields of a record to be any type, including user-defined structured types such as arrays and other records. For example we might have a record type `NewBornType` declared as follows:

```
SUBTYPE OunceRange IS Natural RANGE 0 .. 15;
TYPE WeightType IS RECORD
                      Pounds : Natural;
                      Ounces : OunceRange;
                   END RECORD;
TYPE GenderType IS (Male, Female);
TYPE FullName IS RECORD
                    Forename   : NameString;
                    SecondName : NameString;
                    Surname    : NameString;
                 END RECORD;

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

This record type has five fields, of which one is a user-defined enumeration type and the other four are themselves records. Note also that the field `MothersName` is a record which has three fields all of which are arrays. In theory, arrays and records may be nested to any depth to build up data structures of virtually unlimited complexity. Given the variable declaration:

```
Baby : NewBornType;
```

all the following are valid references to parts of the record variable `Baby`:

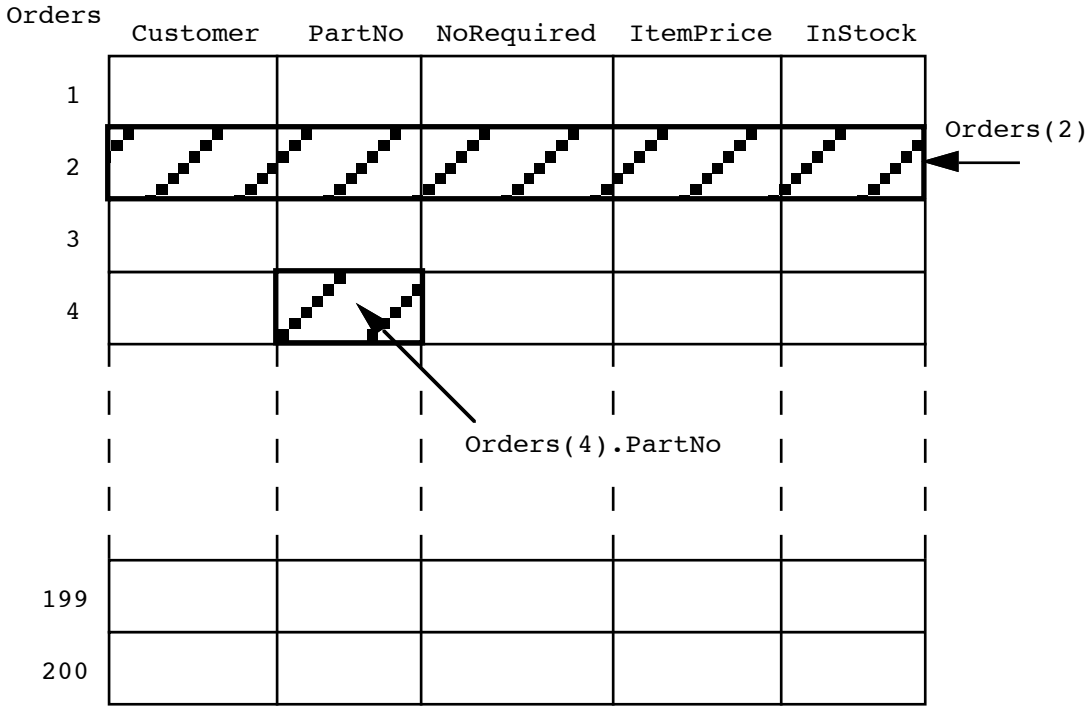
`Baby.DateOfBirth.Month`	gives the month of birth
`Baby.TimeOfBirth.Hours`	gives the hour of birth
`Baby.MothersName.Surname`	gives the mothers surname
`Baby.MothersName.Forename(1)`	gives the mothers first initial

16.8 Arrays of records

In realistic programs it is often necessary to store collections of records; a logical way to do this is to use an array of records. For example to handle 200 orders, we might have the declarations:

```
NumOrders : CONSTANT Positive := 200;
TYPE OrderList IS ARRAY (1.. NumOrders) OF OrderType;
Orders, OldOrders : OrderList;
```

that is an array type in which each element is capable of storing a complete `OrderType` record, with each such record comprising the five fields specified in the declaration given in [16.2]. This is a much better storage arrangement than attempting to represent the data structure using five parallel arrays, each array having 200 elements of type `NameString`, `RefNum`, `Positive`, `Float` and `Boolean` respectively. The two variables `Orders` and `OldOrders` should thus be visualised as:



Thus `Orders(2)` refers to a complete record, and `Orders(4).PartNo` refers to a particular field in a particular record element of the array.

Fields in arrays of records can, once accessed, be utilised in the same way as fields of any other record, for example to add 17.5% VAT to each order in hand:

```
FOR OrderNo IN 1 .. NumOrders LOOP
   Orders(OrderNo).ItemPrice := Orders(OrderNo).ItemPrice * 1.175;
END LOOP;
```

Note that the array index is attached to `Orders` (rather than `ItemPrice`) to select a particular *record* from the array of records and *then* the field `ItemPrice` is selected from that record. Compare this with the example in [16.7] where in `Baby.MothersName.Forename(1)` the array index is attached to the field name since it is the *field* `Forename` which is the *array,* and we want to select the first element of this array to get the mother's initial.

Also, since `Orders` and `OldOrders` are arrays (even though their elements are records consisting of more than single values), whole array assignment such as:

```
OldOrders := Orders;
```

can be used to copy the entire contents of the array of records `Orders` into `OldOrders`. In general in Ada we can use such assignment steps to copy the contents of one variable into another provided that the types of the variables are identical.

Programming example

An airline company wishes to computerise the allocation of seats on its thirty daily outward flights from a particular airport. Each flight has a reference number in the range 1001 to 1030, and the aircraft involved can all carry up to 100 passengers: 20 in first class and 80 in economy class. An Ada program is required to process a list of flight requests of the form:

Booking_reference_no. Flight_no. No_of_seats_required Class

where the *Class* is either the string "`first class`" or "`economy class`" . The program should process each flight request by either:

i) rejecting the request if the desired flight is already fully booked for the requested class, or

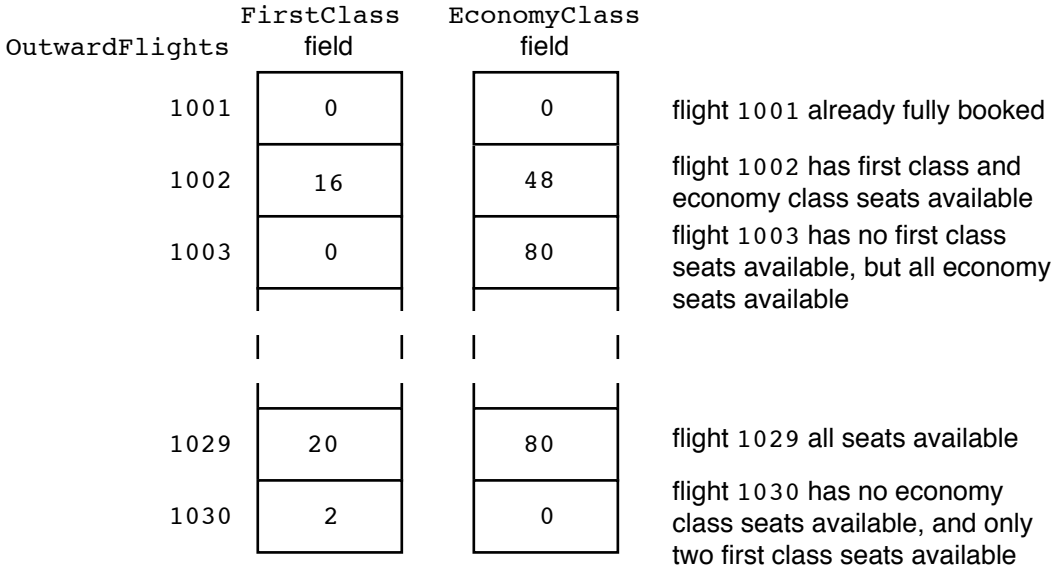
ii) confirming the request, and allocating the required number of seats for the requested flight and class.

Data structures

We see that three pieces of information are associated with each flight; its flight reference number, and the number of seats still available in first and economy class. This suggests that we should define a record type `PlaneSeats` (say) to hold information about seats still available for a particular flight. If we give the maximum number of seats available in each class as the default values for the fields of this record, we can arrange that the number of seats available in each class is always correctly initialised when a record of this type is set up. Now to note the seats still available for each flight, use an array of `PlaneSeats` records (one record for each flight) indexed by the flight reference number:

```
SUBTYPE FirstClRange   IS Natural RANGE 0 .. 20;
SUBTYPE EconomyClRange IS Natural RANGE 0 .. 80;
TYPE PlaneSeats IS
          RECORD
             FirstClass   : FirstClRange   := 20;
             EconomyClass : EconomyClRange := 80;
          END RECORD;
SUBTYPE FlightNumRange IS Positive RANGE 1001 .. 1030;
TYPE FlightList IS ARRAY (FlightNumRange) OF PlaneSeats;
OutwardFlights : FlightList;
```

This array might therefore contain (at some instant part way through processing the flight requests) the following values:



Thus, for example, the number of first and economy class seats available on flight number `1028` would be given by:

```
OutwardFlights(1028).FirstClass
```

and:

```
OutwardFlights(1028).EconomyClass
```

Each booking request involves four pieces of information: a booking reference number, a flight number, the number and class of seats required. Thus it is also convenient to declare a second record type to store the information about booking requests:

```
TYPE SeatType IS (First, Economy);
TYPE  BookingRequest IS
         RECORD
            RefNo     : Positive;
            FlightNum : FlightNumRange;
            Seats     : Positive;
            Class     : SeatType;
         END RECORD;
```

The final program is shown on the following pages.

```
WITH Ada.Text_IO;  USE Ada.Text_IO;
WITH CS_Int_IO;    USE CS_Int_IO;
WITH CS_File_IO;   USE CS_File_IO;

PROCEDURE Bookings IS
   -- Program for Unit 16 of the ISP Ada course.
   -- Written by A Barnes, November 1993, updated for Ada95, August 1996.

   MinFlightNum    : CONSTANT Positive := 1001;
   MaxFlightNum    : CONSTANT Positive := 1030;
   NumFirstSeats   : CONSTANT Natural  := 20;
   NumEconomySeats : CONSTANT Natural  := 80;

   SUBTYPE FirstClRange   IS Natural RANGE 0 .. NumFirstSeats;
   SUBTYPE EconomyClRange IS Natural RANGE 0 .. NumEconomySeats;
   TYPE PlaneSeats IS
           RECORD
              FirstClass   : FirstClRange   := NumFirstSeats;
              EconomyClass : EconomyClRange := NumEconomySeats;
           END RECORD;

   SUBTYPE FlightNumRange IS Positive RANGE MinFlightNum .. MaxFlightNum;
   TYPE FlightList IS ARRAY (FlightNumRange) OF PlaneSeats;
   TYPE SeatType IS (First, Economy);
   TYPE BookingRequest IS
            RECORD
               RefNo     : Positive;
               FlightNum : FlightNumRange;
               Seats     : Positive;
               Class     : SeatType;
            END RECORD;

   PACKAGE SeatType_IO IS NEW Enumeration_IO(SeatType);
   USE SeatType_IO;

   PROCEDURE InputBooking (Booking : OUT BookingRequest) IS
   BEGIN
       Get(Booking.RefNo);
       Get(Booking.FlightNum);
       Get(Booking.Seats);
       Get(Booking.Class);
       Skip_Line;   -- Skip over the word "class" to start of next line.
   END InputBooking;

   PROCEDURE ProcessBooking (Booking : IN BookingRequest;
                             Flight  : IN OUT PlaneSeats) IS
       Reject : Boolean;
   BEGIN
      CASE Booking.Class IS
         WHEN First =>
            Reject := Flight.FirstClass < Booking.Seats;
         WHEN Economy =>
            Reject := Flight.EconomyClass < Booking.Seats;
      END CASE;
      IF Reject THEN
         Put_Line("Booking rejected");
      ELSE
         Put_Line("Booking confirmed");
         CASE Booking.Class IS
            WHEN First =>
               Flight.FirstClass := Flight.FirstClass - Booking.Seats;
            WHEN Economy =>
               Flight.EconomyClass := Flight.EconomyClass
                                                    - Booking.Seats;
         END CASE;
      END IF;
   END ProcessBooking;
```

```
   PROCEDURE OutputDetails (Booking : IN BookingRequest) IS
      Blank : CONSTANT Character := ' ';
   BEGIN
       Put("Booking Reference Number ");
       Put(Booking.RefNo);
       Put_Line(": ");
       Put(Booking.Seats);
       Put(Blank);
       Put(Item => Booking.Class, Width => 1, Set => Lower_Case);
       Put(" class seats requested on flight number ");
       Put(Booking.FlightNum);
       New_Line;
   END OutputDetails;

   -- Main program variables
   OutwardFlights : FlightList;
   Request : BookingRequest;

BEGIN   -- Main program of Bookings

   Put_Line("Please enter the name of the input file");
   OpenInput;

   Put_Line("Please enter a different name for the output file");
   OpenOutput;

   WHILE NOT End_Of_File LOOP
      InputBooking(Request);
      OutputDetails(Request);
      ProcessBooking(Booking => Request,
                     Flight => OutwardFlights(Request.FlightNum));
   END LOOP;

   CloseInput;
   CloseOutput;

END Bookings;
```

Notes

a) In the procedure `OutputDetails` the step:

```
Put(Item => Booking.Class, Width => 1, Set => Lower_Case);
```

was used to output values of the enumeration type `SeatType`. The parameter `Width` controls the number of print positions used to output the enumeration literals (using the value 1 means, in effect, print them in the minimum amount of space required), and the parameter `Set` controls whether they are printed using the upper or lower case character set. If we had wanted the enumeration value printed using upper case letters we could have used the call:

```
Put(Item => Booking.Class, Width => 1, Set => Upper_Case);
```

where the actual parameters `Lower_Case` and `Upper_Case`, are enumeration literals provided by the `Enumeration_IO` package (and hence by the `SeatType_IO` package since it is derived from the `Enumeration_IO` package).

b) Notice how the use of records to group related data values together into a single variable has not only improved the general clarity of our program, but also has simplified how our procedures may be called. For example, the `InputBooking` procedure is called with *one* parameter (of the record type `BookingRequest`), rather than with *four* parameters corresponding to the four values which comprise a booking request.