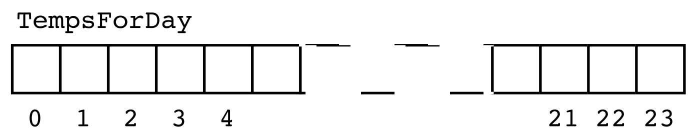
Introduction to Systematic Programming
Unit 19 - Two-dimensional Arrays

19.1 Two-dimensional arrays - introductory example

The arrays we have considered so far are said to be **one-dimensional** arrays. We can visualise such an array as a linear (one-dimensional) collection of storage locations; for example to store the temperature observed at each hour of the day (from 00.00 to 23.00), we could have:

```
TYPE DayReadings IS ARRAY (0..23) OF Float;
TempsForDay : DayReadings;
```

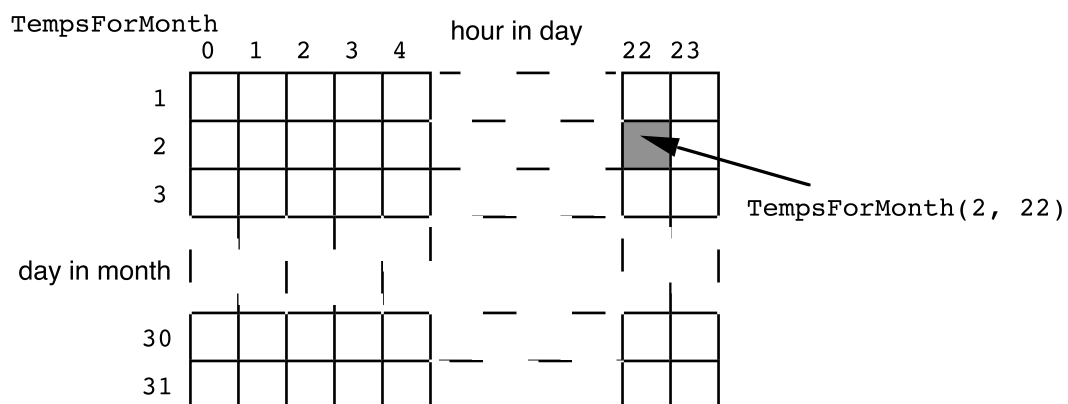
which may be visualised as:



Suppose instead we wish to record the temperatures observed at each hour of every day of a 31-day month? One possibility would be to have a one-dimensional array with one element for each of the 31×24 = 744 observations, that is a variable with type `ARRAY (1..744) OF Float`. However, if we wish to refer to the temperature at noon on the fifteenth of the month (say), a calculation is needed to reveal that this is element number 14×24 + 12 = 348. However, this calculation is rather obscure and error-prone[1]. This suggests that we should declare an array type in which it is easier to specify the day and the hour separately, namely:

```
TYPE MonthReadings IS ARRAY (1..31, 0..23) OF Float;
TempsForMonth : MonthReadings;
```

Such an array is best visualised as a **two-dimensional** structure as shown:



We see that `TempsForMonth` has 31 rows, numbered from `1` to `31`, each of which comprises 24 elements numbered from `0` to `23`. The intention is that row `D` should be used to store the observations for the `D`th day of the month, where element `H` in that row is used to store the observation for the `H`th hour of that day. How are these elements to be accessed in Ada?

Assuming the declarations:

```
SUBTYPE DayInMonth IS Integer RANGE 1..31;
SUBTYPE HourInDay  IS Integer RANGE 0..23;
Day  : DayInMonth;
Hour : HourInDay;
```

then: `TempsForMonth(Day, Hour)`

accesses the element of `TempsForMonth` with the indices `Day` and `Hour` giving the temperature at the specified hour of the given day. The first index `Day` selects the row of the two-dimensional structure and then the second index `Hour` selects the appropriate column in that row.

---

[1] It is actually element number 349 !

© 1996 A Barnes, L J Hazlewood

19.2 General rules on array dimensions

a) Arrays are not restricted to 1 or 2 dimensions - 3, 4, 5 or indeed more are permitted, but practical problems only rarely require the use of 3 dimensions, and virtually never justify the use of 4 or more dimensions.

b) No matter how many dimensions, all the individual elements of an array store the same type of value (`Integer`, `Float`, `Character`, etc.).

c) When defining a 2-D array (as in the example in [19.1] above), the array bounds for *all the dimensions* of the array must be specified in the array type definition.

d) Type and subtype names may be used (instead of explicit ranges) when specifying index bounds in array type declarations just as is the case with one-dimensional arrays. The subscripts in each dimension of a two-dimensional array do not have to be of the same type.

e) Always remember that if `A` is a 2-D array, then `A(I,J)` refers to *row* `I`, *column* `J` *in that order*.

19.2.1 Examples

Here are some examples of multi-dimensional array type and array variable declarations:

```
NumRows CONSTANT Positive := 20;
NumCols CONSTANT Positive := 80;
TYPE PageType IS ARRAY (1..NumRows, 1..NumCols) OF Character;
Page : PageType;

TYPE DayOfWeek IS (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);
TYPE WeekCounts IS ARRAY (DayOfWeek, HourInDay) OF Natural;
NumberOfBirths : WeekCounts;

TYPE SocialGrp IS (A,B,C1,C2,D,E);
TYPE Gender IS (Male, Female);
SUBTYPE AgeRange IS Natural RANGE 0..120;
TYPE Population IS ARRAY (AgeRange, Gender, SocialGrp) OF Natural;
Census : Population;
```

In the second example, the array type `WeekCounts` has its first dimension indexed by one of the values of the enumeration type `DayOfWeek` and its second indexed by an `Integer` value in the range `0..23`.

In the third example a three-dimensional array type `Population` is declared. Array variables of this type could perhaps hold the results of a survey of the number of people in the population by age, sex and social class. Then, for example, the array element:

```
Census(46, Male, C1)
```

would give the number of 46 year old males of social class `C1` in the population survey.

19.2.2 Unconstrained 2-D arrays

It is also possible to define unconstrained 2-D array types, where *both* index ranges are unconstrained, but (as with 1-D arrays) any array variable of this type must be declared to be of a *fully constrained* subtype, for example:

```
TYPE PeriodReadings IS
        ARRAY (DayInMonth RANGE <>, HourInDay RANGE <>) OF Float;

SUBTYPE Week1Readings IS PeriodReadings(1..7, HourInDay);
-- Temperatures during the first week of the month
TempsForWeek1 : Week1Readings;

SUBTYPE AfternoonReadings IS PeriodReadings(DayInMonth, 12..18);
-- Temperatures for every afternoon in the month
AfternoonTemps : AfternoonReadings;

SUBTYPE Week1PMReadings IS PeriodReadings(1..7, 12..23);
-- P.M. temperatures during the first week of the month
TempsForWeek1PM : Week1PMReadings;
```

© 1996 A Barnes, L J Hazlewood

However it is *not* legal to have one index constrained and the other unconstrained; as in:

```
SUBTYPE SomeDaysReadings IS
                 PeriodReadings(DayInMonth RANGE <>, HourInDay);
                                        -- !? Illegal in Ada
```

19.2.3 Array attributes

Since 2-D arrays have two indices, all the array attribute functions take a parameter (either 1 or 2) to select either the first or the second dimension (this idea extends to 3 or more dimensions in the obvious way). Here are a few examples:

`Week1PMReadings'Last(1)`	yields the value `7`
`Week1PMReadings'First(2)`	yields the value `12`
`Week1PMReadings'Last(2)`	yields the value `23`
`Week1PMReadings'Range(1)`	yields the range `1..7`

Note that the above attribute functions have been applied to the subtype `Week1PMReadings`. We could equally well have applied them in this case to the variable `TempsForWeek1PM`.

19.3 Using `FOR` loops with two-dimensional arrays

Given a two-dimensional array such as `TempsForMonth`, it is quite common to want to apply the same operation to all the elements of one row of the array, or to all the elements of one column of the array, or perhaps to all of the elements of the entire array. In such situations `FOR` loops provide a very convenient means of achieving these actions, but it is necessary to be careful to use the correct array index to control the loop, that is either the 1st (row) or 2nd (column) index, as appropriate. Assuming the declarations in [19.1] above, then to obtain the average temperature for some given day (that is across a row):

```
Total : Float := 0.0;
Average : Float;
......
Get(Day);
FOR ThisHour IN 0..23 LOOP
-- Processing across a row implies column index varying
   Total := Total + TempsForMonth(Day, ThisHour);
END LOOP;
Average := Total/24.0;
```

Whereas (assuming the same declarations of `Total` and `Average` as above), to obtain the average at a given hour over all the days (that is down a column) we would write:

```
Get(Hour);
FOR ThisDay IN 1..31 LOOP
-- Processing down a column implies row index varying
   Total := Total + TempsForMonth(ThisDay, Hour);
END LOOP;
Average := Total/31.0;
```

To obtain an average temperature for the whole month, nested `FOR` loops are required:

```
FOR ThisDay IN 1..31 LOOP
   FOR ThisHour IN 0..23 LOOP
     Total := Total + TempsForMonth(ThisDay, ThisHour);
   END LOOP;
END LOOP;
Average := Total/(31.0 * 24.0);
```

With the `FOR` parts in the order shown, the total is accumulated row by row; the same result could equally well be achieved by interchanging the `FOR` parts, calculating the same total in a column by column fashion thus:

```
FOR ThisHour IN 0..23 LOOP
   FOR ThisDay IN 1..31 LOOP
      Total := Total + TempsForMonth(ThisDay, ThisHour);
   END LOOP;
END LOOP;
Average := Total/(31.0 * 24.0);
```

As another example, in order to set each element of the array `Page` to blank, we could use the nested `FOR` loops:

```
FOR Row IN 1..NumRows LOOP
   FOR Column IN 1..NumCols LOOP
      Page(Row, Column) := ' ';
   END LOOP;
END LOOP;
```

19.4 Arrays of arrays

The declaration given for `TempsForMonth` in [19.1] above is not the only way in which we could declare a two-dimensional array variable in Ada. Just as we can define an array of records, that is an array whose elements are records (see Unit 16) we may define an array whose elements are themselves arrays, that is an **array of arrays**. Since we already know how to handle a set of observations for one day (using a one-dimensional array type `DayReadings`) another way to store a full month's observations is using the following declarations:

```
TYPE DayReadings IS ARRAY (0..23) OF Float;
TYPE NewMonthReadings IS ARRAY (1..31) OF DayReadings;
NewTempsForMonth : NewMonthReadings;
```

We are declaring `NewTempsForMonth` to be an array of arrays and again the structure can be visualised as the two-dimensional structure discussed in [19.1]. An 'element' of this array such as:

```
NewTempsForMonth(Day)
```

is *itself* an array (of type `DayReadings`). Now since this 'element' is an array it can, in turn, be subscripted. Thus:

```
NewTempsForMonth(Day)(Hour)
```

accesses the element in 'column' `Hour` and in 'row' `Day` of `NewTempsForMonth`. Note carefully the differences in the notation for selecting an element from a 2-D array and from an array of arrays.

It is also possible to define unconstrained types for an array of arrays, but only the *first dimension* may be unconstrained. Thus, for example:

```
TYPE SomeTimeReadings IS ARRAY (HourInDay RANGE <>) OF Float;
TYPE SomeReadings IS ARRAY (DayInMonth) OF SomeTimeReadings;
                                          -- !? Illegal in Ada
```

is *not* legal as the array type `SomeTimeReadings` is unconstrained. However:

```
TYPE PeriodReadings IS ARRAY (DayInMonth RANGE <>) OF DayReadings;
```

is perfectly legal, and array variables of this type may be declared in the usual way by supplying index constraints, for example:

```
SUBTYPE Week1Readings IS PeriodReadings(1..7);
TempsForWeek1 : Week1Readings;
```

Thus unconstrained array of array types are somewhat less flexible than true 2-D arrays in this respect. However there are occasions when the use of array of array types is more flexible than 2-D array types. For example each row of an array of type `NewMonthReadings` is an array of type `DayReadings` and so the following is valid:

```
TempsForToday : DayReadings;
.....
TempsForToday := NewTempsForMonth(8);
```

whereas the comparable operations for a 2-D array are not legal in Ada.

It is also possible to take a slice of an array of arrays. In Ada slices can be taken of any 1-D array and an array of arrays is just a 1-D array (whose elements happen to be arrays). Thus, for example, the following are both valid:

`NewTempsForMonth(1 .. 7)` and `NewTempsForMonth(14)(12 .. 23)`

In the first example a slice of the first seven elements of `NewTempsForMonth` is selected (each of these elements is an array of type `DayReadings`); this slice gives the temperatures during the first week of the month in question. In the second example `NewTempsForMonth(14)` is an array of `Float` values (of type `DayReadings`) and the slice selected gives the temperatures after noon on that day.

For most problems we may choose to use either a true 2-D array or an array of arrays. If the two indices are logically of equal importance, then a 2-D array structure is more appropriate. However, if the problem is such that the first index is logically of greater importance than the second (if for example it involves the handling of whole rows as single entities) then an array of arrays is more appropriate.

19.5 Whole array assignment and comparison

As might be expected whole array assignment may be used to assign the contents of one 2-D array to another (or of one array of arrays to another) *provided that* they are of the same type. Thus given the declaration:

```
TempsForMonth, TempsForLastMonth : MonthReadings;
```

we could simply copy the 744 elements of `TempsForMonth` into `TempsForLastMonth` by writing:

```
TempsForLastMonth := TempsForMonth;
```

Similarly we may compare two 2-D arrays (or two arrays of arrays) for equality or inequality using the comparison operators = and /= provided again that both arrays are of the same type.

However, it is not possible to assign (or compare) a 2-D array to an array of arrays (even if the index types and index bounds are the same). Thus for example the following assignment is not valid:

```
NewTempsForMonth := TempsForMonth;  -- !? Illegal in Ada
```

19.6 Two-dimensional aggregates

These are logical extensions of the concept of array aggregates in one-dimension: a 2-D aggregate is an aggregate in which each assigned value is itself a 1-D aggregate; two examples should clarify this. Suppose first that we wish to initialise to zero every element in the array of arrays `NewTempsForMonth` of type `NewMonthReadings` considered in [19.4] above, then we would write:

```
NewTempsForMonth := (1..31 => (0..23 => 0.0));
```

or even:

```
NewTempsForMonth := (DayInMonth => (HourInDay => 0.0));
```

Each of the 31 'rows' of the two-dimensional aggregate (indexed by the subrange type `DayInMonth`) is itself an aggregate consisting of 24 zero values (indexed by the subrange type `HourInDay`).

Secondly assuming the following declarations:

```
Blank : CONSTANT Character := ' ';
Star  : CONSTANT Character := '*';
```

the 2-D array `Page`, given in [19.2.1] above could be initialised to blanks with a border of stars by a 2-D aggregate as follows:

```
Page := (1|NumRows => (1..NumCols => Star),
         2..NumRows-1 => (1|NumCols => Star, 2..NumCols-1 => Blank));
```

In this example the first and last rows of the page are initialised to aggregates consisting completely of stars, whereas the remaining rows are initialised to aggregates whose first and last elements are stars and whose remaining elements are blanks.

Note that the aggregates used with a 2-D array have exactly the same form as those used with array of arrays.

19.7 Programming example

A number of first year Computer Science students at Saton University were interviewed several times a week during the Autumn term. They were asked to state the number of hours (to the nearest hour) that they had spent that day seated at a VDU programming in Ada. Their responses were later typed into a data file, one line for each response. An Ada program is required to input this data, and display it as a histogram to fit on a VDU screen in the form:

```
                                                             line

                            *****                               10
                            *   *                                9
                            *   *                                8
                        *****   *                                7
                    *****   *   *                                6
                    *   *   *   *****                            5
                    *   *   *   *   *                            4
                *****   *   *   *   *****                        3
            *****   *   *   *   *   *   *                        2
            *   *   *   *   *   *   *   *                        1
*****************************************************            0
  0   1   2   3   4   5   6   7   8   9   10  11  12
```

The height of the bars being scaled so that the highest bar is 10 lines high. It is reasonable to assume that no student spends more than 12 hours per day seated at a VDU (!?).

19.7.1 Data structures

Firstly we need an array:

```
TYPE FrequencyList IS ARRAY (0..12) OF Natural;
Frequencies : FrequencyList;
```

to count the number of responses for each of the possible number of hours 0, 1, 2, ..., 12 spent at the terminal.

One method of forming and displaying the histogram is to declare an array of arrays:

```
SUBTYPE LineType IS String(1 .. MaxCols);
TYPE PageType IS ARRAY (0 .. 10) OF LineType;
Screen : PageType;
```

within which an 'image' of the histogram can be constructed before it is output to the VDU display.

Each element of the array corresponds to a character position on the VDU screen. The first subscript represents the line number on the screen, numbered from 0 at the bottom of the screen (the scale being output separately). The second subscript represents the column number on the screen, numbered from 1 at the left hand side of the screen. An array of arrays is preferable here to a 2-D array as complete rows of the chart may be output by calling the procedure `Put_Line`.

Final program

```
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 Histogram IS
   -- Example program for Unit 19, ISP Ada Course, illustrating the
   -- use of an array of arrays.
   -- Written by A Barnes, December 1993.
   -- Updated for Ada95 by A Barnes, August 1996.

   MaxHours : CONSTANT Positive := 12;
   MaxLines : CONSTANT Positive := 10;
   Margin   : CONSTANT Positive := 10;
   BarWidth : CONSTANT Positive := 5;
   MaxCols  : CONSTANT Positive
                         := Margin + (BarWidth-1)*(MaxHours+1) + 1;
   Blank    : CONSTANT Character := ' ';
   BarChar  : CONSTANT Character := '*';

   SUBTYPE LineRange IS Natural RANGE 0..MaxLines;
   SUBTYPE ColRange  IS Natural RANGE 1..MaxCols;
   SUBTYPE LineType IS String(ColRange);
   TYPE PageType IS ARRAY (LineRange) OF LineType;

   SUBTYPE HourRange IS Natural RANGE 0..MaxHours;
   TYPE FrequencyList IS ARRAY (HourRange) OF Natural;

   PROCEDURE InputData (Counts : IN OUT FrequencyList) IS
      -- To input the number of hours each student spent hacking
      -- and count the frequency of these values.
      -- Input is terminated by reaching end-of-file.
      NumHours : HourRange;
   BEGIN
      -- Input and count the number of responses for each hour.
      WHILE NOT End_Of_File LOOP
         WHILE NOT End_Of_Line LOOP
            Get(NumHours);
            Counts(NumHours) := Counts(NumHours) + 1;
         END LOOP;
         Skip_Line;
      END LOOP;
   END InputData;

   FUNCTION Maximum (Freq : IN FrequencyList) RETURN Natural IS
      -- Function to find the number of hours value with the
      -- largest frequency.
      MaxSoFar : Natural := 0;
   BEGIN
      FOR NumHours IN HourRange LOOP
         IF Freq(NumHours) > MaxSoFar THEN
            MaxSoFar := Freq(NumHours);
         END IF;
      END LOOP;
      RETURN MaxSoFar;
   END Maximum;

   PROCEDURE WriteOut (Page : IN PageType) IS
      -- To output the contents of the Page array to the VDU.
   BEGIN
      FOR Line IN REVERSE LineRange LOOP
         Put_Line(Page(Line));
      END LOOP;
   END WriteOut;
```

```
   PROCEDURE OutputScale IS
      -- To output the scale which appears under the histogram.
   BEGIN
      -- Procedure stub.
      Null; -- Left as an exercise to the reader.
   END OutputScale;

   PROCEDURE DrawBar (Page   : IN OUT PageType; Time : IN HourRange;
                      Height : IN Natural) IS
      -- To place one histogram bar into the Page array.
      LeftCol  : ColRange;
      RightCol : ColRange;
   BEGIN
      LeftCol  := Margin + Time*(BarWidth-1) + 1;
      RightCol := LeftCol + BarWidth - 1;
      -- Draw left side of bar
      FOR Line IN 1 .. Height-1 LOOP
         Page(Line)(LeftCol) := BarChar;
      END LOOP;
      -- Draw right side of bar
      FOR Line IN 1 .. Height-1 LOOP
         Page(Line)(RightCol) := BarChar;
      END LOOP;
      -- Draw  top of bar
      Page(Height)(LeftCol..RightCol) := (1..BarWidth => BarChar);
   END DrawBar;

   PROCEDURE DisplayHistogram (Counts : IN FrequencyList) IS
      -- Place a histogram whose bars correspond to the number of
      -- hours frequency values into a Page array of characters,
      -- and then output.
      Screen    : PageType := (LineRange => (ColRange => Blank));
      MaxCount  : Natural;
      BarHeight : Natural;
      Factor    : Float;
   BEGIN
      MaxCount := Maximum(Counts);
      Factor   := Float(MaxLines)/Float(MaxCount);
      -- All Counts are scaled by Factor before being displayed.

      -- Draw bottom line of the chart.
      Screen(0) := (1..Margin => Blank, Margin+1..MaxCols => BarChar);

      FOR NumHours IN HourRange LOOP
         BarHeight := Integer(Float(Counts(NumHours))*Factor);
         DrawBar(Page => Screen,
                 Time => NumHours, Height => BarHeight);
      END LOOP;

      New_Line(5);
      WriteOut(Screen);
      OutputScale;
      New_Line(5);
   END DisplayHistogram;

   -- Main program variables.
   Frequencies : FrequencyList := (HourRange => 0);

BEGIN  -- Main program of Histogram
   OpenInput(FileName => "survey.dat");
   InputData(Frequencies);
   CloseInput;
   DisplayHistogram(Frequencies);
END Histogram;
```