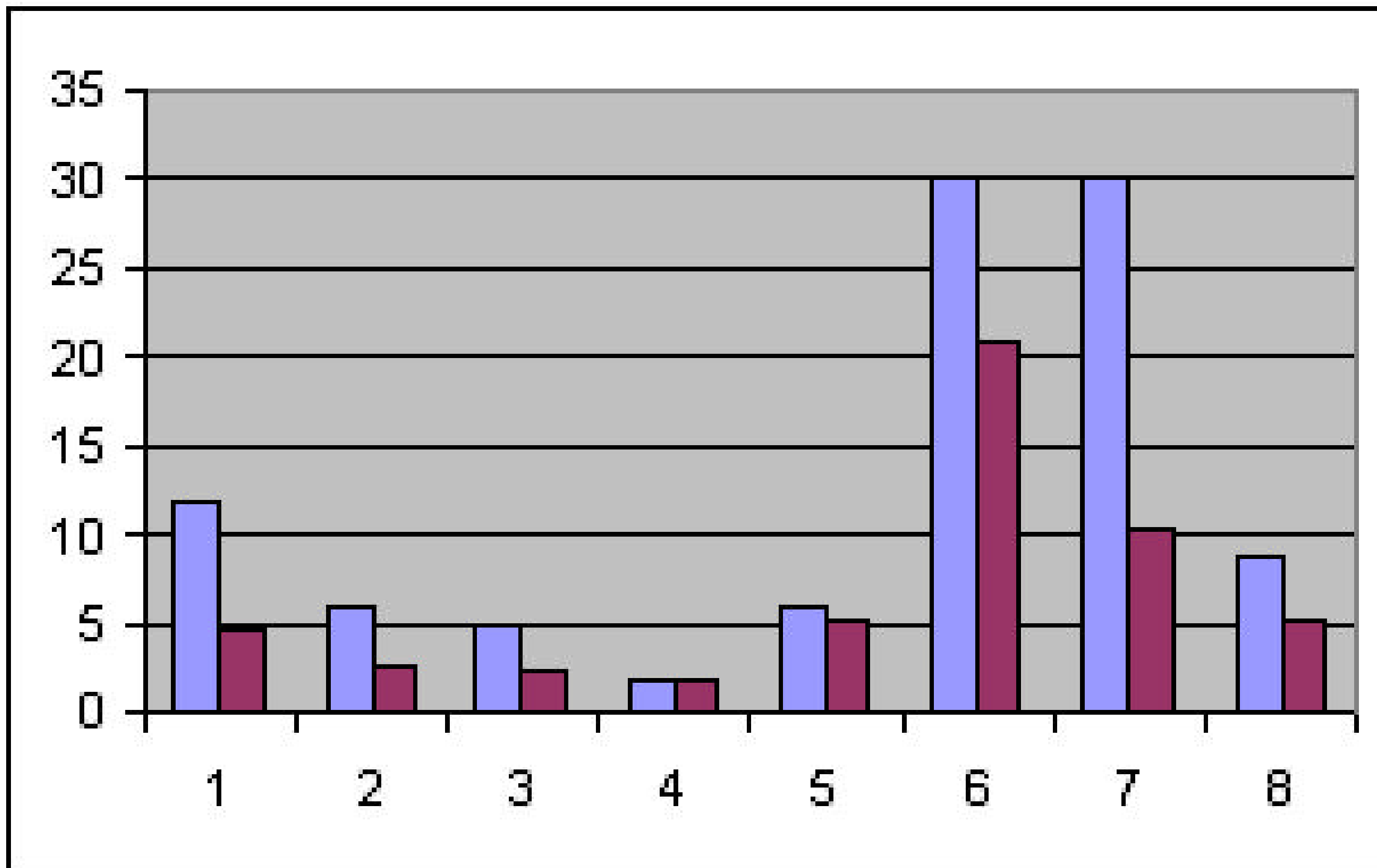
16.070

Introduction to Computers & Programming

Ada: Structured data types – Records, Arrays

Prof. Kristina Lundqvist
Dept. of Aero/Astro, MIT


35
30
25
20
15
10
5
0
1
2
3
4
5
6
7
8

Structured data types

- Till now we have used **scalar** (single value) data types
- **Structured** (multiple value) data type
  - We may want to store multiple items of data that all pertain to the same thing
  - We may want to store similar data about each of many things

- For these we use **records** and **arrays** respectively

Records

- A **record** is a data structure that collects together into one unit several related items of data
  - They are related in that they all pertain to the same thing
    - The name, phone number, sex, age, and weight of a person
    - The day number, month name, and year number that make up a date
    - Etc…
  - They each may have different data types

Records

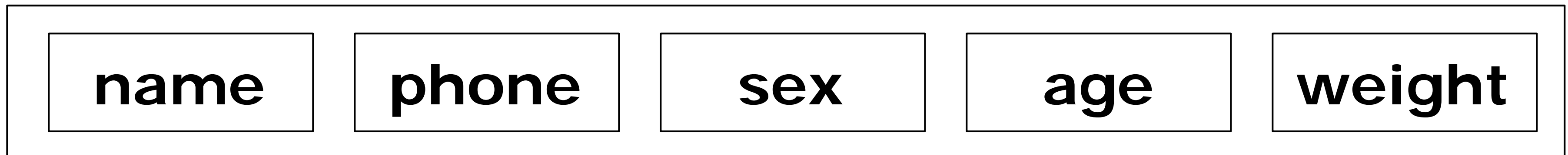
- To use records in Ada programs you need to know the following
  - How to **design** a record
  - How to **declare** record types and variables
  - How to **use** a record
  - What **hierarchical records** are and how to use them

Designing Records

- Programs often use information about real-world objects
  - Usually only a small subset of all possible information
    - For example, the number of things you can record about a person is almost limitless, but for any particular purpose only a few things are relevant.
  - part of design process is to select the appropriate subset
    - For example, a fitness club may need to know the name, phone number, sex, age, and weight of its customers.
  - would use a **record** to combine components

Designing Records

- To design a record:
  - identify the items of data that are relevant in *this* application
  - use a data structure diagram to show the relevant information
    - decide on names for the overall structure, and for the individual fields
  - determine the data types of the fields




```
name   : names;     -- string sub-type
phone  : phones;    -- string sub-type
sex    : sexes;     -- enumerated type
age    : ages;      -- integer sub-range
weight : weights;   -- float sub-type
```

Declaring records

declaring record types

- Each component of a **record** is called a **field**.
- Declaring a record involves declaring the name and type of each field, in a record structure which itself is given a name.
- Form of declaration:
  - ```
    -- declaration of record data type
    type record_type_name is record
       field_name_1 : field_type_1;
       field_name_2 : field_type_2;
       -- various fields in the record
    end record;
    ```
    ```

Declaring records
declaring record types

- Declarations of field types often involve constants, for things like bounds of subrange types, or sizes of strings. Thus a common pattern for the declarations in a program is:
  - constants
    - for sizes of strings and arrays
    - for upper and lower bounds of subtypes
  - elementary data types
    - data types for single variables, or fields of records
  - structured data types
    - record structures, making use of elementary data types

Fitness club example

```
-- various constants used in data types
space        : constant := ' ';     -- ascii space
max_name_lenf   : constant := 25; -- max char in name
max_phone_lenf : constant := 10; -- max char in phone
min_age      : constant := 16;      -- minage of person
max_age      : constant := 80;      -- max age of person
min_weight : constant := 0.00;    -- min person wight
max_weight : constant := 250.00; -- max person weight

-- various types and sub-types for record declarations
subtype names  is STRING(1 .. max_name_lenf);
subtype phones is STRING(1 .. max_phone_lenf);
type sexes is (male, female);
subtype ages    is INTEGER range min_age .. max_age;
subtype weights is FLOAT range min_weight .. max_weight;

-- input/output of sex enumerated type
package sex_io is new ENUMERATION_IO( sexes );

-- declaration of record data type
type persons is record
   name : names   := ( others => space); -- name
   phone : phones := ( others => space); -- phone
   sex : sexes;     -- sex of person
   age : ages;      -- age of person
   weight: weights; -- weight of person
end record;
this_person, that_person : persons; -- various people
```

Declaring records
declaring record variables

- Once record types are declared, you can then declare variables of the record type:

- `this_person,that_person : persons;`

Declaring records
initializing records

- A default value can be specified for fields

```
field_name: f_type := (others => space);
```

- You can use an **aggregate** to set values to the fields of a record.
  - with a **positional aggregate** the values are set for each field in the order in which the fields are declared
  - with a **named aggregate** you specify the name of the field and the value it is to be given
    - fields can be in any order
    - can skip unneeded fields
  - you can mix positional and names aggregates. If you do so, the positional fields must be listed first.

Declaring records
initializing records

- Example - positional aggregate:
  - ```
    average_male : constant persons :=
    ("Mr. A Average            ",
     "          ",
     male, 25, 72.5);
    ```
- Example - named aggregate:
  - ```
    average_female : constant persons :=
    (name=>"Ms. A Average      ",
     phone=>"          ",
     sex=>female, age=>28,
     weight=>62.0);
    ```

Using records

referring to records and fields

- To refer to an entire record variable (for assignment, parameter, comparison, etc) just use its name
  - **Note:** use the name of the record *variable*, not the record *type*
- To refer to a field of a record, use record_name.field_name
  - ```
    average_male.weight
    average_female.name
    ```
    ```

Using records
operations on records

- Assignment
  - You can assign one record variable to another of identical type
    - that_person := this_person;
- Input
  - You cannot read an entire record variable in a single operation. You must read **each field separately**.
  - To input a record variable use a procedure:
    - Prompt for and get each field in turn

Using records

operations on records

```
procedure get_person_record(a_person: out persons) is

  space : constant CHARACTER := ' ';  -- a space char
  nchar : NATURAL;                    -- for GET_LINE

begin -- get_person_record
  a_person.name := (others => space); -- clear name
  PUT("Enter name "); GET_LINE(a_person.name, nchar);

  a_person.phone := (others => space); -- clear phone
  PUT("Enter phone "); GET_LINE(a_person.phone, nchar);

  PUT("Enter sex ");
  safe_get_sex(a_person.sex );

  PUT("Enter age. ");
  gen_int_input(a_person.age, min_age, max_age);

  PUT("Enter weight. ");
  gen_float_input(a_person.weight, min_weight, max_weight);
end get_person_record;
```

Using records
operations on records

- Output
  - You cannot display an entire record variable in a single operation. You must display each field separately.
  - To display a record variable use a procedure:
    - Describe and display each field in turn

Using records

operations on records

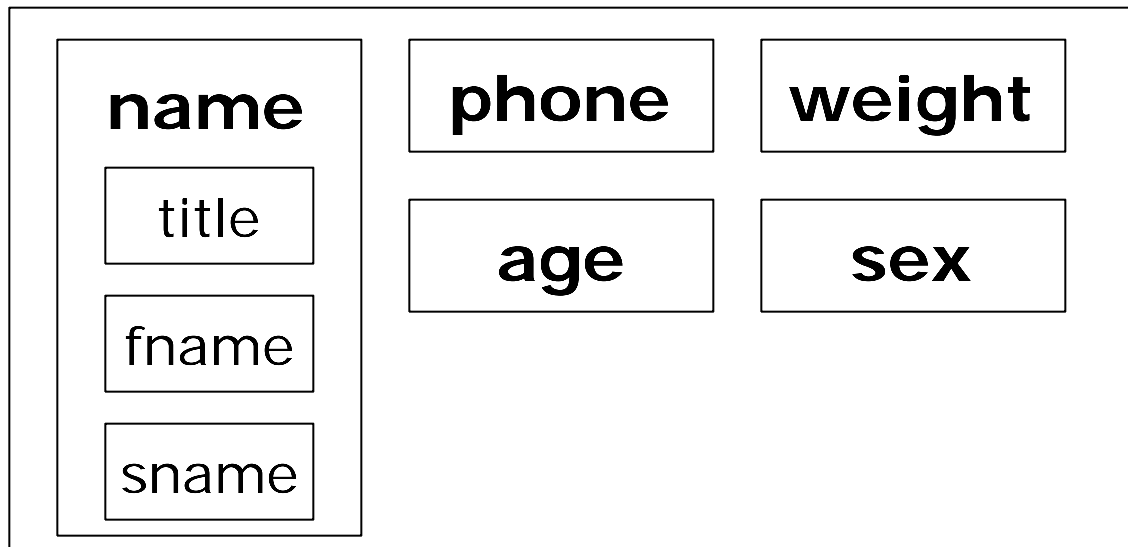
```
procedure show_person_record(a_person: in persons) is

begin
  PUT("Name ");    PUT_LINE( a_person.name );
  PUT("Phone ");   PUT_LINE( a_person.phone );
  PUT("Sex ");     PUT(a_person.sex, SET => LOWER_CASE);
  NEW_LINE;
  PUT("Age ");     PUT(a_person.age, WIDTH => 3);
  NEW_LINE;
  PUT("Weight "); PUT(a_person.weight, EXP => 0, AFT =>2);
  NEW_LINE;
end show_person_record;
```

Using records
operations on records

- **Comparisons**
  - You can compare one record variable to another of identical type using "is equal to" or "is not equal to" operators
    - `if this_person = that_person then`
  - You should use a function to compare specific fields
    - ```
      function is_heavier_than(a_person,
        another_person : persons ) return BOOLEAN is  begin
      -- is_heavier_than
        return a_person.weight > another_person.weight;
      end is_heavier_than;
      ```
  - To use this function:
    - ```
      if is_heavier_than(this_person, that_person) then
        PUT(this_person.name); PUT_LINE(" is heavier.");
      else
        PUT(that_person.name); PUT_LINE(" is heavier.");
      end if;
      ```

Hierarchical records

- The components of a record can be **any** type, including another record
- For example, a name can be a record with three components. A record of information about a person can thus contain a record for the name.



complex_persons

```
max_title_size : constant := 4;
max_fname_size : constant := 15;
max_sname_size : constant := 20;

subtype titles is STRING( 1 .. max_title_size);
subtype fnames is STRING( 1 .. max_fname_size);
subtype snames is STRING( 1 .. max_sname_size);

-- other constants & types as before
type names is record
  title : titles := ( others => space); -- title
  fname : fnames := ( others => space); -- first name
  sname : snames := ( others => space); -- surname
end record;

type complex_persons is record
  name : names;
  phone : phones := ( others => space); -- phone no
  sex : sexes;                          -- sex of person
  age : ages;                           -- age of person
  weight: weights;                      -- weight of person
end record;

a_person : complex_persons;  -- a person
```

Hierarchical records

- Refer to inner components using two dot operators
  - `a_person.name.title`
  - "the title field of the name field of the record variable a_person"
- You can have as many records within records as needed. Good style suggests limiting the number of levels.

Arrays

- An **array** is a *data structure* which groups related items together
  - related in that they record similar data about several different things
    - the mark on a test for each student in a class
    - the temperature on the hour, at each hour during a day
    - etc

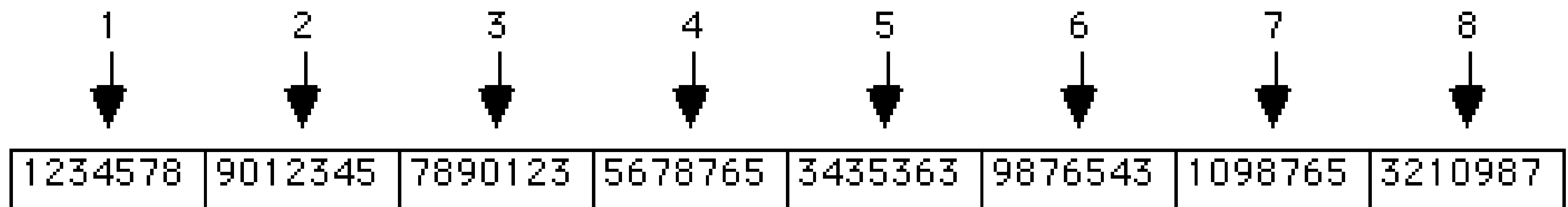
Array index vs. array element

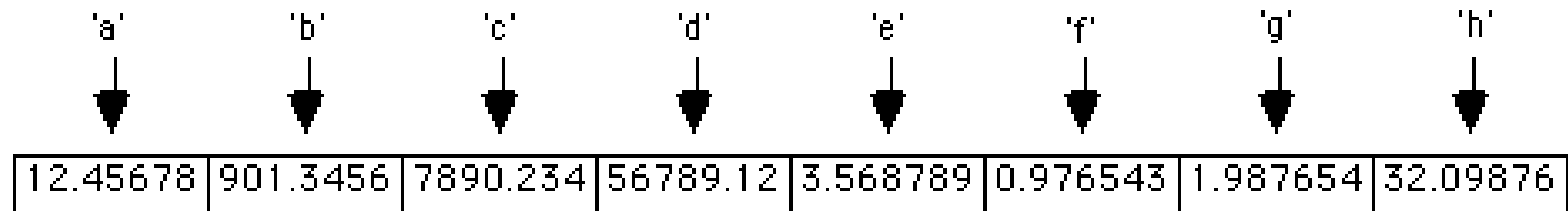
- When designing an array, you need to decide
  - what the labels are going to be
    - the array *index*
    - what *type* of value is the index?
    - what *range* of values can the index take?
    - the array index may be INTEGER, CHARACTER or any ENUMERATED TYPE
  - what type of information can go into each box.
    - the array *element* type
    - the array element type can be *any* type
  - the type of the array index **is not related** to the type of the array items

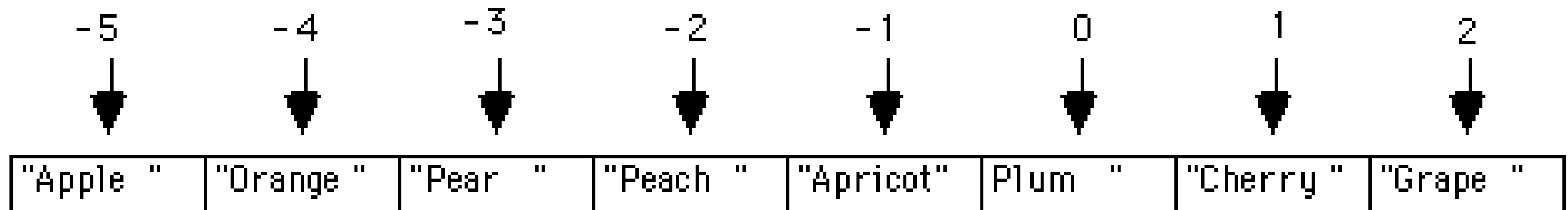
Examples

- INTEGER(1..8)
  - element type is INTEGER
  - index type is INTEGER
  - index can take 8 possible values, ranging from 1..8
- FLOAT('a'..'h')
  - element type is FLOAT
  - index type is CHARACTER
  - index can take 8 possible values, ranging from 'a'..'h'
- STRING(-5..2)
  - element type is STRING
  - index type is INTEGER
  - index can take 8 possible values, ranging from -5..2

Examples







Declaring Arrays

- An array declaration describes the *form* of the array
  - type of each element
    - can be anything
  - type and range of index
    - can be any ordinal type (INTEGER, CHARACTER, enumeration type, or any derived type or subtype of these)
  - element type **is not related** to index type

Example

```
-- various constants used in data types

max_iarr : constant := 8;     -- largest index in int array
min_farr : constant := 'a';   -- low index in float array
max_farr : constant := 'h';   -- high index in float array

-- type declarations

subtype STRING8 is STRING (1 .. 8);

type int_8_array is array (1 .. max_iarr) of INTEGER;
type float_arrays is array (min_farr..max_farr) of FLOAT;
type str_arrays is array (-5 .. 2) of STRING8;
type small_arrays is array ('a' .. 'c') of FLOAT;
```

The declaration gives a name to the array type

then can declare variables of that array type

```
arr1 : int_8_array;
arr2 : float_arrays;
arr3 : str_arrays;
```

Initializing arrays

- An array **aggregate** can be used to list initial values for items in an array variable
  - using positional notation
  - using explicit index references
- Examples:

```
-- init array coord1 using a positional list
coord1 : small_arrays := (1.2, 2.4, 3.6);

-- init array coord1 using explicit index references
coord2 : small_arrays := ('c'=>3.6, 'b'=>2.4, 'a'=>1.2);
```

- You can specify a default for **other** items in array that are not explicitly initialized

```
-- init array coord1 using other
coord3 : small_arrays := ('b'=>5.2, others => 0.0);
```

Using Arrays

- **Referring to arrays**
  - To refer to an entire array variable (for assignment, parameter, comparison, etc) just use the array variable name.
    - **Note:** refer to the array *variable*, not the array *type*
  - To refer to an individual element in the array, you need to specify the array variable name and the index value for the element you want.

```
PUT(coord1('b'));
total := coord1('a') + coord1('b') + coord1('c');
PUT(arr3(-2));
```

Example 1(2)

```
--------------------------------------
-- letters.ada - count lower-case
-- letters read from user
-- written by: Lawrie Brown / 7/29/94
--------------------------------------
with TEXT_IO;
procedure lc_letter_freq is
  package int_io is new TEXT_IO.INTEGER_IO (INTEGER);
  use TEXT_IO, int_io;

  subtype lc_letters is CHARACTER range 'a' .. 'z';
  type freq_table is array (lc_letters) of INTEGER;

  count : freq_table := (others => 0); -- freq counts
  char : CHARACTER; -- letter just read

begin -- lc_letter_freq
  -- give some instructions
  PUT_LINE ("This program will count lc letters");
  PUT_LINE ("Enter lines of text, finish with '.'");
  -- get some characters from the user
  loop
    GET(char);
    exit when char = '.';
    if char in lc_letters then
      count(char) := count(char) + 1;
    end if;
  end loop;
```

Example 2(2)

```
  -- now show how many letters were read
  NEW_LINE;
  PUT_LINE("Letter Freq");
  NEW_LINE;

  for t in lc_letters loop
    PUT(t); PUT(count(t), width=>11); NEW_LINE;
  end loop;
end lc_letter_freq;
```

Array Attributes

- Array attributes give information about the array type or array variable.

```
-- various constants used in data types
max_iarr : constant := 8;     -- largest index in int array
min_farr : constant := 'a';   -- low index in float array
max_farr : constant := 'h';   -- high index in float array
-- type declarations
subtype STRING8 is STRING (1 .. 8);
type int_8_array is array (1 .. max_iarr) of INTEGER;
type float_arrays is array (min_farr..max_farr) of FLOAT;
type str_arrays is array (-5 .. 2) of STRING8;
type small_arrays is array ('a' .. 'c')
arr1 : int_8_array;
arr2 : float_arrays;
arr3 : str_arrays;
```

int_8_array’**FIRST**	1	float_arrays’**LAST**	‘h’
str_arrays’**RANGE**	-5 .. 2	arr3’**LENGTH**	8
small_arrays’RANGE	‘a’..’c’	small_arrays’LENGTH	3
freq_table’RANGE	lc_letters ‘a’..’z’	count’RANGE	lc_letters

```
subtype lc_letter is CHARACTER range 'a' .. 'z';
type freq_table is array (lc_letters) of INTEGER;
count : freq_table := (others => 0); -- freq counts
```

Array Attributes

- Array attributes in loops
  - A useful application of array attributes is setting the bounds of loop control variables:

```
for t in count'RANGE loop
  PUT(t);
  PUT(count(t), width=>11); NEW_LINE;
end loop;
```

  - This causes "t" to take each index value in turn for the array "count", *regardless* of the index type and range.

Operation on arrays

- Assignment
  - You can assign one entire array variable to another of the same type
    - `coord1 := coord2;`
- Comparison
  - You can compare one array variable to another of the same type
    - Compares item by item
    - ```
      if (coord1 /= coord2) then
          PUT("They are different");
      end if;
      ```
      ```

Operation on arrays

- Arrays as Parameters
  - You can use an array variable as an actual parameter to a procedure or function.
  - The amount of flexibility you have in doing so depends on how the formal parameter was declared in the subprogram:
    - if an *unconstrained array type* is used for the formal parameter, then **any** variable based on that type may be passed as an actual parameter.
    - if a constrained array type is used for the formal parameter, then **only** variables of **that** type may be passed as an actual parameter

Unconstrained Arrays

- We have only used **constrained array types** so far
  - the size of array was specified in type declaration, when the range of index values was specified
- Ada also provides **unconstrained array types**
  - element type is specified in type declaration
  - index type is specified in type declaration
  - range of index values (ie size) is **not** specified in type declaration
  - specify range of index values in *variable* declarations
- Examples:

```
type int_u_array is array (INTEGER range <>) of INTEGER;
small_int_array : int_u_array (1 .. 3); -- just 3 items
big_int_array : int_u_array (1 .. 100); -- 100 items
type char_count is array (CHARACTER range <>) of INTEGER;
subtype digit_count is char_count('0' .. '9');
uc_counts : char_count ('A' .. 'Z');
dig_counts : digit_count;
```

- STRING is an **unconstrained array type**

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

```
-------------------------------------------------------------
-- sumarr.ada - sum array elements test program -
-- written by: Lawrie Brown / 29 Jul 94
-------------------------------------------------------------
with TEXT_IO;
procedure sum_array is
  package int_io is new TEXT_IO.INTEGER_IO (INTEGER);
  use TEXT_IO, int_io;
  type int_u_array is array (INTEGER range <>) of INTEGER;
  small_int_array: int_u_array (1..3) :=(18, 15, 4);
  big_int_array: int_u_array (1..6) :=(1,4,9,16,25,36);
  -----------------------------------------------------------
  -- function to sum the the items in an array
  function sum_iarr (in_arr:in int_u_array) return INTEGER is
    the_total : INTEGER := 0; -- the running total
  begin -- sum_iarr
    for i in in_arr'RANGE loop
      the_total := the_total + in_arr(i);
    end loop;
    return the_total;
  end sum_iarr;
  -----------------------------------------------------------
begin -- sum_array
    PUT("sum of small_int_array is ");
    PUT(sum_iarr(small_int_array), width=>4); NEW_LINE;
    PUT("sum of big_int_array is ");
    PUT(sum_iarr(big_int_array), width=>4); NEW_LINE;
end sum_array;
```

Multi-dimensional Arrays

- Often we have information in tabular form
  - tables of data
  - matrices
- Use a **multi-dimensional array** to represent such information
  - items indexed by several subscripts
  - eg row and column for 2D arrays
- You can have as many dimensions as wanted
  - extend declaration to include required index ranges
  - extend references to include required indices

Multi-dimensional Arrays

```
-- type declaration for higher dimensional arrays
type CUBE6 is array (1..6, 1..6, 1..6) of CHARACTER;

-- variable declaration for higher dimensional arrays
tictactoe_3d : CUBE6;

-- reference to element in multi-dimensional array
PUT(tictactoe_3d(2,3,4));
```

- Ada does not limit the number of dimensions. In practice, though, the limit is 3 or 4 dimensions
  - humans have trouble visualizing more than 3D
  - more subscripts often means more errors
  - more dimensions means lots of memory
- By far the most common multi-dimensional arrays are *two dimensional arrays*.

Two dimensional arrays

- You can use two-dimensional arrays to represent tables, matrices etc.
- Ex: Represent distance between cities in a table

	Amsterdam	Berlin	London	Madrid	Paris	Rome	Stockholm
Amsterdam	0	648	494	1752	495	1735	1417
Berlin	648	0	1101	2349	1092	1588	1032
London	494	1101	0	1661	404	1870	1807
Madrid	1752	2349	1661	0	1257	2001	3138
Paris	495	1092	404	1257	0	1466	1881
Rome	1735	1588	1870	2001	1466	0	2620
Stockholm	1417	1032	1807	3138	1881	2620	0

```
-- various constants used in data types
max_dist : constant := 40077; -- max distance on earth

-- type declarations
type distances is range 0 .. max_dist;
type city is (Amsterdam, Berlin, London, Madrid, Paris, Rome, Stockholm);
type distance_table is array (city, city) of distances;

-- distances between various European cities
inter_city : distance_table :=
  -- Amst, Berl, Lond, Madr, Pari, Rome, Stock
   (( 0, 648, 494, 1752, 495, 1735, 1417),    -- Amsterdam
    ( 648, 0, 1101, 2349, 1092, 1588, 1032),  -- Berlin
    ( 494, 1101, 0, 1661, 404, 1870, 1807),   -- London
    (1752, 2349, 1661, 0, 1257, 2001, 3138),  -- Madrid
    ( 495, 1092, 404, 1257, 0, 1466, 1881),   -- Paris
    (1735, 1588, 1870, 2001, 1466, 0, 2620),  -- Rome
    (1417, 1032, 1807, 3138, 1881, 2620, 0)); -- Stockholm

-- distances I have traveled between various cities
traveled : distance_table := (others => (others => 0));
your_travel : distance_table;
```

Using 2-D arrays

- You need to declare two index types, for the two dimensions
  - Each index type may be any discrete type (integer, character, enumerated type)
    ```
    type city is (Amsterdam, Berlin, London,
                  Madrid, Paris, Rome, Stockholm);
    type distance_table is array (city, city) of distances;
    ```
  - To reference elements of a 2-D array variable, use both index values:
    ```
    PUT(inter_city(Berlin, Rome));
    traveled(Stockholm, London) := 1807;
    ```

Using 2-D arrays

- Nested for loops are often used to process 2D arrays
- ```
  -- write out the table
  for from in Amsterdam .. Stockholm loop
    -- write one line of the table
    for to in Amsterdam .. Stockholm loop
      PUT(inter_city(from, to), width=>6);
    end loop;
    NEW_LINE;
  end loop;
  ```
- You can assign one entire array variable to another of the same type
  - `your_travel := traveled;`
  ```

Hourly temperatures for a week

```
----------------------------------------------------------
-- temperature.ada - process week of hourly temperatures
-- Skansholm p 305-306 adapted by: Lawrie Brown / 29 Jul 94
----------------------------------------------------------
with TEXT_IO;
procedure temperature is
  type DAYS is (Monday, Tuesday, Wednesday, Thursday,
                  Friday, Saturday, Sunday); -- days of week
  type HOURS is range 0..23; -- hours in a day
  type TEMPS is digits 3 range -99.9 .. 99.9; -- temps

  type MEASUREMENT_TABLE is array (DAYS, HOURS) of TEMPS;

  package DAY_IO is new TEXT_IO.ENUMERATION_IO (DAYS);
  package HOUR_IO is new TEXT_IO.INTEGER_IO (HOURS);
  package TEMP_IO is new TEXT_IO.FLOAT_IO (TEMPS);
  use TEXT_IO, DAY_IO, HOUR_IO, TEMP_IO;

  num_days : constant := 7; -- num days in week

  measurements : MEASUREMENT_TABLE; -- table of temps
```

Hourly temperatures for a week

```
procedure Read_Temps(Tab : out MEASUREMENT_TABLE) is
begin -- Read_Temps
  -- read values into table
  for d in DAYS loop
    PUT("Enter the temperature for ");
    PUT(d); NEW_LINE;
    -- get all values for one day
    for h in HOURS loop
      GET(Tab(d, h));
    end loop;
    SKIP_LINE;
  end loop;
end Read_Temps;
```

Hourly temperatures for a week

```
 procedure Write_Mean(M_tab : in MEASUREMENT_TABLE) is
  mean : TEMPS;
  begin -- Write_Mean
    -- write heading
    PUT_LINE("hr mean temp"); NEW_LINE;
    for h in HOURS loop
      -- average measurements for this hour
      mean := 0.0;
      for d in DAYS loop
        mean := mean + M_tab(d, h);
      end loop;

      -- compute mean
      mean := mean / TEMPS(num_days);

      -- display result
      PUT(h, width=>2); PUT(mean, exp=>0, fore=>7, aft=>1);
      NEW_LINE;
    end loop;
  end Write_Mean;

begin -- temperature
  Read_Temps(measurements);
  Write_Mean(measurements);
end temperature;
```

Matrices

- In mathematics, we have M row by N column matrices.
- Represent them using 2D arrays in Ada
- Sometimes they are implemented with constrained array types
  - then restricted to only using that size matrix eg 3x4
- More often use **unconstrained array types**
  - Then can write procedures for any size matrix

Matrices

```
type MATRIX is array (POSITIVE range <>,
                      POSITIVE range <>) of INTEGER;

P35, Q35, R35 : MATRIX(1..3, 1..5); -- 3 by 5 matrices
X24, Y24, Z24 : MATRIX(1..2, 1..4); -- 2 by 4 matrices

-- ADD two matrices together
function ADD (A, B : in MATRIX) return MATRIX is
  C : MATRIX (A'RANGE(1), A'RANGE(2));
begin -- ADD
  for I in A'RANGE(1) loop
    for J in A'RANGE(2) loop
      C(I,J) := A(I,J) + B(I,J);
    end loop;
  end loop;
  return C;
end ADD;

-- call procedure with suitable arrays
R35 := ADD(P35, Q35);
Z24 := ADD(X24, Y24);
```

Arrays of arrays

- With 1D arrays, we have said that each element may be of **any** type
  - hence could have each element being itself an array
  - thus have an array of arrays
- 2D array vs. Array of arrays
  - Where a 2D array has two indices, and you specify them both at once to pick out the element you want, an array of arrays has one index into each array.

Arrays of arrays

- Two dimensior

	1	2	3	4	5
1	1	2	4	8	16
2	1	3	9	27	81
3	1	5	25	125	625

```
type MAT45 is array (1 .. 4, 1 .. 5) of INTEGER;
R : MAT45;
```

- Array of arrays

	1	2	3	4	5
1	1	2	4	8	16

2	1	3	9	27	81

3	1	5	25	125	625

```
type ROW5 is array (1 .. 5) of INTEGER;
type MATRIX35 is array (1 .. 3) of ROWS;
X : MATRIX35;
```

An array of STRINGs is a special case of arrays of arrays:

```
subtype NAMES is STRING (1 .. 20);
type REGISTERS is array (POSITIVE range <>) of NAMES;
```

Using arrays of arrays

- To refer to an element of an array of arrays, you must specify indexes separately
- `X(2) entire second row of array X`
  `X(2)(3) a single element of row 2 of X`
- Note the difference from 2D matrix reference.

Advantages and disadvantages

- Advantages of arrays of arrays
  - can refer to a single row as a whole
    - X(2)
  - can refer to a slice of the array
    - X(2..3) means rows 2 and 3
    - X(2..3)(1) means 1st item of rows 2 and 3
    - X(4)(2..4) means items 2,3,4 of row 4
- Disadvantages of arrays of arrays
  - only first index may be unconstrained,
  - component type **must** be of known size

```
type ROW5 is array (1 .. 5) of INTEGER;
type MATRIXN5 is array (POSITIVE range <>) of ROW5;
```

Arrays of Records

- An array item may be of **any** type
  - can have an array with each item being a record
- Purpose:
  - often want to represent a collection of information
  - a simple database of information on something
    - each item has a number of attributes of interest => use a record
    - need this information for many items => use an array of records

Example

- Result list for a sporting event:
  - several items of information about each competitor
    - number, name, club, time to complete the event
    - => record
  - several competitors
    - => array

	id_number	name	club	run_time
1	42	"Joe Bloggs "	"Belconnen "	168.5
2	86	"Wendy Brown "	"Woden "	149.0
3	23	"John Smith "	"Tuggeranong "	151.5
4	01	"Sally Black "	"Dickson "	178.6

Example

```
max_field : constant := 50;

type Number is range 1 .. 1000;

subtype String20 is String(1..20);

type Time is digits 7 range 0.0 .. 600.0;

type Competitor is record
   id_number : Number;
   name : String20;
   club : String20;
   run_time : Time;
end record;

type Result_list is array (1 .. max_field) of Competitor;

race1, race2 : Result_list;
```

Using arrays of records

- Declarations
  - constants: subtype limits, array sizes
  - elementary types: subtypes, enumeration types, field types
  - record data type
  - array type: array of that record type
  - array variables

Using arrays of records

- Referring to portions of the array:
  - can refer to entire database (array of all records) for assignment, as an actual parameter, etc
    - eg `race2 := race1;`
  - can refer to one item (a record in the database)
    - eg race1 is an array of records, so race1(J) is a record
    - eg `Display_Competitor(race1(J));`
  - can refer to a field of one item in database

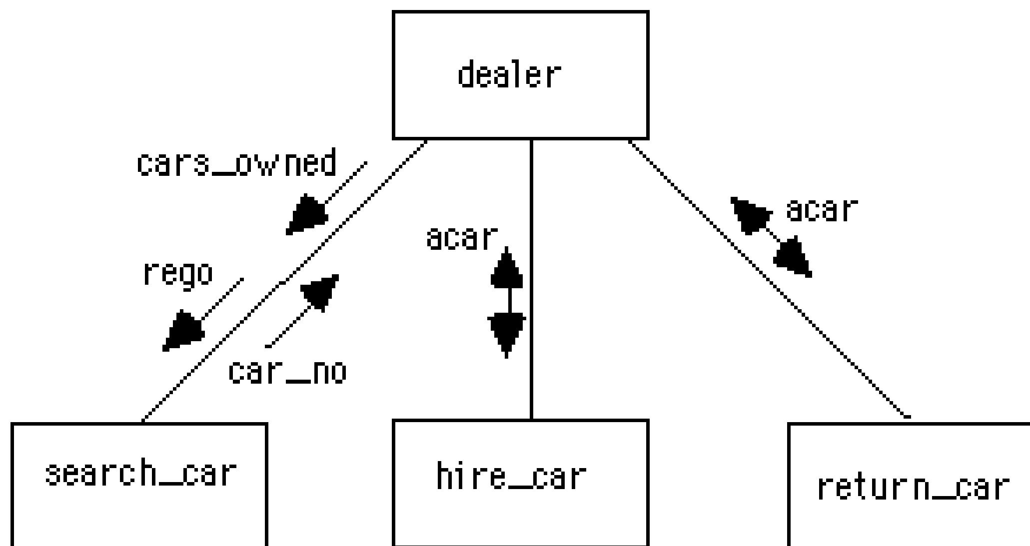
```
race1(3).id_number := 23;
get (race1(I).id_number);
race1(J).name := "Joe Bloggs          ";
PUT(race1(J).club); GET(race1(next).run_time);
```

Array of Records - Example

- **Problem**
  - A system is needed that will allow a car rental company to record details of its cars.
  - The program will allow the information to be updated as cars are hired out and returned.
  - The following information is required for each car:
    - registration number, make, year, weight, power
    - name, client number, and address of the person who is currently hiring it.

Program Design

dealer.ada




```
procedure dealer is

  various types and global variables

  function search_car(c, rego)
    ...
  begin
    ...
  end;

  procedure hire_car(acar)
    ...
  begin
    ...
  end;

  procedure return_car(acar)
    ...
  begin
    ...
  end;

begin -- of dealer
  search_car for wanted rego
  if not hired then hire_car
  if hired then return_car
end dealer
```