

COMP205 IMPERATIVE LANGUAGES

5. COMPOUND (HIGHER LEVEL) DATA TYPES I --- ARRAYS

- 1) Introduction to higher level types
- 2) Arrays and their declaration
- 3) Assigning values to array elements
- 4) Operations on arrays
- 5) Ada array attributes

INTRODUCTION TO HIGHER LEVEL TYPES

- The available basic types can be extended by adding *higher level* types.
- Higher level types are (usually user-defined) data types made up of basic types and other user-defined higher level types.
- There are some standard higher level types available in most imperative languages. These include:
 - 1) Arrays
 - 2) Strings
 - 3) Enumerated types
 - 4) Records/structures
 - 5) Unions

TYPE CONSTRUCTORS

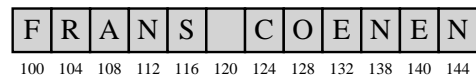
- Higher level types are declared using *type constructors* which "construct" new types out of zero or more existing types.
- Higher level types can either:
 - a) Be used directly and anonymously in a declaration
 - b) Be bound to a type name first using a type declaration statement (the type name can then be used later in the code to create instances of that type).
- Ada type declaration statements are of the form:

```
<type_name> 'is' <type_def>
```
- C type declaration statements are of the form:

```
`typedef' <type_def & type_name>
```

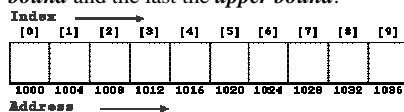
ARRAYS

- The simplest (and oldest) form of higher level data type is the *array*.
- An array is a series of items all of the same type.
- Each item is assigned a location in memory.
- At each location a value may be held.



Features:

- 1) Items in arrays are called *elements*.
- 2) Specific elements in an array can be identified through the use of an *index*.
- 3) Indices can be:
 - a) *Discrete* - for example integers, characters or *enumerated* values (also referred to as linear indexing).
 - b) *Non-discrete* - indexing using non-linear types (e.g. floats) (also referred to as associative indexing).C and Ada only support linear indexing.
- 4) The first index in an array is called the *lower bound* and the last the *upper bound*.



ARRAY STRUCTURE

ARRAY DECLARATIONS

- When declaring arrays we are doing two things:
 - 1) Declaring the type of the array
 - 2) Declaring the nature of the index

APPROACHES TO DEFINING INDEXES

- 1) Assume index is an integer type and define either:
 - Only the upper bound (assumes the lower bound is fixed), or
 - both the lower and upper bound.
- 2) Specify the data type of the index and provide lower and upper bounds as appropriate.
- 3) Define the index only in terms of a data type, in which case the index range for the array is assumed to be equivalent to the range of the data type in question.

C only supports the first, Ada supports all three.

THERE NOW
FOLLOWS A SERIES
OF EXAMPLES
DEFINING VARIOUS
ARRAY DATA ITEMS!

DECLARING ARRAYS IN ADA

In Ada, to declare arrays, we use an array data declaration statement of the form:

```
NAME ':: array' <index_definition> 'of'  
TYPE ';' ;
```

or an array type declaration:

```
'type' TYPENAME 'is' <index_definition>  
'of' TYPE ';' ;
```

EXAMPLE 1 (Ada)

- Assume index is an integer and define lower and upper bounds.
- Format:

```
'(' LOWER_BOUND '..' UPPER_BOUND ')'
```

- Example declarations:

```
ARRAY1 : array (1..9) of integer;  
N: integer;  
ARRAY2 : array (N..N+4) of integer;
```

EXAMPLE 2 (Ada)

- Approach: Index type specified by a lower and upper bound.
- Format:

```
'(' INDEX_TYPE 'range' LOWER_BOUND  
  '..' UPPER_BOUND ')'
```

- Example declaration:

```
ARRAY3 : array (CHARACTER range  
  a..z) of integer;
```

EXAMPLE 3 (Ada)

- Approach: Particular data type. Format:

```
'(' INDEX_TYPE ')'
```

- Example declaration:

```
type ITEMS_T is range 0..100;  
ARRAY4 : array (ITEMS_T) of integer;
```

EXAMPLE 4 (C)

- Assumes:
 - Index is an integer type.
 - Lower bound is 0.
- Declaration format:

```
TYPE NAME '[' NUM_ELEMENTS ']' ';' ;'
```
- Example declaration: `int list1[10];`
- Declares an array of 10 elements with an index range of 0 to 9 inclusive.
- C array declaration is more succinct but harder to read (then the Ada version).

MANIPULATION OF ARRAYS

- Assigning values
- Example programs
- Operations on complete arrays
- Slices
- Conformant array parameters
- Ada array attributes

ASSIGNMENT OF VALUES TO ARRAY ELEMENTS

- Given knowledge of the index for an array element we can assign a value to that element (or change its value) using an "assignment" operation.
- Examples (Ada and C):

```
LIST1(1) := 1;  
  
list1[0] = 1;
```

COMPOUND VALUES

- Some imperative languages support the concept of *compound values* which allow simultaneous assignment to instances of higher level data types.
- Array compound values allow all the elements of an array to be assigned to simultaneously.
- Both C and Ada support compound values (in Ada they are called *aggregates*).
- However, in C their usage is limited to initialisation, while in Ada array aggregates can be used in any assignment statement

- Ada Example

```
type MY_ARRAY_T is array (1..10) of integer;  
LIST : MY_ARRAY_T := (5, 4, 3, 2, 1, 0, 0, 0, 0, 0);
```

- C Example

```
int list[10] = {5, 4, 3, 2, 1, 0, 0, 0, 0, 0};
```

- Ada also supports an alternative to the above:

```
LIST := (5, 4, 3, 2, 1, other => 0);
```

ADA ARRAY EXAMPLE PROGRAM

```
procedure EXAMPLE is  
  START: constant := 2;  
  IA: array (integer  
    range START..START+4  
    of integer := (2,4,6,8,10);  
begin  
  put("Array comprises = ");  
  put(IA(2)); put(", ");  
  put(IA(3)); put(", ");  
  put(IA(4)); put(", ");  
  put(IA(5)); put(", ");  
  put(IA(6)); new_line;  
end EXAMPLE;
```

- Note that the above can be made more succinct if some form of loop construct was used to step through the array.

C ARRAY EXAMPLE PROGRAM

```
void main(void) {
int ia[5] = {2,4,6,8,10};

printf("Size of array = %d (bytes)\n",sizeof(ia));
printf("Num elements = %d\n",sizeof(ia)/sizeof(ia[0]));
printf("Array comprises = %d, %d, %d, %d, %d\n", ia[0],
ia[1], ia[2], ia[3], ia[4]);
printf("Address ia[0] = %d\n",&ia[0]);
printf("ia = %d\n",ia);
}
```

```
Size of array = 20 (bytes)
Num elements = 5
Array comprises = 2, 4, 6, 8, 10
Address ia[0] = 2063808408
ia = 2063808408
```

OPERATIONS ON COMPLETE ARRAYS

- Some languages, such as Pascal and PL/1 (but not Ada or C), allow operations on complete arrays. In PL/1 we can write:

```
IA = 0;
```

- This will make all elements in the array IA equal to 0.
- Alternatively, some languages (Pascal, Ada) support assignment of one array to another array:

```
LIST1 := LIST2
```

Where LIST1 and LIST2 are two arrays of the same type (but not supported by C).

SLICES

- A *Slice* is a sub-array of an array. The process of slicing returns a sub-array (supported by Ada but not C). Thus:

```
IA: array (integer range
START..START+4) of character
:= ('a', 'b', 'c', 'd', 'e');
```

- The statement

```
IA_SUB = IA(START+1..START+3)
```

would assign the sub array ('b', 'c', 'd') to the variable IA_SUB.

```
procedure EXAMPLE is
START: constant := 2;
type MY_ARRAY_T is array (integer range START
.. START+4) of integer;
LIST1: MY_ARRAY_T := (2,4,6,8,10);
LIST2, LIST3: MY_ARRAY_T;

procedure OUTPUT_ARRAY (LIST: MY_ARRAY_T) is
begin
--- Output array contents ---
end OUTPUT_ARRAY;

begin
OUTPUT_ARRAY(LIST1);
LIST2 := LIST1;
OUTPUT_ARRAY(LIST2);
LIST3 := LIST1(2..4) & LIST2(2..3);
OUTPUT_ARRAY(LIST3);
end EXAMPLE;
```

SLICES EXAMPLE

Note: all arrays are of the same type.

CONFORMANT ARRAY PARAMETERS

- To be compatible two arrays must be of the same size and type.
- This means that general purpose array procedures can only work for arrays of a pre-specified size and type.
- To get round this we can allow formal array parameters to be given bounds that can vary depending on the nature of the actual parameter at the time the procedure is called.
- Such parameters are referred to as *conformant array parameters*.
- Supported by Pascal (but not Ada or C).

ADA ARRAY ATTRIBUTES

- T'FIRST Gives first index value for array type T
- T'LAST Gives last index value for array type T
- T'LENGTH Gives number of components in array type T
- Note that instead of an array type an array name may equally well be used.

```
procedure EXAMPLE is
  START: constant := 2;
  IA: array (integer range START..START+4) of
    character := ('a', 'b', 'c', 'd', 'e');
begin
  put("IA'first = ");
  put(IA'first);
  new_line;
  put("IA'last = ");
  put(IA'last);
  new_line;
  put("IA'length = ");
  put(IA'length);
  new_line;
end EXAMPLE ;
```

SUMMARY

- 1) Introduction to higher level types
- 2) Arrays and their declaration
- 3) Assigning values to array elements
- 4) Operations on arrays
- 5) Ada array attributes