

COMP205 IMPERATIVE LANGUAGES

6. COMPOUND (HIGHER LEVEL) DATA TYPES II --- MORE ON ARRAYS

- 1) Constrained (static) and unconstrained (dynamic) arrays.
- 2) Flexible arrays
- 3) Multi-dimensional arrays
- 4) Arrays of arrays
- 5) Lists, sets, bags, Etc.
- 6) Strings

CONSTRAINED AND UNCONSTRAINED ARRAYS

- A **constrained array** is an array where the index is specified (and hence the number of components is specified).
- We say that the bounds are static, hence constrained arrays are sometimes referred to as **static** arrays.
- Many imperative languages (including Ada but not C) also support some mechanism for declaring unconstrained arrays.
- In this case we say that the **bounds** are **dynamic**.
- Ada makes use of the symbol <> to indicate an unconstrained array:

```
type LIST1 is array (INTEGER range <>) of FLOAT;  
L1 : LIST1 (START..END);
```

DYNAMIC ARRAYS

- Although C does not support the concept of unconstrained arrays, however it does provide facilities to delay the declaration of an upper bound of an array till run time, i.e. the upper bound is declared dynamically hence such an array is referred to as a **dynamic** array.
- Two library functions `malloc` and `free` are used.
- The `malloc(<size>)` function obtains a block of memory (for the array) according to the parameter `<size>`. (Note: The type of this parameter is system dependent, but is usually an `int` or an `unsigned int`).
- The `free` function releases a block of memory.

```
int num = 4;  
void main(void) {  
    int *numPtr = NULL;  
  
    numPtr = (int *) malloc(sizeof(int)*num);  
  
    /* Initialise. */  
  
    numPtr[0] = 2; numPtr[1] = 4;  
    numPtr[2] = 6; numPtr[3] = 8;  
  
    /* Output. */  
  
    printf("Array = %d, %d, %d, %d\n",  
        numPtr[0], numPtr[1],  
        numPtr[2], numPtr[3]);  
  
    /* End */  
  
    free(numPtr);  
}
```

C
DYNAMIC
ARRAY
EXAMPLE

OTHER TYPES OF ARRAY

- Apart from the standard array forms described earlier (static or constrained, and dynamic or unconstrained) we can identify a number of alternative forms of array which are a feature of particular imperative languages.
- These include:

- 1) Flexible arrays
- 2) Multi-dimensional arrays
- 3) Arrays of arrays
- 4) Lists

FLEXIBLE ARRAYS

- A powerful feature associated with some imperative languages is the ability to dynamically "resize" an array after it has been declared, i.e. during run-time.
- Such an array is referred to as a **flexible** array.
- Neither Ada or C support flexible arrays (Algol'68 does).
- A similar effect can be achieved in C using the built-in functions `malloc` and `realloc`.
- `realloc` extends or contracts the memory space available for a previously declared array.

```

int num = 4;
void main(void) {
int *numPtr = NULL;
/* Allocate memory and initialise. */
numPtr = (int *) malloc(sizeof(int)*num);
numPtr[0] = 2; numPtr[1] = 4;
numPtr[2] = 6; numPtr[3] = 8;

--- Output code ---
/* Reallocate memory and reinitialise. */
num = 3;
numPtr = (int *) realloc(numPtr,sizeof(int)*num);
numPtr[0] = 1; numPtr[1] = 3; numPtr[2] = 5;

--- Output code ---
free(numPtr);
}

```

**C
FLEXIBLE
ARRAY
EXAMPLE**

MULTI-DIMENSIONAL ARRAYS

- Multi-dimensional arrays are arrays where the elements have more than one index.
- In the case of two-dimensional arrays these can be thought of as comprising rows and columns, where the row number represents one index and the column number a second index.
- Two-dimensional arrays are therefore useful for storing tables of information.

		Columns	
		1	2
Rows	1	1	2
	2	3	4
	3	5	6

```

with CS_IO ; use CS_IO ;

procedure EXAMPLE is
  type TWO_D_ARRAY_T is array (1..3, 1..2)
    of integer;
  IA: TWO_D_ARRAY_T := ((1, 2), (3, 4), (5, 6));
begin
  put(IA(1,1));
  put(IA(1,2));new_line;
  put(IA(2,1));
  put(IA(2,2));new_line;
  put(IA(3,1));
  put(IA(3,2));new_line;
end EXAMPLE ;

```

**ADA 2-D
ARRAY
EXAMPLE**

• Note that the "row" index is declared first, then the "column" index.

```

void main(void) {
int ia[3][2] = {{1, 2}, {3, 4}, {5, 6}};

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

```

**C 2-D
ARRAY
EXAMPLE**

```

Size of array = 24 (bytes)
Num elements = 6
Array comprises = 1, 2, 3, 4, 5, 6

```

ARRAYS OF ARRAYS

Some imperative languages (including Ada, but not C) support *arrays of arrays*. The distinction between *arrays of arrays* and *multi-dimensional* is that in the second case the result need not be "rectangular".

```

procedure ADA_EXAMPLE is
  type A is array (1..2) of integer;
  A1: A:= (1, 2);
  A2: A:= (3, 4);
  A3: A:= (5, 6);
  type A_OF_A is array (1..3) of A;
  IA: A_OF_A:= (A1, A2, A3);
begin
  put(IA(1)(1));
  put(IA(1)(2));new_line;
  put(IA(2)(1));
  put(IA(2)(2));new_line;
  put(IA(3)(1));
  put(IA(3)(2));new_line;
end ADA_EXAMPLE ;

```

LISTS

- Lists** (or sequences) can be considered to be special types of arrays but:
 - with an unknown number of elements, and
 - without the indexing capability.
- Lists are popular in logic and functional languages but not in imperative languages.

SETS

- A *set* is a group of (distinct) elements, all of the same type, which are all possible values of some other type referred to as the *base type*.
- The relationship is similar to that of an Ada sub-type to its "super-type", e.g. positive integers to integers.
- The distinction between an array and a set is that the elements are not ordered (indexed) in anyway.
- The number of elements in a set is referred to as its *cardinality*.
- The only operations that can be performed on sets are "set operations", e.g. member, union, intersection, etc.
- Neither Ada or C feature sets, however Pascal and Modula-2 do.

PASCAL SET EXAMPLE

```
program SET_EXAMPLE (output);
  type
    SOMEBASE_T = 0..10;
    SOMESET_T = set of SOMEBASE_T;
  var
    SET1, SET2 : SOMESET_T;
begin
  SET1 := [1, 2, 5];
  SET2 := [6, 8, 9];

  --- More code in here ---

end.
```

BAGS (MULTISETS)

- Bags (multisets) are similar to sets except that they can contain an element more than once and record how many times a value has been inserted.
- The primary operations on bags are "insert value" and "remove value" (as opposed to the union, intersection, etc. operations found in sets).
- Very few imperative languages feature bags.

STRINGS

- A further type of array is the *string*.
- A string is a sequence of characters usually enclosed in double quotes.
- For this reason strings are sometimes referred to as a *character arrays*.
- As such we can use standard array operations on strings:
 - We can access any character in a string using an index.
 - Where supported (Ada) we can access slices of strings.
- In C the last member of the array must always be the *null terminator* '\0' (note single quotes).

STRING DECLARATIONS

- In C we declare a 15 character string, name, thus:

```
char name[15];
```

i.e. as an array of characters.

- Ada provides the basic data type *string*:

```
NAME: string(1 .. 15);
```

- Note the similarity with array declarations.

OPERATIONS ON STRINGS

OPERATION	Ada	C
Concatenation	S1 & S2	strcat(s1,s2)
Comparison	S1 = S2	strcmp(s1,s2)
Copy	S1 := S2	strcpy(s1,s2)

- Note: The C *strcpy* function does not require *s1* and *s2* to be of the same length (this is not the case when using the Ada := operator).

```

with CS_IO ; use CS_IO ;

procedure EXAMPLE is
  NAME1: string(1..5) := "Henri";
  NAME2: string(1..15);
begin
  NAME2(1..6) := "E. Bal";
  put("Name = "); put(NAME1);
  put(" "); put(NAME2(1..6));
  new_line;
  put("Initials = ");
  put(NAME1(1)); put(". ");
  put(NAME2(1..4)); put(".");
  new_line;
end EXAMPLE ;

```

ADA STRING EXAMPLE

```

Name = Henri E. Bal
Initials = H. E. B.

```

```

#include <stdio.h>

void main(void) {
  char name1[] = "Dick";
  char name2[15];

  strcpy(name2, "Grune");
  printf("Name = %s %s\n", name1, name2);
  printf("Initials = %c. %c.\n", name1[0], name2[0]);
  printf("Address of name1[0] = %d\n", &name1[0]);
  printf("name1 = %d\n", name1);
}

```

C STRING EXAMPLE

```

Name = Dick Grune
Initials = D. G.
Address of name1[0] = 2063808416
name1 = 2063808416

```

SUMMARY

- 1) Constrained (static) and unconstrained (dynamic) arrays.
- 2) Flexible arrays
- 3) Multi-dimensional arrays
- 4) Arrays of arrays
- 5) Lists, sets, bags, etc.
- 6) Strings