**COMP205 IMPERATIVE LANGUAGES**

7. COMPOUND (HIGHER LEVEL) DATA TYPES III

1) Enumerated types
2) Records and structures
3) Accessing fields in records/structures
4) Operations on records/structures
5) Variant records
6) Dynamic and static arrays of records/structures

**ENumerated Types**

- An enumerated type is a user defined discrete type where the values are itemised.
- Ada example:
  ```
  type DAYS_T is (MONDAY, TUESDAY, WEDNESDAY, 
  THURSDAY, FRIDAY, SATURDAY, SUNDAY); 
  NEWDAY: DAYS_T := WEDNESDAY; 
  ```
- C example:
  ```
  typedef enum days { Monday, Tuesday, 
  Wednesday, Thursday, Friday, Saturday, 
  Sunday | DAYS_T; 
  DAYS_T newDay = Wednesday; 
  ```

```c
void main(void) {
    typedef enum days {
        monday=1, tuesday, wednesday, 
        thursday, friday, saturday, sunday
    } DAYS_T;
    DAYS_T newDay1 = monday, newDay2;
    printf("%d\n",newDay1);
    printf("Successor to newDay1 = %d\n",newDay1+1);
    newDay2 = newDay1+1;
    printf("%d\n",newDay2);
    printf("%d\n",newDay2);
    if (newDay2 == 3) printf("newDay2 = Wednesday\n";
    }
```

**ADA ATTRIBUTES FOR ENUMERATED TYPES**

- First - gives first value of the enumeration.
- Last - gives last value of the enumeration
- pred(X) - gives the predecessor of the enumerated value X (which must not be the first value).
- succ(X) - gives the successor of the enumerated value X (which must not be the last value).
- Some languages (e.g. Pascal) support an attribute ord(X) which returns the ordinal value of X.

**Standard High Level Types (Reminder) :**

1) Arrays
2) Strings
3) Enumerated types
4) Records/structures
5) Unions

- The values in an enumerated type are "ordered" so that the value listed first is least and the value listed last is greatest (an enumerated type is a discrete type).
- Conceptually at least, each value for an enumeration also has an ordinal value.
- This allows comparison of enumerated type values using operators such as < (less than) and > (greater than).
- In C simple arithmetic can be performed on enumerated type values (e.g. finding successors, predecessors, etc.).
A well known enumerated type (although not often recognised as such) is the Boolean type.

- **Ada**
  - `type BOOLEAN is (FALSE, TRUE);`

- **C**
  - `typedef enum {false, true} BOOLEAN;`

**NOTE:** the type boolean is actually a built in (enumerated) type in some imperative languages (Ada, Pascal).

It is also possible to consider any discrete type (e.g., integer, character) to be an enumerated type.

---

**RECORDS and STRUCTURES**

1) Overview
2) Accessing fields in records and structures
3) Operations on records and structures
4) Examples in C and Ada
5) Variant records
6) Dynamic and static arrays of records and structures

---

**ACCESING COMPONENTS OF RECORDS/STRUCTURES**

- The fields of a record/structure are accessed using the record name and the field name (used in this way the latter is referred to as a selector) linked by an operator.

- In both C and Ada this operator is a “.” (dot).

- However, in C, if a pointer to a structure is used instead of its name an “arrow” operator is used, `->`.

---

**ASSIGNMENT OF RECORDS/STRUCTURES**

- In some imperative languages (C) each member of a structure must be assigned a value individually.

- Other imperative languages (Ada) support the use of aggregates.

---

**COMPARISON OF RECORDS**

- Some imperative languages (Ada, Pascal) provide an operator to carry out comparison of records (Ada uses the = and /= operators).
**ADA RECORD EXAMPLE**

```ada
procedure EXAMPLE is
    type DATE_T is record
        DAY: integer;
        MONTH: string(1..9);
        YEAR: integer;
    end record;

    BIRTHDAY: DATE_T := (15,"May",1993);
begin
    putline("Birthday = ");
    put(BIRTHDAY.DAY); put(" ");
    put(BIRTHDAY.MONTH);
    put(BIRTHDAY.YEAR); new_line;
end EXAMPLE;
```

**C STRUCTURE EXAMPLE**

```c
typedef struct date {  
    int day; char month[9];  
    int year;  
} DATE_T, *DATE_PTR_T;  

void output_structure(DATE_PTR_T);  

void main(void) {  
    DATE_T stPatricksDay;  
    stPatricksDay.day = 17;  
    strcpy(stPatricksDay.month,"March");  
    stPatricksDay.year = 1996;  
    output_structure(stPatricksDay);  
    }  

void output_structure(DATE_PTR_T datePtr) {  
    printf("%d, %s, %d\n",  
            *datePtr->day,  
            *datePtr->month,  
            *datePtr->year);  
    }
```

**VARIANT RECORDS**

- It is sometimes useful to be able to specify a number of alternative fields for a record, this is referred to as a **variant record**.
- Such records are a feature of imperative languages such as Pascal and Modula-2 (but not Ada or C).
- A variant record comprises a fixed part, with fields as in a normal record, and a variant part, in which alternative fields are specified.
- The group of alternative fields which are applicable in any given instance of the record type is indicated by the value assigned to a tag field which is contained in the fixed part of the record.

**ARRAYS OF STRUCTURES/RECORDS**

- It is often necessary, given a particular application, to create a number of records/structures of the same type in which to store data, e.g. an array of structures.

```c
typedef struct date {  
    int day; char month[9];  
    int year;  
} DATE_T, *DATE_PTR_T;  

void main(void) {  
    int numOFEl=3;  
    DATE_T structArray[numOFEl];  
    /* Load array */  
    /* Output array */  
    }
```

**DYNAMIC ARRAYS OF RECORDS/STRUCTURES**

- In many cases the number of elements in an array are known in advance, and consequently the amount of memory required is known in advance.
- We refer to such an array as a static array of structures/records.
- In other cases we do not know how many elements are to be in the array until run time, i.e. a dynamic array of structures/records.
- In this case we must create sufficient memory at run time.
- To create a dynamic data item we must use an "allocator" which reserves space in memory for the data item.
REMEMBER ON ALLOCATORS

• Ada Example: \texttt{new integer} (Access value)

• C Example: \texttt{malloc(sizeof(integer))} (pointer)

• Remember that an allocator (e.g. \texttt{new} or \texttt{malloc}) returns a reference (address), referred to as an \textit{access value} in Ada and a \textit{pointer} in C, which indicates the first byte of the allocated memory.

SUMMARY

1) Enumerated types
2) Records and structures
3) Accessing fields in records/structures
4) Operations on records/structures (assignment, comparison)
5) Examples in C and Ada
6) Variant records
7) Dynamic and static arrays of records/structures

```c
typedef struct date {
    int day; char month[9];
    int year;
} DATE_T, *DATE_PTR_T;

void main(void) {
    int numOfEl;
    DATE_PTR_T structArray;
    printf("Enter number of structures ");
    scanf("%d", &numOfEl);
    if (structArray = (DATE_PTR_T) malloc(sizeof(DATE_T) * numOfEl)) == NULL) {
        printf("Insufficient space\n");
        exit(-1);
    }

    --- Rest of code here ---
```