Data Types

Lecture 11

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Record Types

- A ***record*** is an aggregate of data elements in which the individual elements are identified by ***names*** and accessed through offsets from the beginning of the structure.
- There is frequently a need in programs to model ***a collection of data*** in which the individual elements are ***not of the same type or size***.
  - For example, information about a college student
    - Name, student number, grade point average
    - A character string for the name
    - An integer for the student number
    - A floating-point for the grade point average, and so forth
- ***Records*** are designed for this kind of need.

Record Types

- It may appear that ***records*** and heterogeneous ***arrays*** are the same, but that is not the case.
- The elements of a heterogeneous ***array*** are all data objects with ***same type***.
- The elements of a record are of potentially ***different sizes*** and reside in adjacent memory locations.

Record Types

- In C, C++, and C#, ***records*** are supported with the ***struct*** data type.

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeber;
}
```

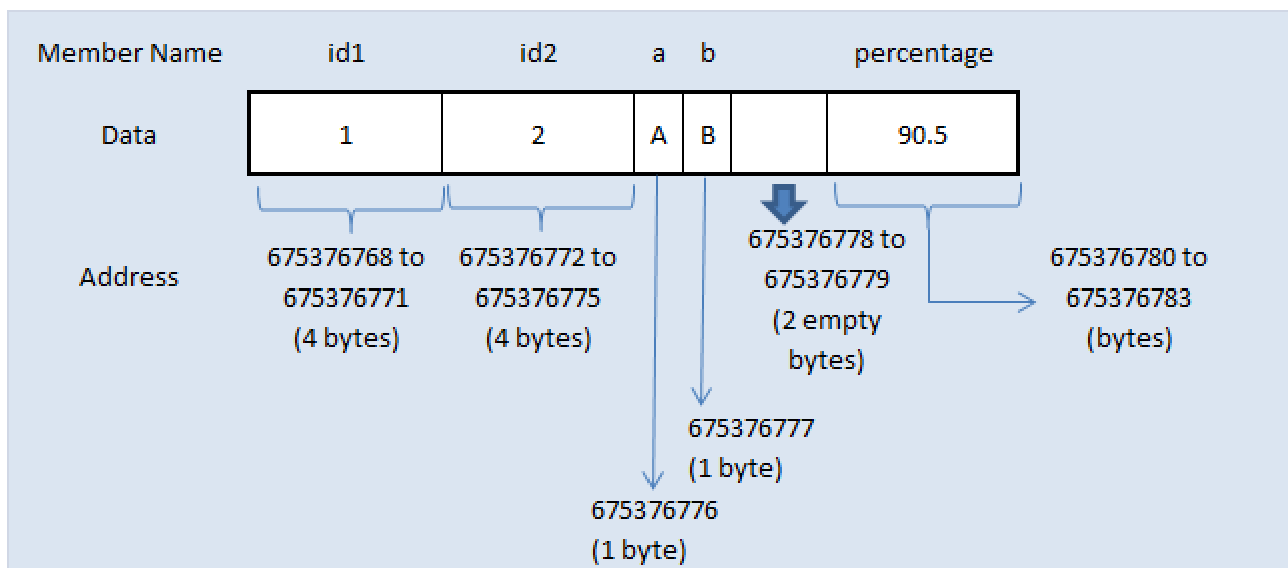
```
struct student
{
        int id1;
        int id2;
        char a;
        char b;
        float percentage;
};
```

Record Types

- In C, C++, and C#, ***records*** are supported with the ***struct*** data type.

```
struct student
{
    int id1;
    int id2;
    char a;
    char b;
    float percentage;
};
```

The elements of a record are of potentially ***different sizes*** and *reside in adjacent memory locations*.



Record Types: Definitions of Records

- The fundamental difference between a ***record*** and an ***array*** is that ***record elements***, or ***fields***, are ***not referenced by index***.
- Instead, the ***fields*** are ***named*** with ***identifiers***, and references to the fields are made using these ***identifiers***.
- Another difference between ***arrays*** and ***records*** is that records in some languages are allowed to include ***record*** or ***unions***.
- The COBOL form of a record declaration, which is part of the data division of a COBOL program, is illustrated in the following example:

```
01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        05  FIRST       PICTURE IS X(20).
        05  MIDDLE      PICTURE IS X(10).
        05  LAST        PICTURE IS X(20).
    02  HOURLY-RATE     PICTURE IS 99V99.
```

Record Types: Definitions of Records

- Ada uses a different syntax for records;
  - Rather than using the level numbers of COBOL, record structures are indicated by simply nesting record declarations inside record declarations.
- In Ada, records cannot be anonymous—they must be named types. Consider the following Ada declaration:

```
type Employee_Name_Type is record
        First : String (1..20);
        Middle : String (1..10);
        Last : String (1..20);
end record;
type Employee_Record_Type is record
        Employee_Name: Employee_Name_Type;
        Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

Record Types: Definitions of Records

- In Java and C#, ***records*** can be defined as ***data classes***, with nested records defined as nested classes.
- Data members of such classes serve as the record fields.

```
public class Student
{
    private String m_name;
    private int m_age;
    private String m_course;
    private String m_year;
    private String m_section;

    public Student( String name, int age, String course, String year, String section )
    {
        m_name = name;
        m_age = age;
        m_course = course;
        m_year = year;
        m_section = section;
    }
    …
}
```

Record Types: References to Record Fields

- References to the individual fields of records are syntactically specified by several different methods, two of which name the desired field and its enclosing records.

- COBOL field references have the form

  field_name OF record_name_1 OF . . . OF record_name_n

  - The first record named is the ***smallest or innermost record that contains the field***.
  - The next record name in the sequence is that of the record that contains the previous record, and so forth.

Record Types:
References to Record Fields

- For example, the MIDDLE field in the COBOL record example above can be referenced with

MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD

```
01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        05  FIRST        PICTURE IS X(20).
        05  MIDDLE       PICTURE IS X(10).
        05  LAST         PICTURE IS X(20).
    02  HOURLY-RATE      PICTURE IS 99V99.
```

Record Types:
References to Record Fields

- Most of the other languages use ***dot notation*** for field references
  - The components of the reference are connected with ***periods***.
- Names in ***dot notation*** have the opposite order of COBOL references:
  - They use the name of the largest enclosing record first and the field name last.

Record Types: References to Record Fields

- For example, the following is a reference to the field Middle in the earlier Ada record example:

Employee_Record.Employee_Name.Middle

```
type Employee_Name_Type is record
        First : String (1..20);
        Middle : String (1..10);
        Last : String (1..20);
end record;
type Employee_Record_Type is record
        Employee_Name: Employee_Name_Type;
        Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

Record Types: References to Record Fields

- C and C++ use this same syntax for referencing the members of their structures.

```
struct student
{
        int id1;
        int id2;
        char a;
        char b;
        float percentage;
};
```

```
struct student stu1;

stu1.id1 = 23;

stu1.id2 = 25;

stu1.float = 3.14;
```

Record Types: Implementation of Record Types

- The fields of records are stored in ***adjacent memory locations***.
- But because the sizes of the fields are not necessarily the same, the access method used for arrays is not used for records.
- Instead, the ***offset address***, relative to the beginning of the record, is associated with each field.
- Field accesses are all handled using these offsets.

Tuple Types

- A ***tuple*** is a data type that is similar to a record, except that the elements are ***not named***.

- Python includes an ***immutable tuple*** type.
  - If a tuple needs to be changed, it can be converted to an array with the list function.
  - After the change, it can be converted back to a tuple with the tuple function.

- One use of tuples is when an array must be ***write protected***,
  - When it is sent as a parameter to an external function and the user does not want the function to be able to modify the parameter.

Tuple Types

- Python's ***tuples*** are closely related to its ***lists***, except that tuples are ***immutable***.
- A tuple is created by assigning a tuple literal, as in the following example:

  myTuple = (3, 5.8, 'apple')

- The elements of a tuple can be referenced with indexing in brackets, as in the following:

  myTuple[1]

  - This references the second element of the tuple, because tuple indexing begins at 0.

Tuple Types

- Tuples can be contenated with the plus (**+**) operator.
- They can be deleted with the ***del*** statement.
- There are also other operators and functions that operate on tuples.

Tuple Types

- ML includes a tuple data type.
- An ML tuple must have at least two elements, whereas Python's tuples can be empty or contain one element.
- As in Python, an ML tuple can include elements of mixed types.
- The following statement creates a tuple:

```
val myTuple = (3, 5.8, 'apple');
```

Tuple Types

- The syntax of a tuple element access is as follows:

  #1(myTuple);

  - This references the first element of the tuple.

- A new tuple type can be defined in ML with a type declaration, such as the following:

  type intReal **= int * real;**

List Types

- ***Lists*** were first supported in the first functional programming language, LISP.
- They have always been part of the functional languages, but in recent years they have found their way into some imperative languages.
- Lists in Scheme and Common LISP are delimited by parentheses and the elements are not separated by any punctuation.
- For example,

(A B C D)

- Nested lists have the same form, so we could have

(A (B C) D)

List Types

- Data and code have the same syntactic form in LISP and its descendants.
  - If the list (A B C) is interpreted as code, it is a call to the function A with parameters B and C.

- The fundamental list operations in Scheme are two functions that take lists apart and that build lists.

- The ***CAR*** function returns the first element of its list parameter.

- For example, consider the following example:

(CAR '(A B C))

List Types

- The ***CDR*** function returns its parameter list minus its first element.

- For example, consider the following example:

  (CDR '(A B C))

  - This function call returns the list (B C).

Union Types

- A ***union*** is a type whose variables may store ***different type*** values at ***different times*** during program execution.
- Difference between ***Struct*** and ***Union***

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Union Types: Discriminated Versus Free Unions

- C and C++ provide union constructs in which there is no language support for type checking.
- In C and C++, the ***union*** construct is used to specify union structures.
- The unions in these languages are called ***free unions***, because programmers are allowed complete freedom from type checking in their use.
- For example, consider the following C union:

```
union flexType {
    int intEl;
    float floatEl;
};
union flexType el1;
float x;
. . .
el1.intEl = 27;
x = el1.floatEl;
```

Pointer and Reference Types

- A ***pointer*** type is one in which the variables have a range of values that consists of ***memory addresses***.
- Languages that provide a pointer type usually include two fundamental pointer operations: ***assignment*** and ***dereferencing***.
- The first operation sets a pointer variable's value to some useful ***address***.
- The second operation is used to access or manipulate data contained in memory location pointed to by a pointer.

Pointer and Reference Types: Pointer Operations

- In C++, it is explicitly specified with the ***asterisk*** (*) as a prefix unary operator.
- Consider the following example of dereferencing:
  - If ptr is a pointer variable with the value 7080 and the cell whose address is 7080 has the value 206, then the assignment

j = *ptr