ICOM 4036 Programming Languages

Data Types

- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer and Reference Types

This lecture covers review questions 10-21

Dr. Amirhossein Chinhaei, ECE, UPRM Spring 2010

***Some slides are adapted from the Sebesta’s textbook

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example
    ```
    int list [] = {4, 5, 7, 83};
    ```
  - Character strings in C and C++
    ```
    char name [] = "freddie";
    ```
  - Arrays of strings in C and C++
    ```
    char *names [] = {"Bob", "Jake", "Joe"};
    ```
  - Java initialization of String objects
    ```
    String[] names = {"Bob", "Jake", "Joe"};
    ```
  - Ada
    ```
    List : array (1..5) of Integer :=
    (1 => 17, 3 => 34, others => 0);
    ```

Arrays Operations

- APL provides the most powerful array processing operations for vectors and matrices as well as unary operators (for example, to reverse column elements)
- Ada allows array assignment, also concatenation
- Python’s array assignments, but they are only reference changes. Python also supports array concatenation and element membership operations
- Ruby also provides array concatenation
- Fortran provides elemental operations that are between pairs of array elements
  - For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements
- A jagged matrix has rows (or columns) with varying number of elements
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays
- Fortran, Ada, and C# support rectangular arrays (C# also supports jagged arrays)
Slices

- A slice is some substructure of an array; nothing more than a referencing mechanism
- Slices are only useful in languages that have array operations
- **Fortran 95 Examples**
  - `Integer, Dimension (10) :: Vector`
  - `Integer, Dimension (3, 3) :: Mat`
  - `Integer, Dimension (3, 3, 4) :: Cube`
  - `Vector (3:6)` is a four element array
- **Ruby** supports slices with the `slice` method
  - `list.slice(2, 2)` returns the third and fourth elements of `list`

Implementation of Arrays

- **Access function** maps subscript expressions to an address in the array

- Access function for single-dimensional arrays:
  
  \[
  address(list[k]) = address(list[lower_bound]) + ((k - lower_bound) \times element_size)
  \]

Accessing Multi-dimensioned Arrays

- Two common ways:
  - **Row major order** (by rows)
    - used in most languages
  - **Column major order** (by columns)
    - used in Fortran
Locating an Element in a Multi-dimensioned Array

- General format
  \[ \text{Location}(a[i,j]) = \text{address of a}[\text{row_lb}, \text{col_lb}] + ((i - \text{row_lb}) \times n) + (j - \text{col_lb}) \times \text{element_size} \]

Compile-Time Descriptors

<table>
<thead>
<tr>
<th>Array</th>
<th>Multidimensional array</th>
<th>Element type</th>
<th>Element type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element type</td>
<td>Index type</td>
<td>Index type</td>
<td>Index type</td>
</tr>
<tr>
<td>Index lower bound</td>
<td>Index upper bound</td>
<td>Index range 1</td>
<td>Index range 1</td>
</tr>
<tr>
<td>Address</td>
<td>Index range n</td>
<td>Address</td>
<td>Address</td>
</tr>
</tbody>
</table>

Single-dimensioned array

Multi-dimensional array

Associative Arrays

- An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called **keys**
  - User-defined keys must be stored
- **Design issues:**
  - What is the form of references to elements?
  - Is the size static or dynamic?
- Built-in type in **Perl**, **Python**, **Ruby**, and **Lua**

Associative Arrays in **Perl**

- Names begin with `%`; literals are delimited by parentheses
  ```perl
  $hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" => 65, ...);
  ```
- Subscripting is done using braces and keys
  ```perl
  $hi_temps("Wed") = 83;
  ```
- Elements can be removed with `delete`
  ```perl
  delete $hi_temps("Tue");
  ```
**Record Types**

- A record is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names.

- **Design issues:**
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed?

**Definition of Records in COBOL**

- COBOL uses level numbers to show nested records:
  
  ```cobol
  01 EMP-REC.
     02 EMP-NAME.
        05 FIRST PIC X(20).
        05 MID PIC X(10).
        05 LAST PIC X(20).
     02 HOURLY-RATE PIC 99V99.
  ```

**Definition of Records in Ada**

- In Ada, record structures are indicated in an orthogonal way
  
  ```ada
  type Emp_Nam_Type is record
     First: String (1..20);
     Mid: String (1..10);
     Last: String (1..20);
  end record;
  
  type Emp_Rec_Type is record
     Emp_Nam: Emp_Nam_Type;
     Hourly_Rate: Float;
  end record;
  ```

**References to Records**

- Record field references
  - COBOL: `field_name OF record_name_1 OF ... OF record_name_n`
  - Others (dot notation): `record_name_1.record_name_2 ... record_name_n.field_name`

- **Fully qualified references** must include all record names.

- **Elliptical references** allow leaving out record names as long as the reference is unambiguous,
  - for example in COBOL:
    ```cobol
    FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC
    ```
    are elliptical references to the employee’s first name.
Operations on Records

- Assignment is very common if the types are identical
- **Ada** allows record comparison
- **Ada** records can be initialized with aggregate literals
- **COBOL** provides `MOVE CORRESPONDING`
  - Copies a field of the source record to the corresponding field in the target record

Evaluation and Comparison to Arrays

- Records are used when collection of data values is heterogeneous
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Implementation of Record Type

- Offset address relative to the beginning of the records is associated with each field

Unions Types

- A **union** is a type whose variables are allowed to store different type values at different times during execution
- **Design issues**
  - Should type checking be required?
  - Should unions be embedded in records?
Discriminated vs. Free Unions

- **Fortran, C, and C++** provide union constructs in which there is no language support for type checking; the union in these languages is called **free union**
- Type checking of unions require that each union include a type indicator called a **discriminant**
  - Supported by **Ada**

Ada Union Types

```ada
type Shape is (Circle, Triangle, Rectangle); type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
  Filled: Boolean;
  Color: Colors;
  case Form is
    when Circle => Diameter: Float;
    when Triangle =>
      Leftside, Rightside: Integer;
      Angle: Float;
    when Rectangle => Side1, Side2: Integer;
  end case;
end record;
```

Evaluation of Unions

- Free unions are unsafe
  - Do not allow type checking
- **Java** and **C#** do not support unions
  - Reflective of growing concerns for safety in programming language
- **Ada**’s discriminated unions are safe