Array accessing

- An array is an ordered sequence of identical objects.
- The ordering is determined by a scalar data object (usually integer or enumeration data). This value is called the subscript or index, and written as A[I] for array A and subscript I.
- Multidimensional arrays have more than one subscript. A 2-dimensional array can be modeled as the boxes on a rectangular grid.
- The L-value for array element A[I,J] is given by the accessing formula on the next slide.

Array storage

![Diagram showing array storage]

L-value(A[I,J]) = offset + number of rows * rowsize + number of columns * elementsize

\[
\text{L-value}(A[I,J]) = \alpha + \text{offset} + \text{rowsize} \\
(\text{rowsize} \times \text{columnsize}) + (\text{elementsize})
\]

- \(d_1\) = NumberElements * elementsize = \((L2 - L2 + 1) \times d2\)
- Rows to skip: \((I - L1)\)
- Columns to skip: \((J - L2)\)

\[
\text{L-value}(A[I,J]) = \alpha + d_1 \times (I - L1) + d_2 \times (J - L2)
\]
Array accessing (continued)

Rewriting access equation:

\[ \text{L-value}(A[I,J]) = \alpha - d1*L1 - d2*L2 + I*d1 + J*d2 \]

Set I = 0; J = 0;

\[ \text{L-value}(A[0,0]) = \alpha - d1*L1 - d2*L2 + 0*d1 + 0*d2 \]

\[ \text{L-value}(A[0,0]) = \alpha - d1*L1 - d2*L2, \text{ which is a constant.} \]

• Call this constant the virtual origin (VO); It represents the address of the 0th element of the array.

\[ \text{L-value}(A[I,J]) = VO + I*d1 + J*d2 \]

• To access an array element, typically use a dope vector:

\[
\begin{array}{c}
\text{VO} \\
d1 \\
d2
\end{array}
\]

\[ \text{Array storage} \]

Array accessing summary

To create arrays:

1. Allocate total storage beginning at \( \alpha \):
   \[ (U2-L2+1)*(U1-L1+1)*\text{eltsize} \]
2. \( d2 = \text{eltsize} \)
3. \( d1 = (U2-L2+1)*d2 \)
4. \( \text{VO} = \alpha - L1*d1 - L2*d2 \)
5. To access \( A[I,J] \):
   \[ \text{L-value}(A[I,J]) = \text{VO} + I*d1 + J*d2 \]

This works for 1, 2 or more dimensions.

• May not require runtime dope vector if all values are known at compile time. (e.g., in Pascal, \( d1, d2, \) and VO can be computed by compiler.)

• Next slide: Storage representation for 2-dimensional array.
Array storage representation

Array example

Given following array: var A: array [7..12, 14..16] of real;

- Give dope vector if array stored beginning at location 500.
  \[ d_2 = 4 \text{ (real data)} \]
  \[ d_1 = (16-14+1) \times 4 = 3 \times 4 = 12 \]
  \[ VO = 500 - 7 \times 12 - 14 \times 4 = 500 - 84 - 56 = 360 \]
  \[ L-value(A[I,J]) = 360 + 12 \times I + 4 \times J \]

1. VO can be a positive or negative value, and can have an address that is before, within, or after the actual storage for the array:

2. In C, VO = \( \alpha \) since bounds start at 0.
   
   Example: char A[25]
   
   \[ L-value(A[I]) = VO + (I-L1) \times d_1 = \alpha + I \times 1 = \alpha + I \]
**Slices**

**Array slices**

Given array : \(A[L1:U1, L2:U2]\): Give \(d1, d2,\) and \(VO\) for vector:

Dope vector \(A[1,:] = B[L2:U2]\)

\(VO = \text{L-value}(A[I,L2]) - d2\times L2\)

\(M1 = \text{eltsize} = d2\)

Dope vector \(A[:,J] = B[L1:U1]\)

\(VO = \text{L-value}(A[L1,J]) - d1\times L1\)

\(M1 = \text{rowsize} = d1\)

Create new dope vector that accesses original data
More on slices

- **Diagonal slices:**
  
  \[
  V_O = L\text{-value}(A[L_1,L_2]) - d_1* L_1 - d_2* L_2 \\
  M_1 = d_1 + d_2
  \]

- **Other possibilities:**

Associative arrays

Access information by name without having a predefined ordering or enumeration:

- **Example:** Names and grades for students in a class:
  
  \[
  \text{NAME}[i] = \text{name of i}\text{th student} \\
  \text{GRADE}[i] = \text{Grade for i}\text{th student}
  \]

**Associative array:** Use Name as index:

\[
\text{CLASS}[\text{name}] \text{ will be grade.}
\]

**Problem:** Do not know enumeration before obtaining data so dope vector method of accessing will not work.

Implemented in Perl and in SNOBOL4 (as a table)
**Structs in C**

- **Representation:** A sequence of objects:
  ```
  record { A: object; 
  B: object; 
  C: object }
  ```

**Union types**

```
typedef union { int X; 
  float Y; 
  char Z[4]; } B;
```

- B P;

- Similar to records, except all have overlapping (same) L-value.

- But problems can occur. What happens below?
  ```
  P.X = 142;
  printf("%On", P.Z[3])
  ```

  All 3 data objects have same L-value and occupy same storage. No enforcement of type checking.
  ⇒ Poor language design
**Variant records**

```pascal
type PayType = (Salaried, Hourly);
var Employee: record
  ID: integer;
  Dept: array[1..3] of char;
  Age: integer;
  case PayClass: PayType of
    Salaried: (MonthlyRate: real;
               StartDate: integer);
    Hourly: (HourRate: real;
              Reg: integer;
              Overtime: integer)
  end
end
```

**Variant records (continued)**

**Tagged union type - Pascal variant records**

```pascal
type whichtype = (inttype, realtype, chartype);
type uniontype = record
  case V: whichtype of
    inttype: (X: integer);
    realtype: (Y: real);
    chartype: (Z: char4); *Assumes string of length 4*
  end
end
```

**But can still subvert tagging:**

```pascal
var P: uniontype
P.V = inttype;
P.X = 142;
P.V = chartype;
```

**What is P.V value now?**