Structured Data Types

Structured Data Types

A structured data type (data structure) is an aggregate which has elements which have a different type

- The components may be elementary data objects or other data structures.
- Data structures may be system-defined or programmer defined.

- Some common data structures: (most languages provide at least one or two of these)
  - Arrays – multiple components of the same type
  - Records – multiple components which may be of different types
  - Sets
  - Lists
  - Hash table
- Issues for implementing data structures
  - Specification
  - Storage management

Data Structure Specification

Number of components

fixed: arrays, records   int a[10] → 10 fixed

variable: stacks, lists, sets, tables, files

Component type(s)

homogeneous: arrays, sets, files

heterogeneous: records, lists

Component Selection

use a subscript or name   a[10] or p.y

can be direct access or sequential access   array vs linked list

Limit on maximum number of components?

Component Organization – contiguous in memory?

Data Structure Operations

whole structure operations

assignment, parameter passing

struct x {int a; int b; int c;} ;

```
f(struct x)
   main() {
   struct x b;
   f(b);
}
```

relatively limited in most languages

Data Structure Operations

insertion and deletion of components – only makes sense for some data structures

inserting in a struct ? Inserting in a linked list.

creation/deletion of entire structure

Implementation of Data Structures

Implementing a data structure requires finding a balance between efficient element selection and efficient storage management

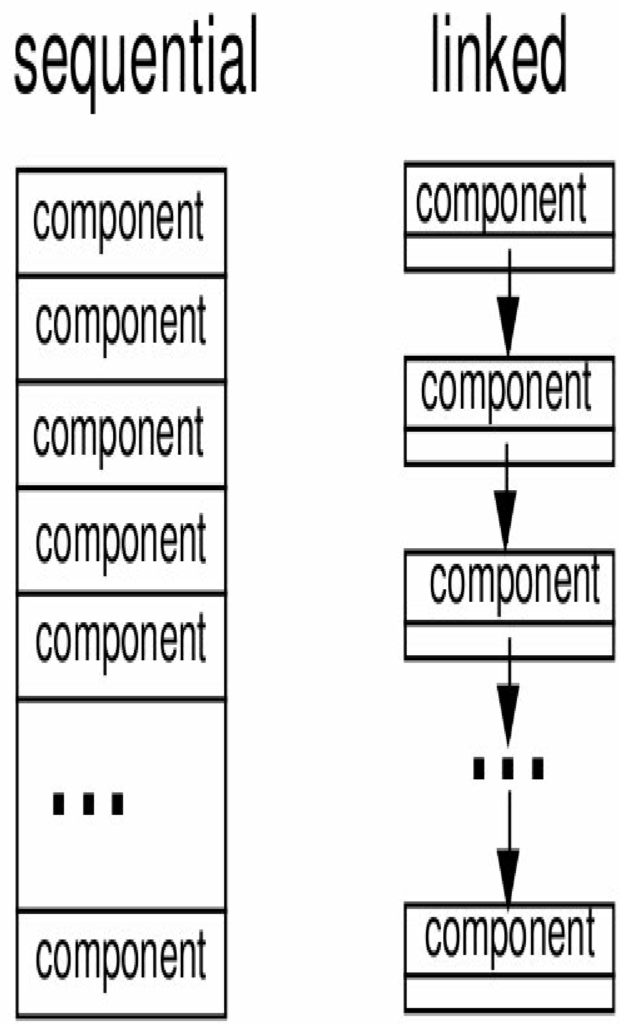
Storage representation

- need component storage and descriptor
- storage may be sequential (contiguous) or linked
- operations need efficient element access

Storage management

- lifetimes of individual components may be different
- aliases to a given object may have different lifetimes than the object itself (garbage and dangling references)

Implementation of Data Structures



Declarations and Type-checking

data structures have more attributes than scalar types

- type of data structure
- number of components
- component type

type checking more complicated

- static checking not always sufficient

int a [10], i; .... a[i] = 10;

- need to know if the element exists
- would like to be sure element has correct type

a[i] = “pqr”

Arrays

An array is an ordered sequence of identical objects.

Attributes

- Number of components, Data type int a[10] → 10 no. of compo, type int
- Subscript type and range for element access
  - in C, number of components determines the subscript range a[10] → Index 0 to 9
  - some languages like Pascal, Ada let you specify a range var a: array[-3..10] of int;
  - sometimes (Pascal) enumerated types are allowed as subscripts var a: array[color] of int;

Arrays

Array Operations

element access – usually a numeric index

a[i]= 5;

creation and destruction of entire array

a = new int[10];

assignment – language dependent: deep vs. shallow copy int a[10],b[10]; a= b;

arithmetic – APL provides a large set of array operations int a[10]; a+=5;

Arrays, cont.

- Implementation
  - usually sequential
    - homogeneity means components all have the same size
    - fixed size of array means component locations don't change
- For some languages, there may also be a run-time descriptor (dope vector)
  - upper and lower bounds for bounds checking
  - types for dynamic type-checking

Element access for 1-D arrays

A vector is a one-dimensional or linear sequence of elements.

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.

Computing the address of I'th element of array

lvalue(A[I]) = lvalue(A[0]) + I * elementSize

Multi-dimensional Arrays

Multidimensional arrays have more than one subscript.

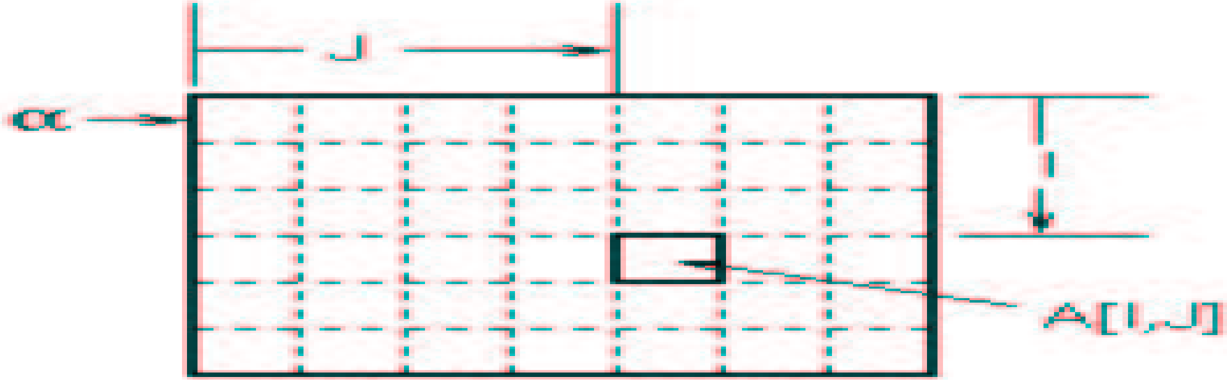
A 2-dimensional array can be modeled as the boxes on a rectangular grid.

Two-dimensional arrays can be stored in either row-major (most common) or column-major order.

The L-value for array element A[I,J] is given by the accessing formula on the nextslide

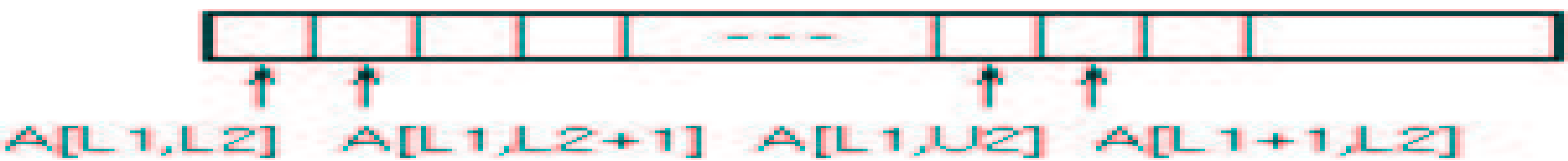
0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

0,0	1,0	2,0
0,1	1,1	2,1
0,2	1,2	2,2



L−value(A[I,J]) = α + skip I rows + j columns

Actual storage: A[L1:U1, L2:U2]



$$L\text{-value}(A[I,J]) = \alpha + \text{number of rows} * \text{rowsize} + \text{number of columns} * \text{elementsize}$$

$$= \alpha + a * d1 + b * d2$$

d2 = element size
4 − usually for integer
4 − usually for float
1 − usually for char

$$d1 = \text{NumberElements} * \text{elementsize} = (U2 - L2 + 1) * d2$$

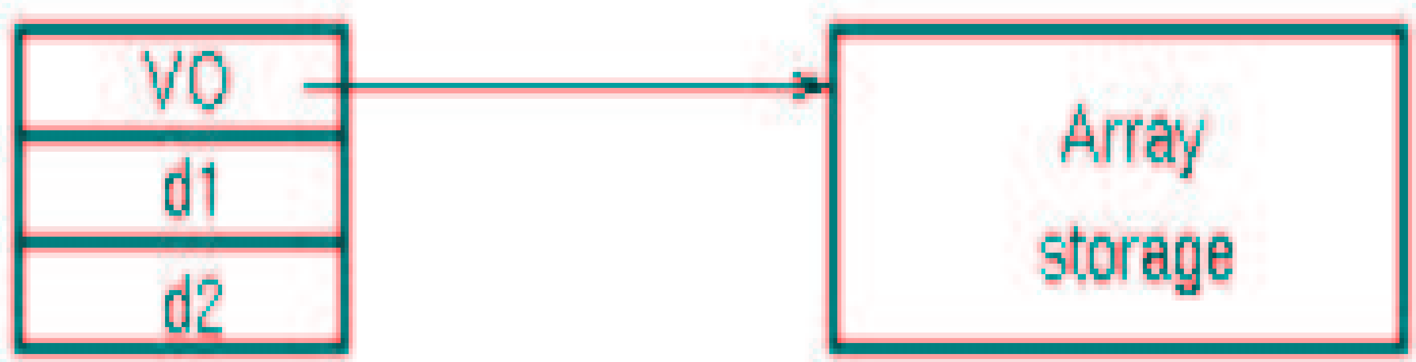
Rows to skip: (I−L1)
Columns to skip: (J−L2)

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

Dope Vector

A dope vector contains information about the array that is needed at run-time.

- Needed for doing run time checks (C does not do it !)
- starting address in memory (VO Virtual Origin)
- total amount of memory allocated (d1) (row)
- element size (d2)



Array Access

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:

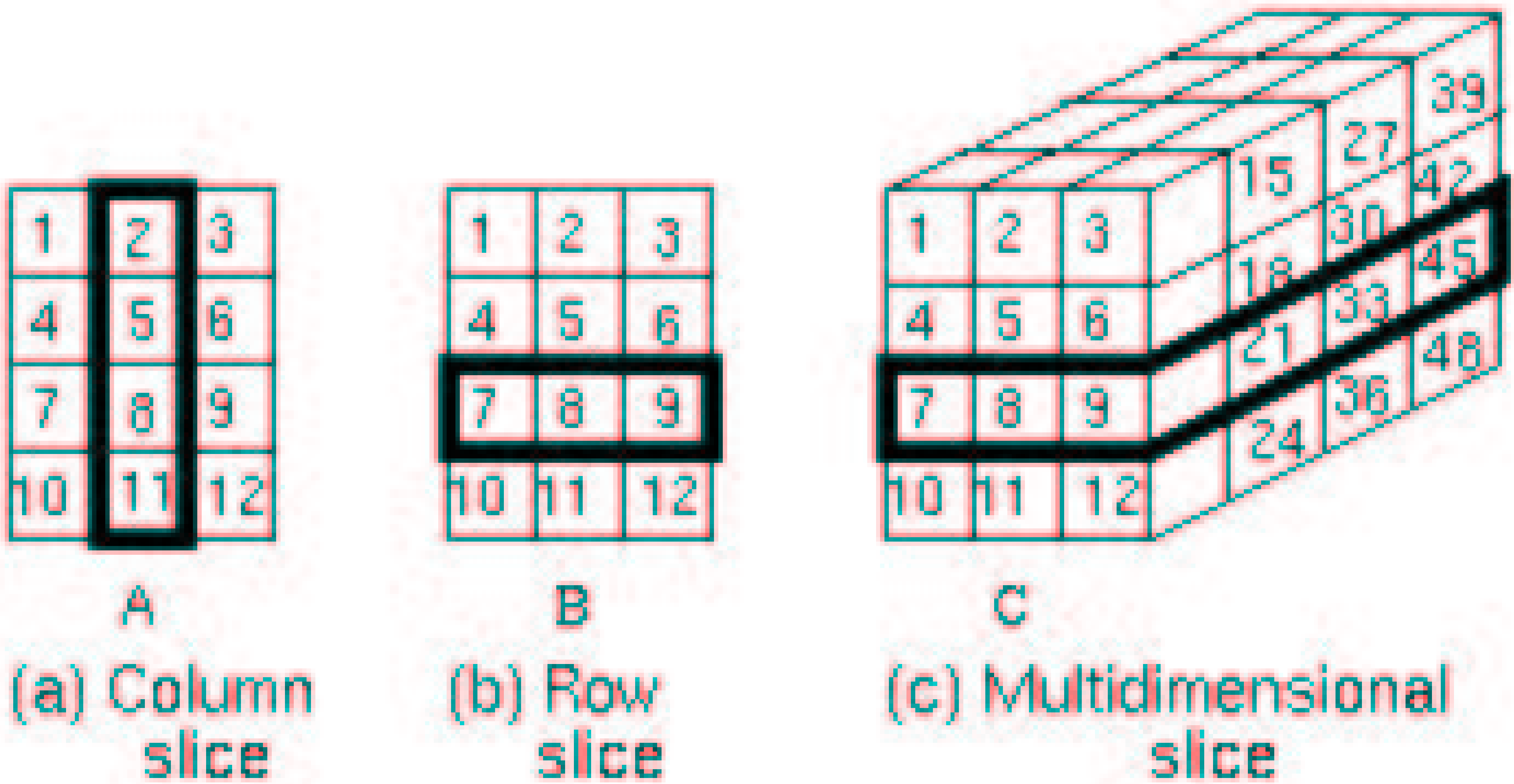
2. In C, VO is the same as the starting address for the element storage since bounds start at 0. Example:

char A[25]

L-value(A[I]) = VO + (I-L1) * d1 = + I * 1 = α + I

- Review and Self Study

Slices

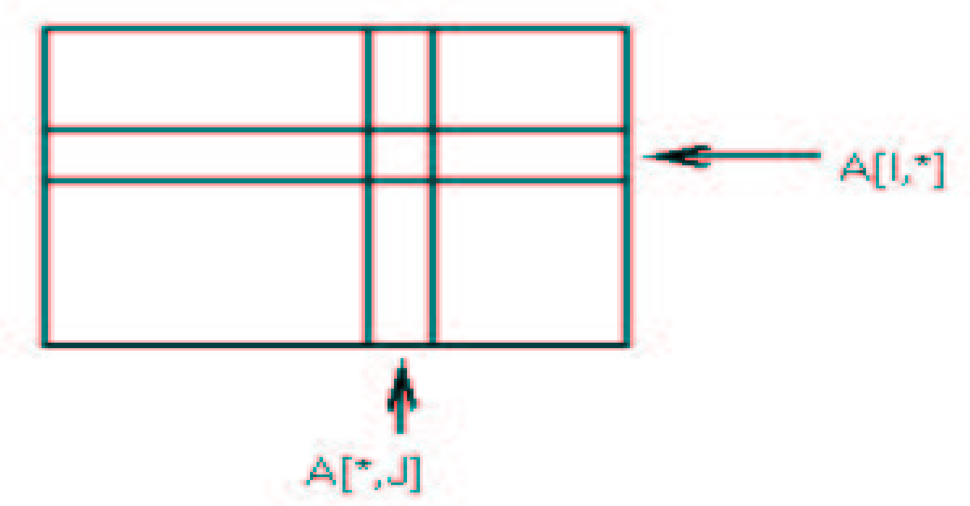


1
2
3
4
5
6
7
8
9
10
11
12
A
(a) Column slice
B
(b) Row slice
15
27
39
42
30
18
45
33
21
48
36
24
C
(c) Multidimensional slice

Array slices

Given array : A[L1:U1, L2:U2]:
Give d1, d2, and VO for v

Create new dope
vector that accesses
original data



Dope vector A[I,*] = B[L2:U2]	Dope vector A[*,J] = B[L1:U1]
VO = L-value(A[I,L2]) - d2*L2	VO = L-value(A[L1,J]) - d1*L1
M1 = eltsize = d2	M1 = rowsize = d1

More on slices

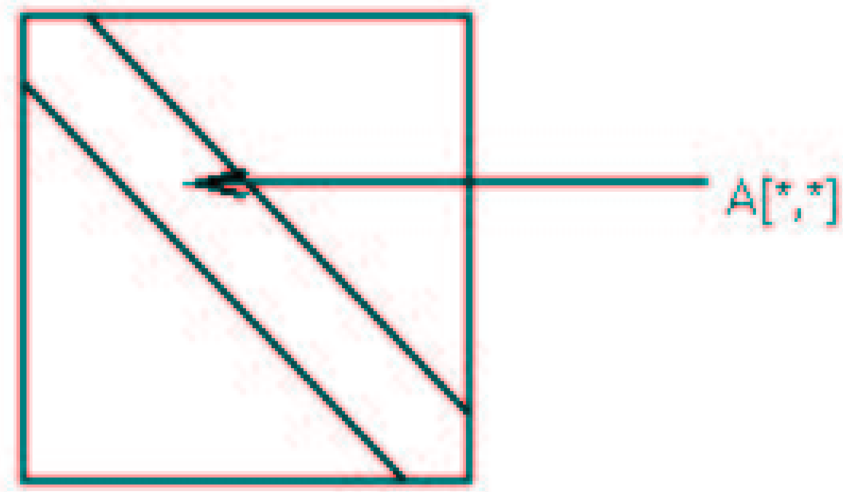
Diagonal slices:

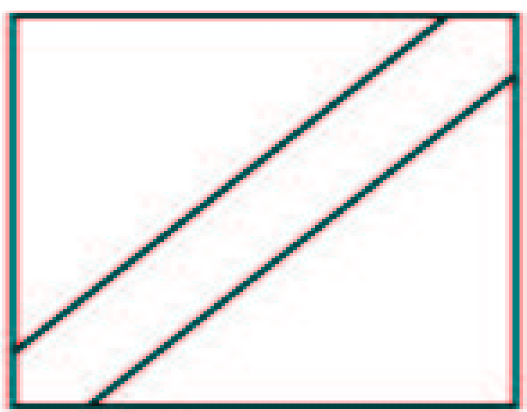
VO = L - value(A[L1, L2])

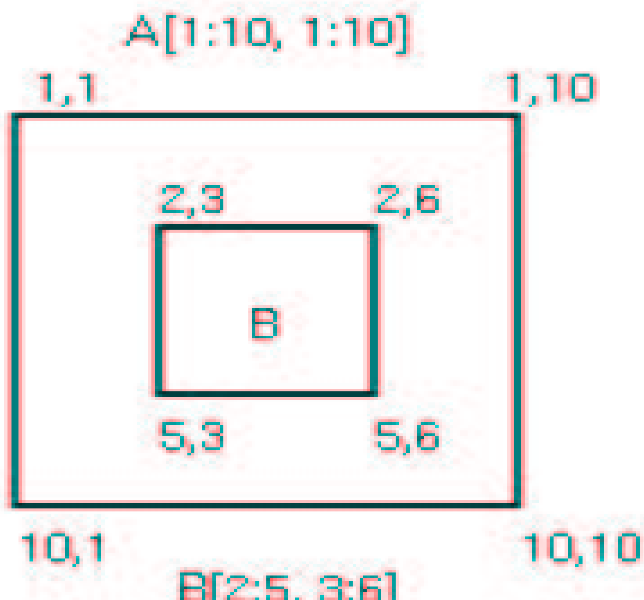
- d1 * L1 - d2 * L2

M1 = d1 + d2

Other possibilities:







Associative arrays

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

Example: Names and grades for students in a class:

NAME[I] = name of Ith student

GRADE[I] = Grade for Ith student

Associative array: Use Name as index:

CLASS[name] will be grade.

Associative Arrays

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)

A hashtable is a commonly used data structure even when it is not built in to the language. Java has a HashTable class.

Perl example

- `# % operator makes an associative array`
- `%ClassList = ("Michelle", ’A’, "Doris", ’B’, "Michael", ’D’);`
- `$ClassList{"Michelle"} has the value ’A’`
- `@y = %ClassList   # Converts ClassList to an enumeration`
- `                  # array with index 0..5`
- `$I= 0   $y[$I]  = Doris`
- `$I= 1   $y[$I]  = B`
- `$I= 2   $y[$I]  = Michael`
- `$I= 3   $y[$I]  = D`
- `$I= 4   $y[$I]  = Michelle`
- `$I= 5   $y[$I]  = A`

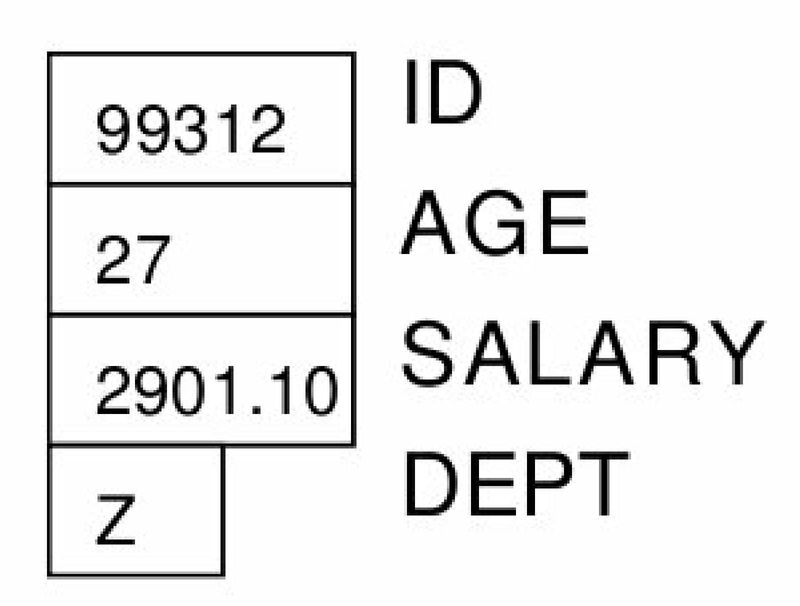
Records

Records are a fixed-length linear data structure like arrays except

The components may have different types.

Symbolic names are used for element access.

```
typedef  struct EmployeeType {
     int ID;
     int AGE;
     double SALARY;
     char DEPT;
}
```



Records

Specification – need name and type of each component

- Implementation

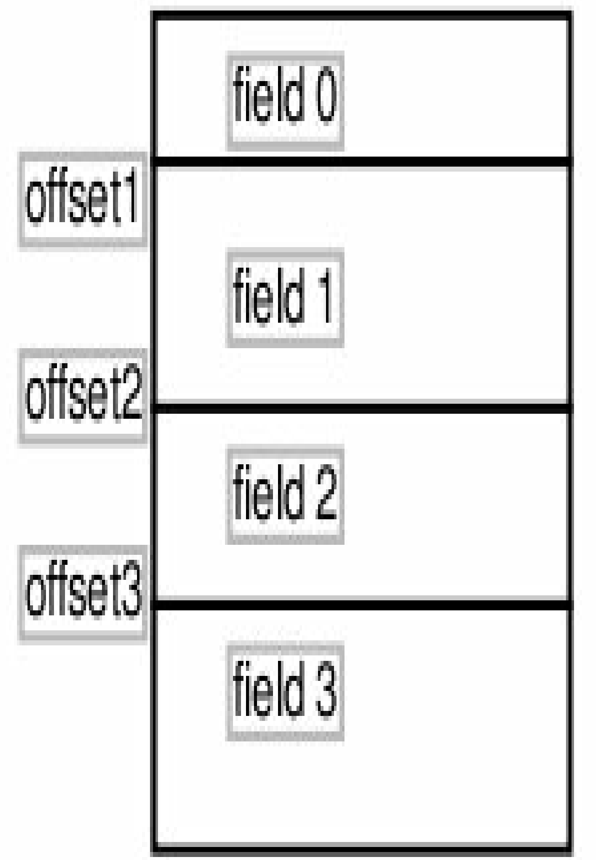
  use single sequential block of memoı

  components are stored in order

  position of component relative to sta
  be computed at translation time

  L – value(R.I) = α +   ∑
  }(sizeof R.j)

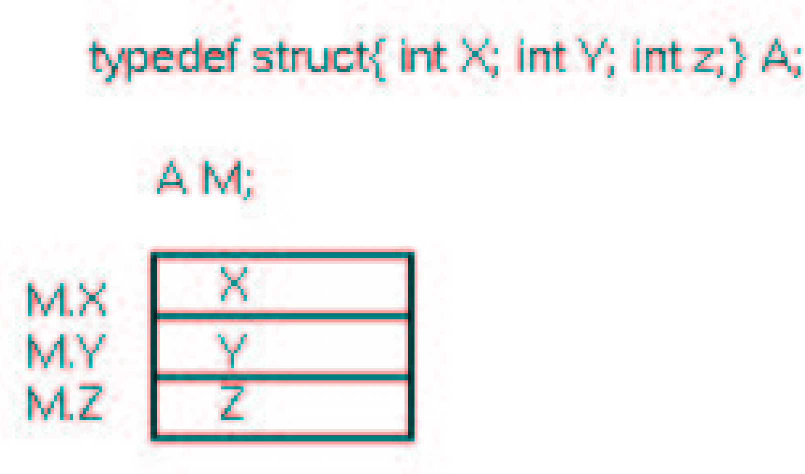
descriptors not usually needed

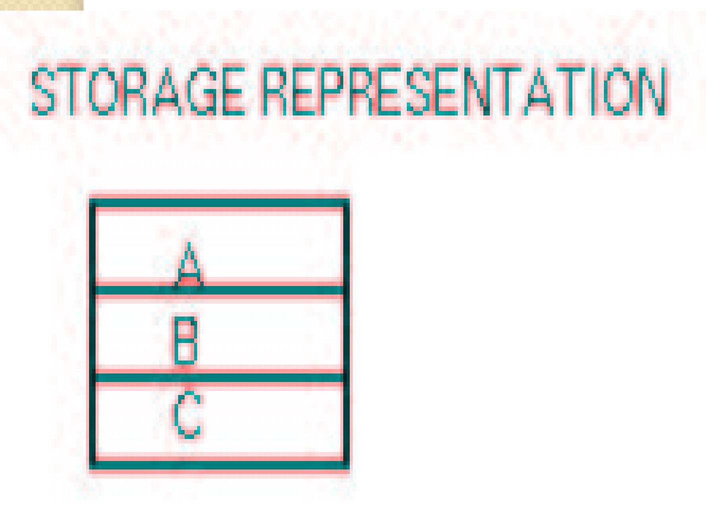


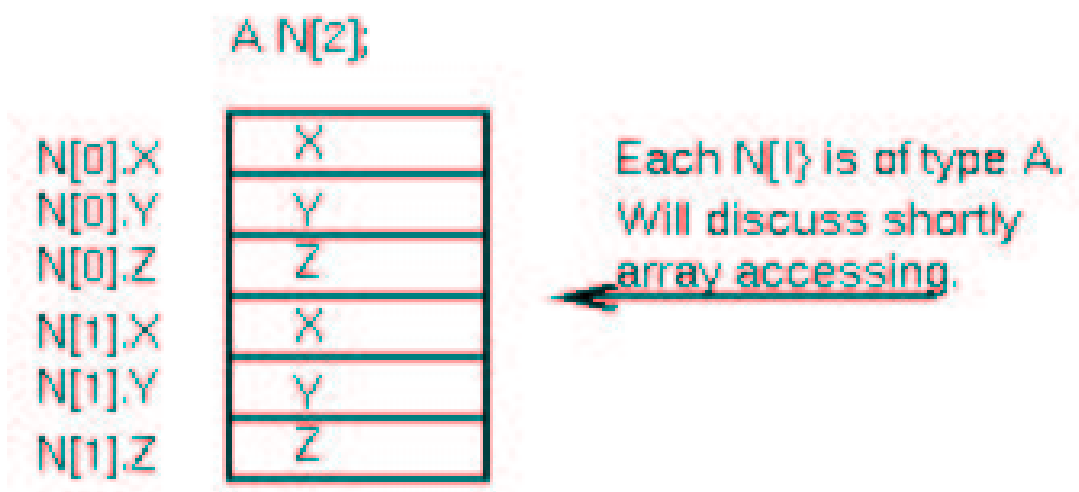
Structs in C

Representation: a sequence of objects

```
record { A:
B: object;
C: object }
```







Unions and Variant records

Situations often occur when you need one type that has can have different sets of data for different data objects.

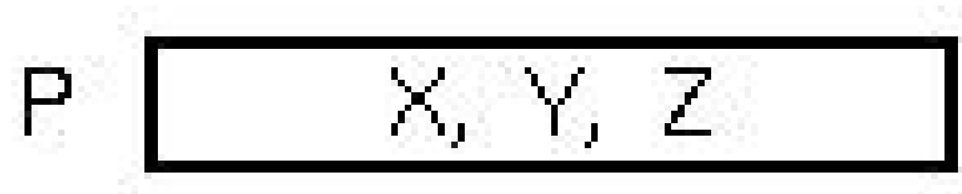
Think about your lexeme type - sometimes it needs no value, sometimes a string value, sometimes a numeric value.

Unions and variant records provide this flexibility.

Union types

Unions are similar to records, except all components share the same memory – only one can be used at a time.

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



Unions can be free (no type-checking) o discriminated.

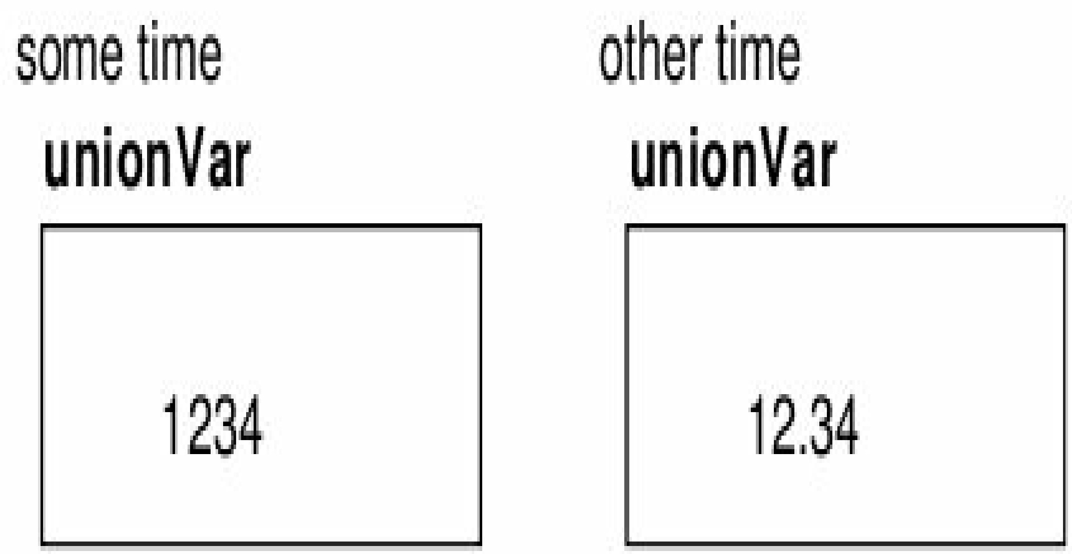
Union types

But problems can occur. What happens below?

```
P.X = 1234;
printf("%O\n", P.Z[3])
```

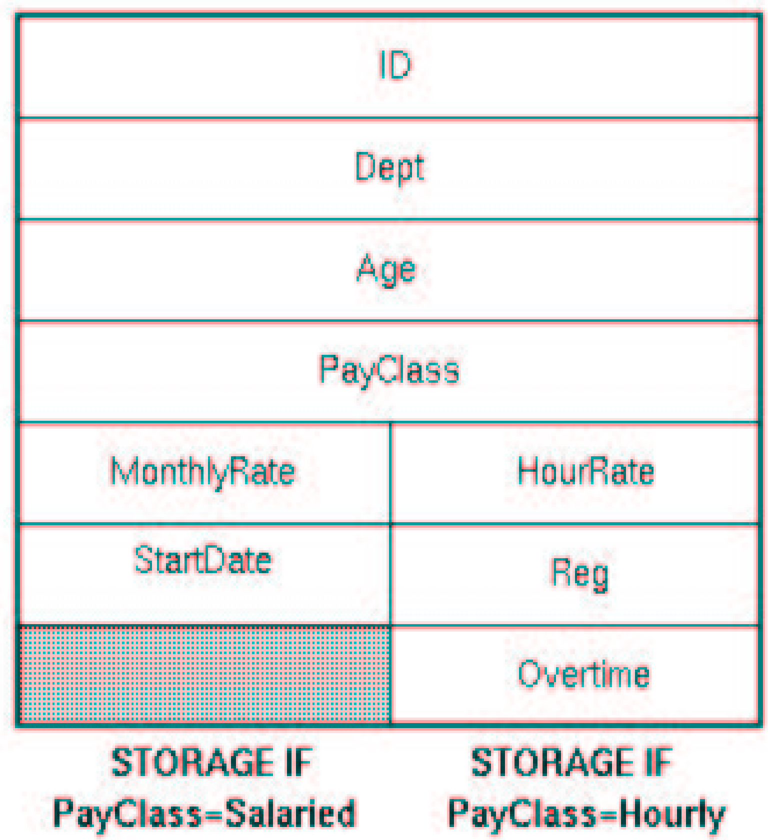
All 3 data objects have same L-value and occupy same storage. No enforcement of type checking.

Poor language design



Variant records

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



*

Variant records (continued)

Tagged union type - Pascal variant records

```
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
```

But can still subvert tagging: What is P.V value now?

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