CS 206

Lecture 18 – Data Types

Arrays

- Arrays (found in all imperative and OO PLs) can be understood as mappings.
- If the array's elements are of type T (base type) and its index values are of type S, the array 's type is S → T
- An array's length is the number of components, #S.
- Basic operations on arrays: –
  - **construction** of an array from its components
  - **indexing** – using a computed index value to select a component.

Arrays

- An array of type S → T is a finite mapping
- S is nearly always a finite range of consecutive values {*l*, *l*+1, …, u}. This is called the array's **index range**.
- In C and Java, the index range must be {0, 1, …, n–1}. In Pascal and Ada, the index range may be any scalar (sub)type other than real/float.

Index Ranges

- A **static array** is an array variable whose index range is fixed by the program code.
- A **dynamic array** is an array variable whose index range is fixed at the time when the array variable is created.
  - In Ada, the definition of an array type must fix the index type, but need not fix the index range. Only when an array variable is created must its index range be fixed.
  - Arrays as formal parameters of subroutines are often dynamic
- A flexible (or fully dynamic) array is an array variable whose index range is not fixed at all, but may change whenever a new array value is assigned

C Static Arrays

- Array variable declarations

```
float v1[] = {2.0, 3.0, 5.0, 7.0};
float v2[10];
```

- Use in functions

```
void print_vector (float v[], int n) {
// Print the array v[0], …, v[n-1] in the form "[…  …]".
   int i;
   printf("[%f", v[0]);
   for (i = 1; i < n; i++)
      printf(" %f", v[i]);
   printf("]");
}

…
print_vector(v1, 4);  print_vector(v2, 10);
```

A C array doesn't know its own length!

Example: Java flexible arrays

- Array variable declarations:

```
float[] v1 = {1.0, 0.5, 5.0, 3.5};
float[] v2 = {0.0, 0.0, 0.0};
…
v1 = v2;
```

index range is {0, …, 3}

index range is {0, …, 2}

v1's index range is now {0, …, 2}

- Method:

```
static void printVector (float[] v) {
// Print the array v in the form "[… … …]".
     System.out.print("[" + v[0]);
     for (int i = 1; i < v.length; i++)
        System.out.print(" " + v[i]);
     System.out.print("]");
}
…
printVector(v1);  printVector(v2);
```

Enhanced for:

```
for  (float f : v)
System.out.print(" " + f)
```

Array Allocation

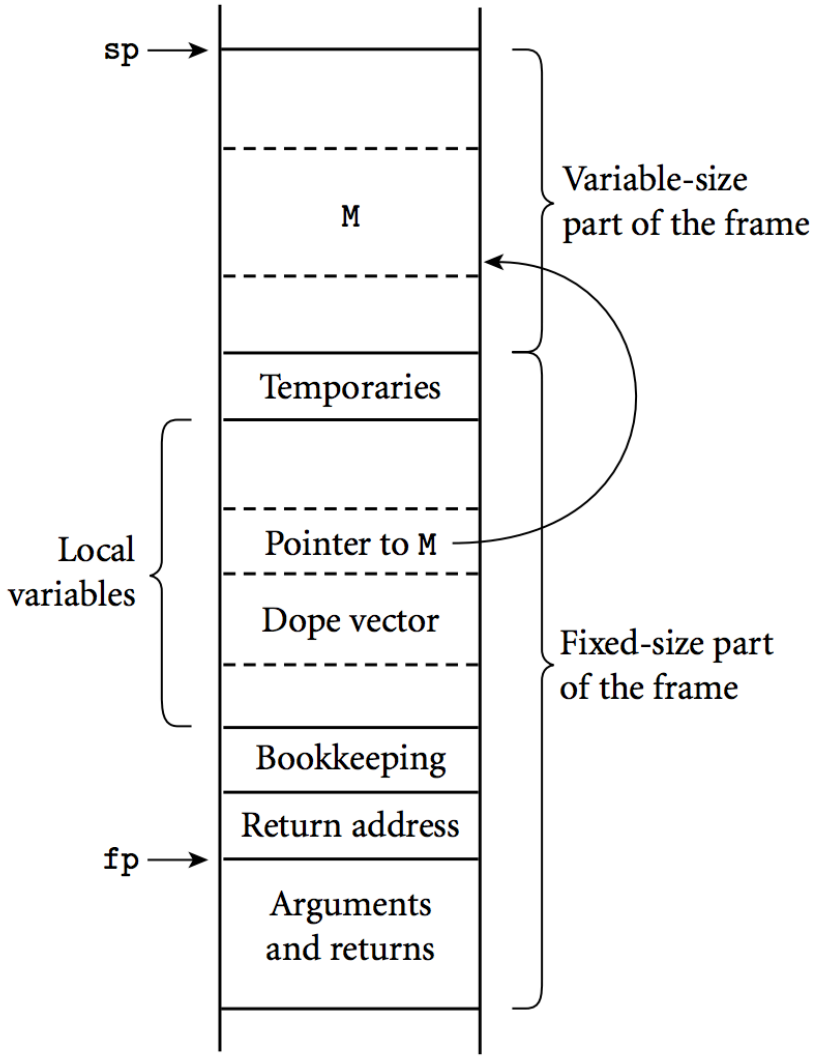
- **static array, global lifetime**
  - If a static array can exist throughout the execution of the program, then the compiler can allocate space for it in static global memory
- **static array, local lifetime**
  - If a static array should not exist throughout the execution of the program, then space can be allocated in the subroutine's stack frame at runtime.
- **dynamic array, local lifetime**
  - If the index range is known at runtime, the array can still be allocated in the stack, but in a variable size area
- **fully dynamic**
  - If the index range can be modified at runtime it has to be allocated in the heap

Array Management

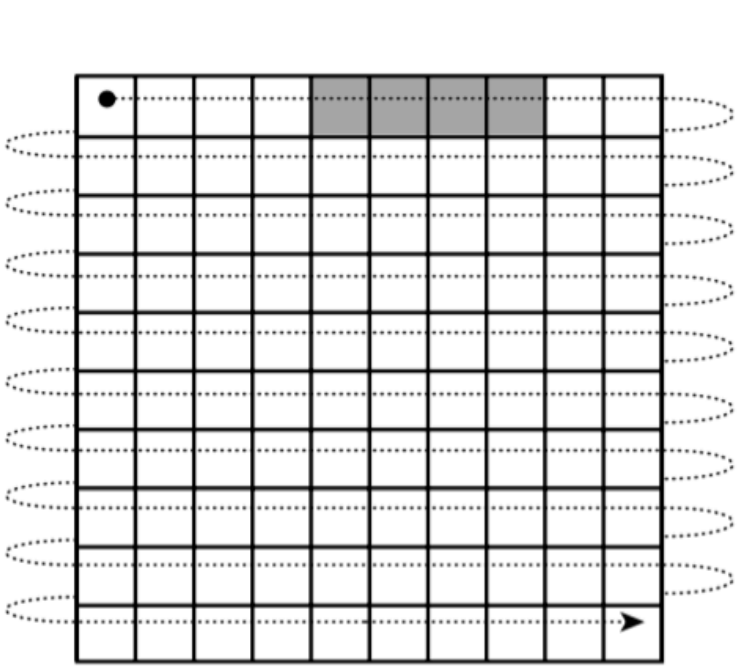
- Storage management is complex for arrays whose shape is not known until elaboration time, or whose shape may change during execution
- For these, the compiler must
  - Arrange / allocate space
  - make shape information available at run time
- Compiled languages
  - May allow dynamic bounds
  - require dimensions to be static
- Dope Vector
  - Used when dimensions and bounds are not statically known
  - Contains - lower bound of each dimension and size of each dimension

Dynamic Array Allocation on Stack

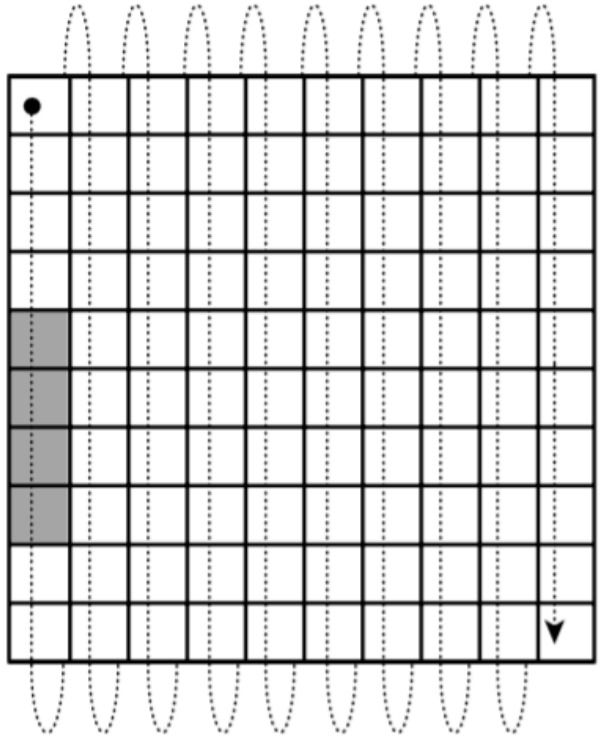
```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```



Array Layout



Row-major order

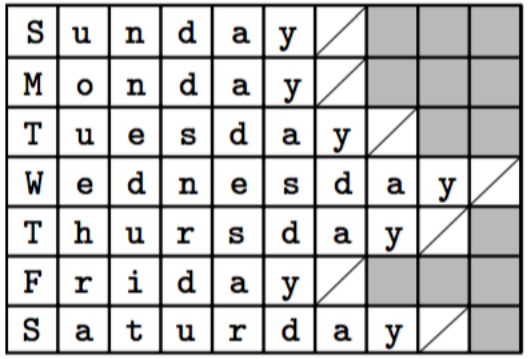


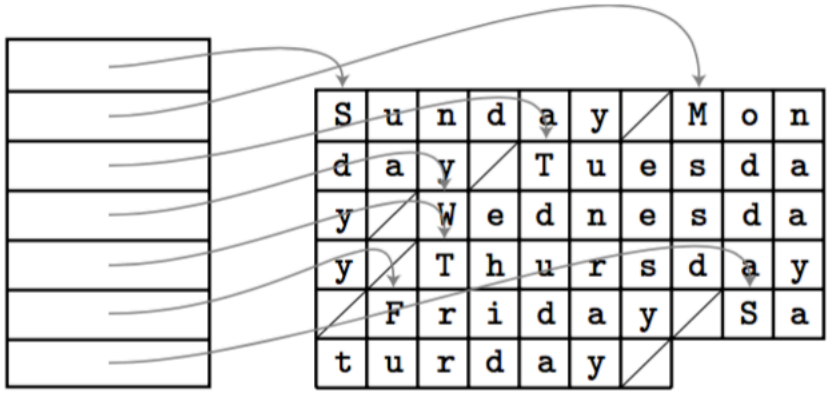
Column-major order

Contiguous vs Row Pointer layout

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```





- Optional in C; rule in Java
- allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
- Drawbacks?

Strings

- A string is a sequence of 0 or more characters.
- Some PLs (ML, Python) treat strings as primitive.
- Haskell treats strings as lists of characters. Strings are thus equipped with general list operations (length, head selection, tail selection, concatenation, …).
- Ada treats strings as *arrays* of characters. Strings are thus equipped with general array operations (length, indexing, slicing, concatenation, …).
- Java treats strings as *objects*, of class String.

Variant Records (Unions)

- Origin: Fortran I equivalence statement: variables should share the same memory location
- Example: C's **union** types
- Pros:
  - Saves space
  - Need of different access to the same memory locations for system programming
  - Alternative configurations of a data type

C -- union

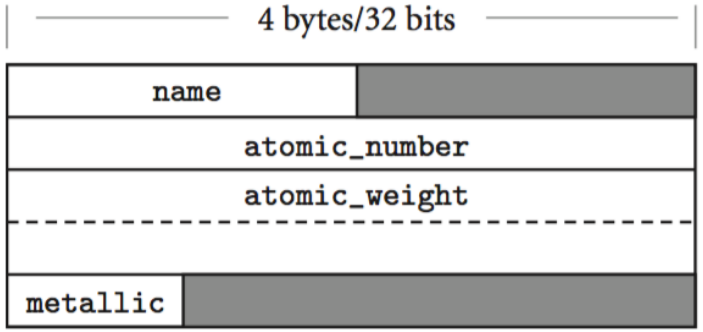
```
union {
    int i;
    double d;
    _Bool b;
};
```

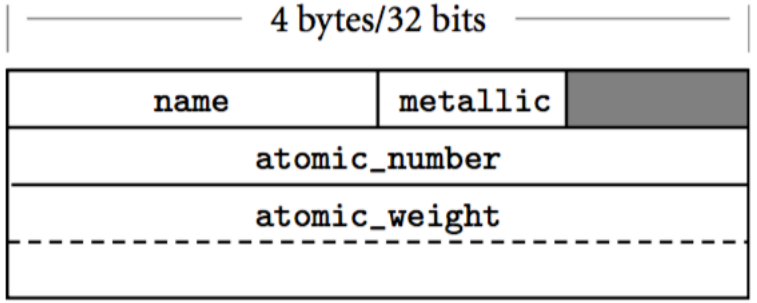
Structures

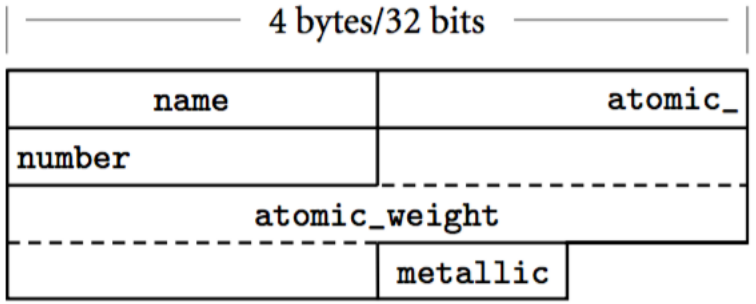
```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```

```
struct ore {
    char name[30];
    struct {
        char name[2];
        int atomic_number;
        double atomic_weight;
        _Bool metallic;
    } element_yielded;
};
```

```
struct ore {
    char name[30];
    struct element element_yielded;
};
```







Pointers

- Pointers serve two purposes:
  - efficient (and sometimes intuitive) access to elaborated objects (as in C)
  - dynamic creation of linked data structures, in conjunction with a heap storage manager

- What are these?

```
int **a, int *a[];
int *a[n];
int a[n][m];
```

More pointers

- Given this initialization:

```
int n;
int *a;
int b[10];
a = b;
```

- Are these equivalent?

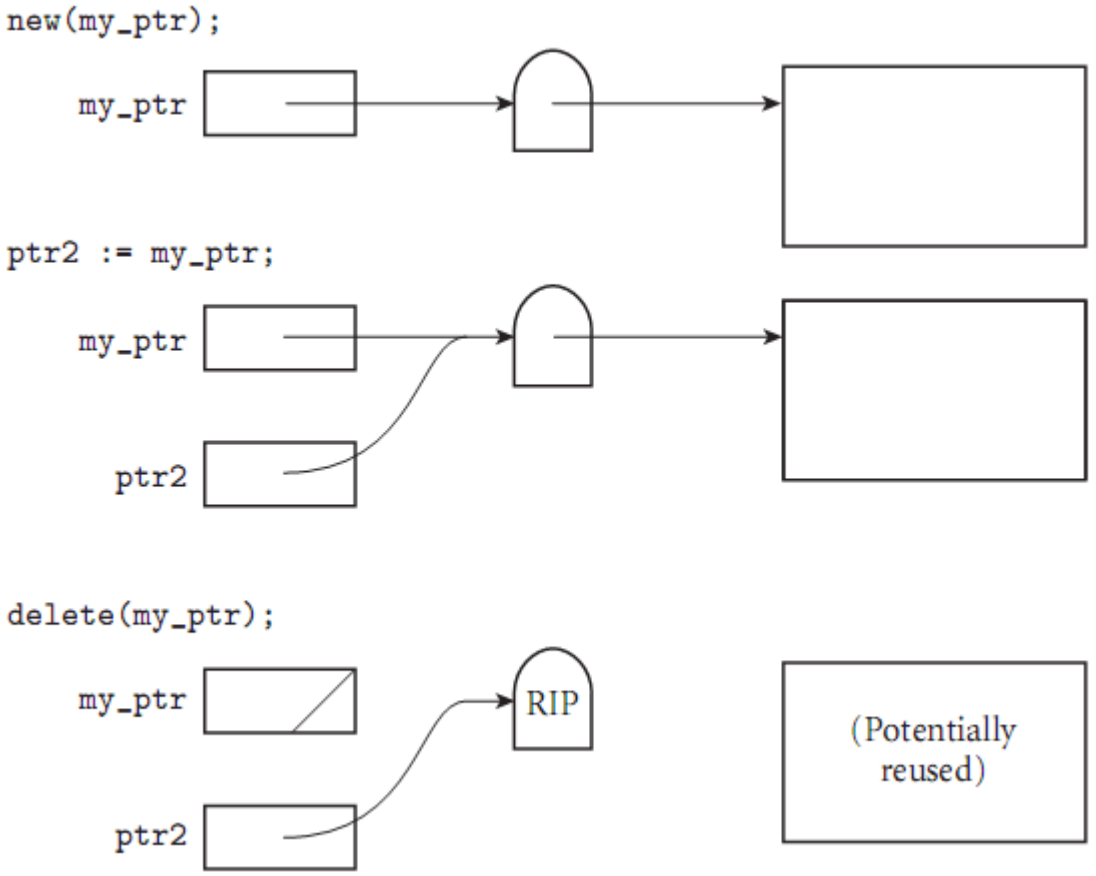
```
n = a[3];
n = *(a+3);
```

Dangling Pointers

- Problems with dangling pointers are due to
  - explicit deallocation of heap objects
    - only in languages that *have* explicit deallocation
  - implicit deallocation of elaborated objects
- Two implementation mechanisms to catch dangling pointers
  - Tombstones
  - Locks and Keys

Tombstone

- Tombstones
  - A pointer variable refers to a tombstone that in turn refers to an object
  - If the object is destroyed, the tombstone is marked as “expired”



Locks and Keys

- Heap objects are associated with an integer (lock) initialized when created.
- A valid pointer contains a key that matches the lock on the object in the heap.
- Every access checks that they match
- A dangling reference is unlikely to match.