Programming Paradigms

Composite Types (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart
Summer 2020
Overview

- Records
- Arrays
- Pointers and Recursive Types
Arrays

- Most common composite data type
- Conceptually: *Mapping from index type to element type*
  - Index types: Usually a discrete type, e.g., integer
  - Element type: Usually any type
Syntax

Varies across PLs

- Declaration
  - C: `char upper[26];`
  - Fortran: `character (26) upper`

- Accessing elements
  - C: `upper[3]` (indices start at 0)
  - Fortran: `upper(3)` (indices start at 1)
Multi-Dimensional Arrays

- **Indexing along multiple dimensions**
  - Single dimension: Sequence of elements
  - Two dimensions: 2D matrix of elements
  - Three dimensions: 3D matrix of elements
  - etc.

- **E.g., two-dimensional array in C:**
  ```c
  int arr[3][4];
  ```
  - 3 rows, 4 columns
Array Operations (Beyond Element Access)

- **Slicing**: Extract “rectangular” portion of array
  - Some PLs: Along multiple dimensions
- **Comparison**
  - Element-wise comparison of arrays of equal length:
    \[
    \text{arr1} < \text{arr2}
    \]
- **Mathematical operations**
  - Element-wise addition, subtraction, etc.
Example: Array Slicing in Fortran

10 x 10 array: matrix

matrix (3:6, 4:7)

matrix (6:, 5)

matrix (:4, :2:8:2)

Note: Fortran use column-major indexing, i.e.,
the first index refers to the row and
the second index refers to the column.
Memory Layout

- **Single dimension:** Elements are contiguous in memory
- **Multiple dimensions**
  - Option 1: **Contiguous, row-major layout**
    - E.g., in C
  - Option 2: **Contiguous, column-major layout**
    - E.g., in Fortran
  - Option 3: **Row-pointer layout**
    - E.g., in Java
Example:

```c
int arr[2][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8}
};
```

**Row-major:**

```
memory

1 2 3 4 5 6 7 8
```

**Column-major:**

```
1 5 2 6 3 1 7 4 8
```

**Row pointers:**

```
1 1 5 1 6 1 7 8
```

Larger addresses
Significance of Memory Layout

Layout determines efficiency of nested loops that iterate through multi-dimensional arrays

- CPU fetches entire cache lines from memory
- Accessing all data in a cache line is efficient
- Accessing data outside of current cache line: Cache miss
  - Causes expensive reading of another cache line
Quiz: Efficient Array Access

Given a large, two-dimensional array, which loop is faster in C and Fortran?

// C code, option 1
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        // access arr[i][j]
    }
}

// C code, option 2
for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
        // access arr[i][j]
    }
}

! Fortran code, option 1
do i=1,N
do j=1,N
    ! access arr(i,j)
    end do
end do

! Fortran code, option 2
do j=1,N
do i=1,N
    ! access arr(i,j)
    end do
end do

Please vote via Ilias.
Quiz: Efficient Array Access

Given a large, two-dimensional array, which loop is faster in C and Fortran?

// C code, option 1
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        // access arr[i][j]
    }
}

// C code, option 2
for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
        // access arr[i][j]
    }
}

! Fortran code, option 1
do i=1,N
do j=1,N
    ! access arr(i,j)
    end do
end do

! Fortran code, option 2
do j=1,N
do i=1,N
    ! access arr(i,j)
    end do
end do

Please vote via Ilias.