Arrays and Strings

SOME OF THE SLIDES ARE FROM:

Array References

Storage schemes

Row-major order (most languages)

- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
- A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Column-major order (Fortran)

- Lay out as a sequence of columns
- Leftmost subscript varies fastest
- A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Indirection vectors (Java)

- Vector of pointers to pointers to … to values
- Takes much more space, trades indirection for arithmetic

Laying Out Arrays

The Concept

A

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

These can have distinct & different cache behavior

Row-major order

A

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

Column-major order

A

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

Indirection vectors

Computing an Array Address

A[i]

In general: base(A) + (i – low) x sizeof(A[1])

Depending on how A is declared, base(A) may be:

- an offset from the ARP,
- an offset from some global label, or
- an arbitrary address.

The first two are compile time constants.

Computing an Array Address

A[i]

@A + (i – low) x sizeof(A[1])

In general: base(A) + (i – low) x sizeof(A[1])

Color Code:
Invariant

int A[1:10] ⇒ low is 1
Make low 0 for faster
access (saves a –)

Almost always a power of
2, known at compile-time
⇒ use a shift for speed

Computing an Array Address

A[i_1, i_2]

Color Code:
Invariant

Row-major order:

@A + ((i_1 – low_1) x ($high_2$ – low_2 + 1) + i_2 – low_2) x sizeof(A[1,1])

A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Column-major order:

@A + ((i_2 – low_2) x ($high_1$ – low_1 + 1) + i_1 – low_1) x sizeof(A[1,1])

Indirection vectors:

*(A[i_1])[i_2] — where A[i_1] is, itself, a 1-d array reference

e.g., @A + (i_1 – low) x sizeof(A[1])

Example

Calculate address A[3,5] in:

	1	2	3	4	5	6
1						
2						
3						
4						
5						

Using: @A + ((i1 – low1) x (high2 – low2 + 1) + i2 – low2) x sizeof(A[1,1])

And using: @A + ((i_2 – low_2) x ($high_1$ – low_1 + 1) + i_1 – low_1) x sizeof(A[1,1])

Optimizing Address Calculation

In row-major order.

where w = sizeof(A[1,1])

Start with:

$@A + ((i - low_1) * (high_2 - low_2 + 1) + j - low_2) * w$

Distribute to:

$@A + (i - low_1) * (high_2 - low_2 + 1) * w + (j - low_2) * w$

Factor into:

$@A + i * (high_2 - low_2 + 1) * w + j * w - low_1 * (high_2 - low_2 + 1) * w - low_2 * w$

If low_i, $high_i$, and w are known, the last two terms are a constant

Define $@A_0$ as: $@A - low_1 * (high_2 - low_2 + 1) * w - low_2 * w$

and len_2 as $(high_2 - low_2 + 1)$

Then, the address expression becomes: $@A_0 + (i * len_2 + j) * w$

Array Address Calculations

Array address calculations are a major source of overhead

- Scientific applications make extensive use of arrays and array-like structures
  - Computational linear algebra, both dense and sparse matrices
- Non-scientific applications use arrays, too
  - Representations of other data structures
    - →*Hash tables, adjacency matrices, tables, structures, …*

Array calculations tend iterate over arrays

- Loops execute more often than code outside loops
- Array address calculations inside loops make a huge difference in efficiency of many compiled applications

---

Array Address Calculations in a Loop

Naïve: Perform the address calculation twice:

```
DO j = 1, N
  R1 = $@A_0 + (j * len_1 + i) * sizeof(A[1,1])$
  R2 = $@B_0 + (j * len_1 + i) * sizeof(A[1,1])$
  MEM(R1) = MEM(R1) + MEM(R2)
END DO
```

```
DO j = 1, N
    A[i, j] = A[i, j] + B[i, j]
END DO
```

Array Address Calculations in a Loop

More sophisticated: Move common calculations out of loop

```
DO j = 1, N
  R1 = $@A_0$ + (j * $len_1$ + i) * sizeof(A[1,1])
  R2 = $@B_0$ + (j * $len_1$ + i) * sizeof(A[1,1])
  MEM(R1) = MEM(R1) + MEM(R2)
END DO
```

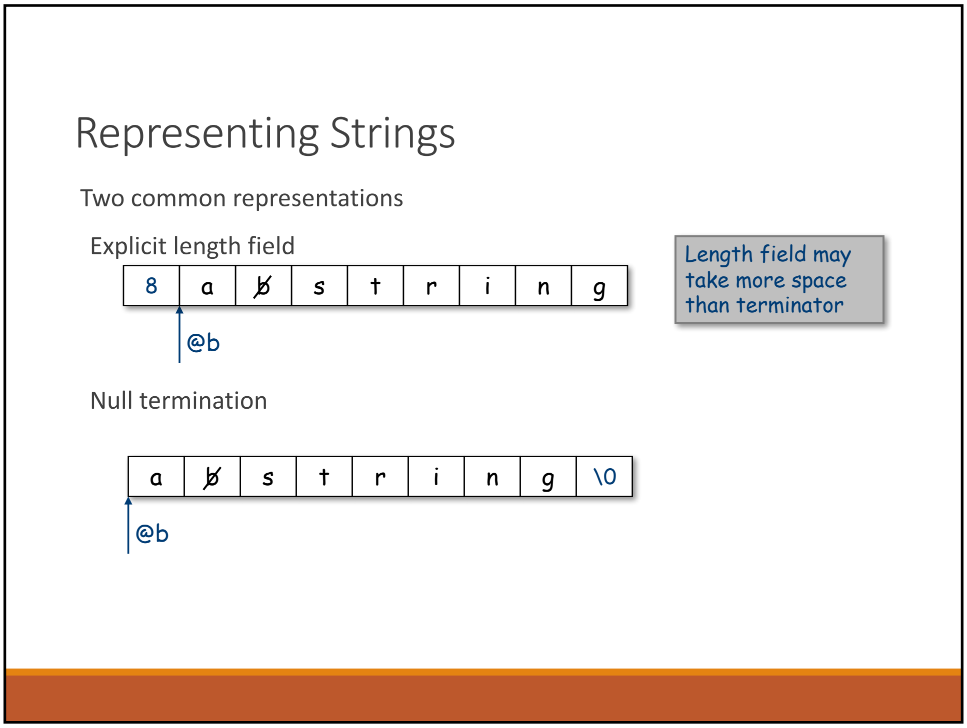
```
R1 = i * sizeof(A[1,1])
c = $len_1$ x sizeof(A[1,1])
R2 = $@A_0$ + R1
R3 = $@B_0$ + R1
DO j = 1, N
  a = j * c
  R4 = R2 + a
  R5 = R3 + a
  MEM(R4) = MEM(R4) + MEM(R5)
END DO
```

Array Address Calculations in a Loop

Even more sophisticated: Use addition rather than multiplication

```
R1 = i * sizeof(A[1,1])
c = $len_1$ x sizeof(A[1,1])
R2 = $@A_0$ + R1
R3 = $@B_0$ + R1
DO j = 1, N
  a = j * c
  R4 = R2 + a
  R5 = R3 + a
  MEM(R4) = MEM(R4) + MEM(R5)
END DO
```

```
R1 = i * sizeof(A[1,1])
c = $len_1$ * sizeof(A[1,1])
R2 = $@A_0$ + R1
R3 = $@B_0$ + R1
DO J = 1, N
  R2 = R2 + c
  R3 = R3 + c
  MEM(R2) = MEM(R2) + MEM(R3)
END DO
```



Representing Strings

Two common representations

Explicit length field

8	a	b̸	s	t	r	i	n	g

Length field may take more space than terminator

Null termination

a	b̸	s	t	r	i	n	g	\0