Memory Layout

Arrays in most language implementations are stored in contiguous locations in memory. For arrays of records, alignment constraints may result in small holes between consecutive relements.

For multidimensional arrays, you can store the elements in **row-major** or **column-major** order.

In **row-major order**, consecutive locations in memory hold elements that differ by one in the *final* subscript (except at the ends of rows). `matrix`$[2, 4]$ is followed by `matrix`$[2, 5]$

In **column-major order**, consecutive locations hold elements that differ by one in the *initial* subscript. `matrix`$[2, 4]$ is followed by `matrix`$[3, 4]$

The advantage of row-major order is that is make it easy to define a multidimensional array as an array of subarrays. With column-major order, the elements of the subarray would not be contiguous in memory.

Row-Pointer Layout

Some languages employ an alternative o contiguous allocation for some arrays. Rather than require the rows of an array to be adjacent, they allow them to lie anywhere in memory, and create an auxiliary array ofpointers to the rows. Row-pointer layout allows the rows to have different lengths, without devoting space to holes at the ends of the rows. This representation is sometime called a **ragged array**. Row-pointer layout also allows a program to construct an array from preexisting rows without copying. C, C++, and C# provide both contiguous and row-pointer organizations for multidimensional arrays. Java uses the row-pointer layout for all arrays.

The most common use of row-pointer layout in C is to represent arrays of strings.

Address Calculations

For the usual contiguous layout of arrays, calculating the address of a particular element is somewhat complicated, but straightforward. Suppose a compiler is given the following declaration for a three-dimensional array:

$$\texttt{A: array}[L_1 \cdots U_1] \texttt{ of array}[L_2 \cdots U_2] \texttt{ of array}[L_3 \cdots U_3] \texttt{ of elem\_type;}$$

Lets define constants for the sizes of the three dimensions:

$$\begin{aligned} S_3 &= \text{size of elem\_type} \\ S_2 &= (U_3 - L_3 + 1) \times S_3 \\ S_1 &= (U_2 - L_2 + 1) \times S_2 \end{aligned}$$

Here the size of a row (S_2) is the size of an individual element (S_3) times the number of elements in a row (assuming row-major layout). The size of a plane (S_1) is the size of a row (S_2) times the number of rows in a plane. The address of $A[i, j, k]$ is then

$$\begin{aligned} &\text{address of } A \\ &\quad + (i - L_1) \times S_1 \\ &\quad + (j - L_2) \times S_2 \\ &\quad + (k - L_3) \times S_3 \end{aligned}$$

We can compute the entire expression at run time, but in most cases a little rearrangement reveals that much of the computation can be performed at compile time. If the bounds of the array are known at compile time, then S_1, S_2 and S_3 are compile-time constants, and the subtractions of lower bounds can be distributed out of the parentheses:

$$(i \times S_1) + (j \times S_2) + \text{address of } A - [(L_1 \times S_1) + (L_2 \times S_2) + (L_3 \times S_3)]$$

The backeted expression in this formula is a compile-time constant (assuming the bounds of A are statically known). If A is a global varibale, then the address of A is statically known as well, and can be incorporated in the bracketed expression. If A is a local variable of a subroutine (with static shape), then the address of A can be decomposed into a static offset plus the contents of the frame pointer at run time.

If i, j, and/or k is known at compile time, then additional portions of the calculations of the address of $A[i, j, k]$ will move from the dynamic to the static part of the formula. If all the subscripts are known, then the entire address can be calculated statically. Conversley, if any of the bounds of the array are not known at compile time, then portions of the calculation will move from the static to the dynamic part of the formula. If lower bounds are always restricted to zero, as they are in C, then they never contribute to run-time cost.