

# CSE 351 Section 6 – Arrays and Structs

Welcome back to section, we're happy that you're here ☺

## Arrays

- Arrays are contiguously allocated chunks of memory large enough to hold the specified number of elements of the size of the datatype. Separate array allocations are not guaranteed to be contiguous.
- 2-dimensional arrays are allocated in row-major ordering in C (i.e. the first row is contiguous at the start of the array, followed by the second row, etc.).
- 2-level arrays are formed by creating an array of pointers to other arrays (i.e. the second level).

## Structs

- Structs are contiguously allocated chunks of memory that hold a programmer-defined collection of potentially disparate variables.
  - Individual fields appear in the struct in the order that they are declared
  - Each field follows its variable alignment requirement, with internal fragmentation added between fields as necessary.
  - The overall struct is aligned according to the largest field alignment requirement, with external fragmentation added at the end as necessary.
- 

Let's do a comparison of different data structure representations in C!

We'll see later in the course how the following questions become important for program performance (execution time) in terms of memory usage, speed of memory allocation, and speed of data access.

< questions on reverse side >

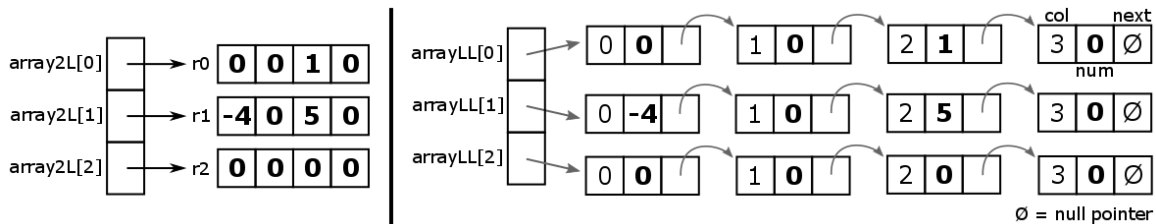
We have a two-dimensional matrix of integer data of size  $M$  rows and  $N$  columns. We are considering 3 different representation schemes:

- 1) 2-dimensional array `int array2D[][]`, //  $M*N$  array of ints
- 2) 2-level array `int* array2L[]`, and //  $M$  array of int arrays
- 3) array of linked lists `struct node* arrayLL[]`. //  $M$  array of linked lists (struct node)

Consider the case where  $M = 3$  and  $N = 4$ . The declarations are given below:

2-dimensional array:	2-level array:	Array of linked lists:
<code>int array2D[3][4];</code>	<code>int r0[4], r1[4], r2[4]; int* array2L[] = {r0,r1,r2};</code>	<code>struct node { int col, num; struct node* next; }; struct node* arrayLL[3]; // code to build out LLs</code>

For example, the diagrams below correspond to the matrix  $\begin{bmatrix} 0 & 0 & 1 & 0 \\ -4 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$  for `array2L` and `arrayLL`:



a) Fill in the following comparison chart:

	2-dim array	2-level array	Array of LLs:
Overall Memory Used			
Largest <i>guaranteed</i> continuous chunk of memory			
Smallest <i>guaranteed</i> continuous chunk of memory			
Data type returned by:	<code>array2D[1]</code>	<code>array2L[1]</code>	<code>arrayLL[1]</code>
Number of memory accesses to get <code>int</code> in the <i>BEST</i> case			
Number of memory accesses to get <code>int</code> in the <i>WORST</i> case			

b) Sam Student claims that since our arrays are relatively small ( $N < 256$ ), we can save space by storing the `col` field as a `char` in `struct node`. Is this correct? If so, how much space do we save? If not, is this an example of *internal* or *external* fragmentation?