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)

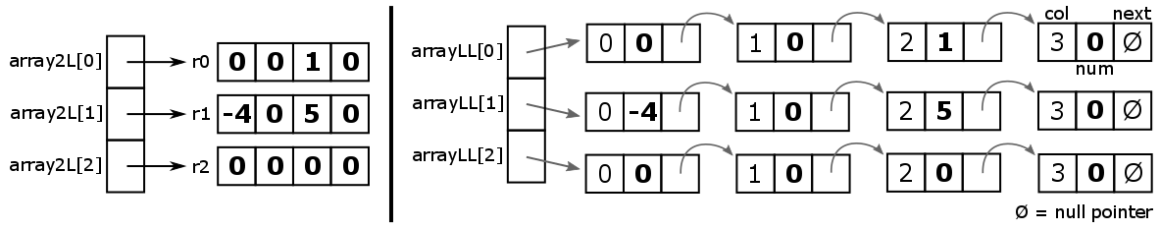
Consider the case where $M = 3$ and $N = 4$. The declarations are given below:

2-dimensional array:	**2-level array:**	**Array of linked lists:**

`int array2D[3][4];`	`int r0[4], r1[4], r2[4];`
	`int* array2L[] = {r0,r1,r2};`
		`struct node {`
		`  int col, num;`
		`  struct node* next;`
		`};`
		`struct node* arrayLL[3];`
		`// code to build out LLs`

4 bytes each
8 byte pointer
note each struct nod

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`:



Ints are 4 bytes and pointers are 8. Each struct node is going to be 16 bytes

a) Fill in the following comparison chart:

	2-dim array	2-level array	Array of LLs:
Overall Memory Used	12*sizeof(int) = 48bytes	12*sizeof(int) for the subarrays + 3 * sizeof(int*) for pointers = 72 bytes	24 * sizeof(struct node) for the nodes and 3 * sizeof(pointer) for the array pointing to the start of each row == 216 bytes
Largest *guaranteed* continuous chunk of memory	48 bytes (whole array is contiguous)	24 bytes (the array of pointers)	24 bytes (the array of pointers
Smallest *guaranteed* continuous chunk of memory	48 bytes (whole array)	16 bytes (one of the arrays for a given row)	16 bytes (a single struct node)
Data type returned by:	`array2D[1]` int*	`array2L[1]` int*	`arrayLL[1]` struct node *
Number of memory accesses to get `int` in the *BEST* case	1	2 (ptr address then index in row)	2 (ptr address then first element in a row)
Number of memory accesses to get `int` in the *WORST* case	1	2	1 + N (ptr address then last element in a row)

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?

We cannot save space because of the alignment issue. Namely, the int num has to align to an address that is a multiple of 4, and the struct itself has to align to an address that is a multiple of 8 (the size of the pointer struct node* next); so we would have to fill the space left by col anyways