CSE 351 Section 5 – Arrays and Buffer Overflow

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).

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

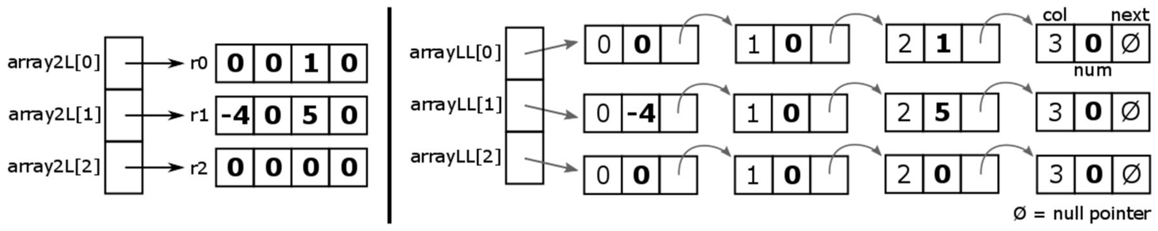
1) 2-dimensional array **int** `array2D[][]`, // M*N array of ints
2) 2-level array **int*** `array2L[]`, and // M array of int arrays
3) array of linked lists **struct node*** `arrayLL[]`. // M array of linked lists (struct node)

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`

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 *guaranteed* continuous chunk of memory			
Smallest *guaranteed* continuous chunk of memory			
Data type returned by:	`array2D[1]`	`array2L[1]`	`arrayLL[1]`
Number of memory accesses to get `int` in the *BEST* case			
Number of memory accesses to get `int` in the *WORST* 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?

c) Provide a scenario where a 2-dimensional array would be more useful and another where a 2-level array would be more useful.

d) Sam wants to create a 2-D matrix of the countries of the world that can be accessed alphabetically. Which implementation should Sam choose to represent this information? Describe what this implementation would look like.

$$\begin{bmatrix} Afghanistan & Albania & \dots & Azerbaijan \\ Bahamas & \dots & Burundi & --- \\ \vdots & \vdots & \vdots & \vdots \\ Zambia & Zimbabwe & --- & --- \end{bmatrix}$$

Buffer Overflow

Consider the following C program:

```
void main() {
  read_input();
}

int read_input() {
  char buf[8];
  gets(buf);
  return 0;
}
```

Address	Value (hex)
%rsp+15	00
%rsp+14	00
%rsp+13	00
%rsp+12	00
%rsp+11	00
%rsp+10	40
%rsp+9	AF
%rsp+8	3B
%rsp+7	
%rsp+6	
%rsp+5	
%rsp+4	
%rsp+3	
%rsp+2	
%rsp+1	
%rsp+0	

Here is a diagram of the stack in `read_input()` right before the call to `gets()`:

a) What is the value of the return address stored on the stack?

Assume that the user inputs the string "`jklmnopqrs`"

b) Write the values in the stack before the "return 0;" statement is executed. Cross out the values that were overwritten and write in their new values.
(Hint: use the ASCII table at the bottom to convert from letters to bytes)

c) What is the new return address after the call to `gets()`?

d) Where will execution jump to after the "`return 0;`"?

e) How many characters would we have to enter into the command line to overwrite the return address to `0x6A6B6C6D6E6F`?

f) Create a string that will overwrite the return address, setting it to `0x6A6B6C6D6E6F`

Char	Hex
a	61
b	62
c	63
d	64
e	65
f	66
g	67
h	68
i	69
j	6A
k	6B
l	6C
m	6D
n	6E
o	6F
p	70
q	71
r	72
s	73
t	74
u	75
v	76
w	77
x	78
y	79
z	7A

In Lab 3, we are given a tool called `sendstring`, which converts hex digits into the actual bytes

```
>echo "61 62 63" | ./sendstring
abc
```

g) If we want to overwrite the return address to a stack address like 0x7FFFFFAB1234, we need to use a tool like sendstring to send the correct bytes.
Why can't we just manually type the characters like we did earlier with "`jklmnopqrs`"?

Check out the Lab 3 video on Phase 0 before you start the lab! It's linked on the Lab 3 page