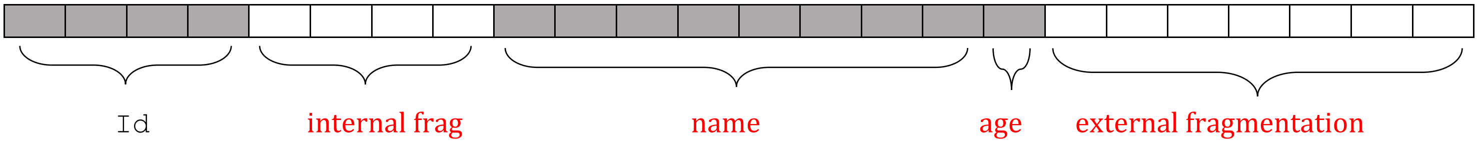
CSE 351 Section 6 Solutions – Arrays and Structs

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

```
struct Student {
  int id;
  char* name;
  char age;
};
```

a) Fill in which bytes are used by which variables and label the rest as internal or external fragmentation. The first variable "id" is given.



b) What is the size of `struct Student`? **24 bytes**

c) Give a reordering of the fields in `struct Student` such that there is no internal fragmentation

```
struct Student {

     char* name;
     int id;
     char age;

};
```

d) How much external fragmentation does this new `struct Student` have? **3 bytes**

e) What is the size of this new `struct Student`? **16 bytes** (smaller than before)

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

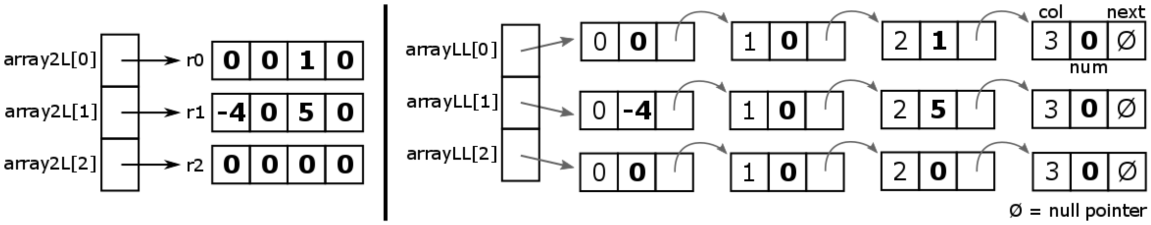
1) 2-dimensional array **`int`** `array2D[][]`, // M*N array of ints
2) 2-level array **`int*`** `array2L[]`, and // M array of int arrays
3) array of linked lists **`struct node*`** `arrayLL[]`. // M array of linked lists (struct node)

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	M*N*sizeof(int) = 48 B	M*N*sizeof(int) + M*sizeof(int *) = 72 B	M*sizeof(struct node *) + M*N*sizeof(struct node) = 216 B
Largest *guaranteed* continuous chunk of memory	The whole array (48 B)	The array of pointers (24 B) > row array (16 B)	The array of pointers (24 B) > struct (16 B)
Smallest *guaranteed* continuous chunk of memory	The whole array (48 B)	Each row array (16 B)	Each struct node (16 B)

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	First node in LL: 2
Number of memory accesses to get `int` in the *WORST* case	1	2	Last node in LL: 5 (we have to read `next`)

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?

No. Alignment requirement of $K = 4$ for **`int`** `num` leaves 3 bytes of internal fragmentation between `col` and `num`.

Buffer Overflow

Consider the following C program:

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

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

Here is a diagram of the stack at the beginning of the call to read_input():

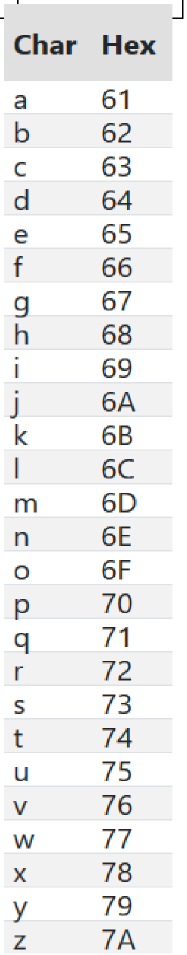
Address	Value (hex)
%rsp+15	00
%rsp+14	00
%rsp+13	00
%rsp+12	00
%rsp+11	00
%rsp+10	~~40~~ 00 (null terminator)
%rsp+9	~~AF~~ 73
%rsp+8	~~3B~~ 72
%rsp+7	71
%rsp+6	70
%rsp+5	6F
%rsp+4	6E
%rsp+3	6D
%rsp+2	6C
%rsp+1	6B
%rsp+0	6A

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

0x40AF3B

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)



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

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

0x7372

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

It will try to jump to 0x7372, but it will crash with a segfault

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

14 = 8 for padding (the length of buf) + 6 for the length of the address in bytes. A null terminator is appended, but it's okay because the upper bytes were going to be 0x00 anyway

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

"`ababababonmlkj`" (The first 8 characters don't matter since they're just padding)

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`"?

There is no character in ASCII we can type that will give us a byte value of 0x7F, 0xFF, or 0x12

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