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.

```
Id    internal frag    name    age    external fragmentation
```

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:

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

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`:

For the comparison chart:

<table>
<thead>
<tr>
<th>Overall Memory Used</th>
<th>Largest guaranteed continuous chunk of memory</th>
<th>Smallest guaranteed continuous chunk of memory</th>
<th>Data type returned by</th>
<th>Number of memory accesses to get ( \text{int} ) in the ( \text{BEST} ) case</th>
<th>Number of memory accesses to get ( \text{int} ) in the ( \text{WORST} ) case</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>M*N*sizeof(int) = 48 B</code></td>
<td>The whole array (48 B)</td>
<td>The whole array (48 B)</td>
<td><code>array2D[1] int *</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>M*N*sizeof(int) + M*sizeof(int*) = 72 B</code></td>
<td>The array of pointers (24 B) &gt; row array (16 B)</td>
<td>Each row array (16 B)</td>
<td><code>array2L[1] int *</code></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><code>M*sizeof(struct node*) + M*N*sizeof(struct node) = 216 B</code></td>
<td>The array of pointers (24 B) &gt; struct (16 B)</td>
<td>Each struct node (16 B)</td>
<td><code>arrayLL[1] struct node*</code></td>
<td>First node in LL: 2</td>
<td>Last node in LL: 5 (we have to read next)</td>
</tr>
</tbody>
</table>

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:

```c
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():

- What is the value of the return address stored on the stack?
  0x40AF3B

Assume that the user inputs the string “jklmnopqrs”

- 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)

- What is the new return address after the call to gets()?
  0x7372

- Where will execution jump to after the “return 0;”?
  It will try to jump to 0x7372, but it will crash with a segfault

- 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

- 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
```

- If we want to overwrite the return address to a stack address like 0x7FFFFFA1234, 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp+15</td>
<td>00</td>
</tr>
<tr>
<td>%rsp+14</td>
<td>00</td>
</tr>
<tr>
<td>%rsp+13</td>
<td>00</td>
</tr>
<tr>
<td>%rsp+12</td>
<td>00</td>
</tr>
<tr>
<td>%rsp+11</td>
<td>00</td>
</tr>
<tr>
<td>%rsp+10</td>
<td>40 00 (null terminator)</td>
</tr>
<tr>
<td>%rsp+9</td>
<td>AF 73</td>
</tr>
<tr>
<td>%rsp+8</td>
<td>3B 72</td>
</tr>
<tr>
<td>%rsp+7</td>
<td>71</td>
</tr>
<tr>
<td>%rsp+6</td>
<td>70</td>
</tr>
<tr>
<td>%rsp+5</td>
<td>6F</td>
</tr>
<tr>
<td>%rsp+4</td>
<td>6E</td>
</tr>
<tr>
<td>%rsp+3</td>
<td>6D</td>
</tr>
<tr>
<td>%rsp+2</td>
<td>6C</td>
</tr>
<tr>
<td>%rsp+1</td>
<td>6B</td>
</tr>
<tr>
<td>%rsp+0</td>
<td>6A</td>
</tr>
</tbody>
</table>

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