CS 31: Introduction to Computer Systems

14-15: Arrays and Pointers
March 21-26
Announcements

• Everything up to Lab 5 graded
  – Check Github repos for comments
  – Check Gradesource for grades
• Midterm debrief last 15 minutes
• Final Exam Time Posted:
  – May 12 9 – 12pm SCI 199
• Please choose partners for Lab 7!
Data Collections in C

• Many complex data types out there (CS 35)

• C has a few simple ones built-in:
  – Arrays
  – Structures (struct)
  – Strings (arrays of characters)

• Often combined in practice, e.g.:
  – An array of structs
  – A struct containing strings
Today

• Accessing *things* via an offset
  – Arrays, Structs, Unions

• How complex structures are stored in memory
  – Multi-dimensional arrays & Structs
So far: Primitive Data Types

• We’ve been using ints, floats, chars, pointers

• Simple to place these in memory:
  – They have an unambiguous size
  – They fit inside a register*
  – The hardware can operate on them directly

(*There are special registers for floats and doubles that use the IEEE floating point format.)
Composite Data Types

• Combination of one or more existing types into a new type. (e.g., an array of multiple ints, or a struct)

• Example: a queue
  – Might need a value (int) plus a link to the next item (pointer)

```c
struct queue_node{
    int value;
    struct queue_node *next;
}
```
Recall: Arrays in Memory

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```

[Diagram showing memory allocation with iptr[0] to iptr[3]]
Recall: Assembly While Loop

```
movl $0 eax //return value
movl $0 edx //loop counter
loop:
    addl (%ecx), %eax
    addl $4, %ecx
    addl $1, %edx
    cmpl $5, %edx
    jne loop
```

Using (*dereferencing*) the memory address to access memory at that location.

ecx was a pointer to the beginning of the array. Manipulating the pointer to point to something else.

Note: This did NOT read or write the memory that is pointed to.
Pointer Manipulation: Necessary?

- Previous example: advance `%ecx` to point to next item in array.

```c
iptr = malloc(...);
sum = 0;
while (i < 4) {
    sum += *iptr;
    iptr += 1;
    i += 1;
}
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iptr[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iptr[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iptr[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iptr[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Moves +1 by size of the data type!
Pointer Manipulation: Necessary?

- Previous example: advance `%ecx` to point to next item in array.

```c
iptr = malloc(...);
sum = 0;
while (i < 4) {
    sum += *iptr;
    iptr += 1;
    i += 1;
}
```

Reminder: addition on a pointer advances by that many of the type (e.g., ints), not bytes.
Pointer Manipulation: Necessary?

- Problem: `iptr` is changing!
- What if we wanted to free it?
- What if we wanted something like this:
  ```c
  iptr = malloc(...);
  sum = 0;
  while (i < 4) {
    sum += iptr[0] + iptr[i];
    iptr += 1;
    i += 1;
  }
  ```
  Changing the pointer would be really inconvenient now!
Base + Offset

• We know that arrays act as a pointer to the first element. For bucket \([N]\), we just skip forward \(N\).

```
int val[5];
```

![Diagram of an array with elements val[0] to val[4]]

• “We’re goofy computer scientists who count starting from zero.”
Base + Offset

- We know that arrays act as a pointer to the first element. For bucket \([N]\), we just skip forward \(N\).

```
int val[5];
```

```
```

- “We’re goofy computer scientists who count starting from zero.”
Base + Offset

- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.

```java
int val[5];
```

This is why we start counting from zero! Skipping forward with an offset of zero ([0]) gives us the first bucket...
Which expression would compute the address of iptr[3]?

A. 0x0824 + 3 * 4
B. 0x0824 + 4 * 4
C. 0x0824 + 0xC
D. More than one (which?)
E. None of these
Which expression would compute the address of iptr[3]?

A. 0x0824 + 3 * 4
B. 0x0824 + 4 * 4
C. 0x0824 + 0xC
D. More than one (which?)
E. None of these
Which expression would compute the address of iptr[3]?

A. 0x0824 + 3 * 4 (requires an extra multiplication step)
B. 0x0824 + 4 * 4
C. 0x0824 + 0xC
D. More than one (which?)
E. None of these

What if this isn’t known at compile time?

Heap

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824</td>
<td>iptr[0]</td>
</tr>
<tr>
<td>0x0828</td>
<td>iptr[1]</td>
</tr>
<tr>
<td>0x082C</td>
<td>iptr[2]</td>
</tr>
<tr>
<td>0x0830</td>
<td>iptr[3]</td>
</tr>
</tbody>
</table>
Indexed Addressing Mode

• What we’d like in IA32 is to express accesses like iptr[N], where iptr doesn’t change – it’s a base.

• Displacement mode works, if we know which offset to use at compile time:
  – Variables on the stack: -4(%ebp)
  – Function arguments: 8(%ebp)

• If we only know at run time?
  – How do we express i(%ecx)?
Indexed Addressing Mode

- **General form:**
  \[ \text{displacement}(\%\text{base, } \%\text{index, scale}) \]

- **Translation:** Access the memory at address...
  - base + (index * scale) + displacement

- **Rules:**
  - Displacement can be any 1, 2, or 4-byte value
  - Scale can be 1, 2, 4, or 8.
Suppose \( i \) is at \( \%ebp - 8 \), and equals 2.

**User says:**

\[
iptr[i] = 9;
\]

**Translates to:**

\[
\texttt{movl} -8(\%ebp), \%edx
\]

**Registers:**

<table>
<thead>
<tr>
<th>ECX: Array base address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
</tr>
<tr>
<td>%edx</td>
</tr>
</tbody>
</table>

**Heap**

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824:</td>
<td>iptr[0]</td>
</tr>
<tr>
<td>0x0828:</td>
<td>iptr[1]</td>
</tr>
<tr>
<td>0x082C:</td>
<td>iptr[2]</td>
</tr>
<tr>
<td>0x0830:</td>
<td>iptr[3]</td>
</tr>
</tbody>
</table>

Example
Suppose i is at %ebp - 8, and equals 2.

User says:
\[ \text{iptr}[i] = 9; \]

Translates to:
\[ \text{movl } -8(\%ebp), \%edx \]
Example

Suppose i is at %ebp - 8, and equals 2.

User says:

iptr[i] = 9;

Translates to:

movl -8(%ebp), %edx
movl $9, (%ecx, %edx, 4)

Registers:

<table>
<thead>
<tr>
<th>%ecx</th>
<th>0x0824</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>2</td>
</tr>
</tbody>
</table>

Heap

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824</td>
<td>iptr[0]</td>
</tr>
<tr>
<td>0x0828</td>
<td>iptr[1]</td>
</tr>
<tr>
<td>0x082C</td>
<td>iptr[2]</td>
</tr>
<tr>
<td>0x0830</td>
<td>iptr[3]</td>
</tr>
</tbody>
</table>

Slide 29
Example

Suppose \( i \) is at \( \%ebp - 8 \), and equals 2.

User says:
\[
\text{iptr}[i] = 9;
\]

Translates to:
\[
\text{movl} \ -8(\%ebp), \ %edx \\
\text{movl} \ \$9, \ (%ecx, \ %edx, \ 4)
\]

\[
0x0824 + (2 \times 4) + 0 = 0x0828 \\
0x0824 + 8 = 0x082C
\]
Example:

Suppose i is at %ebp - 8, and equals 2.

User says:

\[ iptr[i] = 9; \]

Translates to:

\[ \text{movl } -8(\%ebp), \%edx} \]
\[ \text{movl } \$9, (\%ecx, \%edx, 4) \]

\[ 0x0824 + (2 \times 4) + 0 \]
\[ 0x0824 + 8 = 0x082C \]

Allowed us to preserve ecx, and compute an offset without changing the pointer to the base of our array

<table>
<thead>
<tr>
<th>Registers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
</tr>
<tr>
<td>%edx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824: iptr[0]</td>
</tr>
<tr>
<td>0x0828: iptr[1]</td>
</tr>
<tr>
<td>0x082C: iptr[2]</td>
</tr>
<tr>
<td>0x0830: iptr[3]</td>
</tr>
</tbody>
</table>

Example:

\%ecx 0x0824
\%edx 2
What is the final state after this code?

```assembly
addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)
```

Displacement: \( \text{base} + (\text{index} \times \text{scale}) + \text{displacement} \)

<table>
<thead>
<tr>
<th>Registers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
</tr>
<tr>
<td>%ecx</td>
</tr>
<tr>
<td>%edx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory: Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2464: 5</td>
</tr>
<tr>
<td>0x2468: 1</td>
</tr>
<tr>
<td>0x246C: 42</td>
</tr>
<tr>
<td>0x2470: 3</td>
</tr>
<tr>
<td>0x2474: 9</td>
</tr>
</tbody>
</table>

What is the final state after this code?

```
addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)
```

(Initial state)

<table>
<thead>
<tr>
<th>Registers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
</tr>
<tr>
<td>%ecx</td>
</tr>
<tr>
<td>%edx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory: Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0x2464: 5</td>
</tr>
<tr>
<td>0x2468: 1</td>
</tr>
<tr>
<td>0x246C: 42</td>
</tr>
<tr>
<td>0x2470: 3</td>
</tr>
<tr>
<td>0x2474: 9</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
What is the final state after this code?

addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)

Add 4 to eax = 0x2468

(Initial state)
Registers:
%eax 0x2464
%ecx 0x246C
%edx 7

Memory:
Heap
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2464</td>
<td>5</td>
</tr>
<tr>
<td>0x2468</td>
<td>1</td>
</tr>
<tr>
<td>0x246C</td>
<td>42</td>
</tr>
<tr>
<td>0x2470</td>
<td>3</td>
</tr>
<tr>
<td>0x2474</td>
<td>9</td>
</tr>
</tbody>
</table>

Slide 34
What is the final state after this code?

```
addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)
```

1. Add 4 to %eax = 0x2468

(Initial state)

<table>
<thead>
<tr>
<th>Registers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
</tr>
<tr>
<td>%ecx</td>
</tr>
<tr>
<td>%edx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory: Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2464: 5</td>
</tr>
<tr>
<td>0x2468: 1</td>
</tr>
<tr>
<td>0x246C: 42</td>
</tr>
<tr>
<td>0x2470: 3</td>
</tr>
<tr>
<td>0x2474: 9</td>
</tr>
</tbody>
</table>
What is the final state after this code?

```
addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)
```

1. Add 4 to %eax = 0x2468
2. Overwriting the value of eax with 1
What is the final state after this code?

```
addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)
```

1. Add 4 to %eax = 0x2468
2. Overwriting the value of eax with 1
What is the final state after this code?

```
addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)
```

1. Add 4 to %eax = 0x2468
2. Overwriting the value of eax with 1
3. shifting left by 1 = overwriting to 2
What is the final state after this code?

displacement(%base, %index, scale)  
base + (index * scale) + displacement

1. Add 4 to %eax = 0x2468
2. Overwriting the value of eax with 1
3. shifting left by 1 = overwriting to 2
What is the final state after this code?

displacement(%base, %index, scale)
base + (index * scale) + displacement

1. Add 4 to %eax = 0x2468
2. Overwriting the value of %eax with 1
3. shifting left by 1 = overwriting to 2
4. 0x246C + 2*2 = 0x2470

moving edx to the memory address
What is the final state after this code?

displacement(%base, %index, scale)
base + (index * scale) + displacement

```
addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)
```

1. Add 4 to %eax = 0x2468
2. Overwriting the value of %eax with 1
3. shifting left by 1 = overwriting to 2
4. 0x246C + 2*2 = 0x2470

Moving edx to the memory address

Slide 41
Indexed Addressing Mode

• General form:
  displacement(%base, %index, scale)

• You have seen these probably in your maze.
Two-dimensional Arrays

• Why stop at an array of ints?
  How about an array of arrays of ints?

```java
int twodims[3][4];
```

• “Give me three sets of four integers.”

• How should these be organized in memory?
Two-dimensional Arrays

```c
int twodims[3][4];
for (i=0; i<3; i++) {
    for (j=0; j<4; j++) {
        twodims[i][j] = i+j;
    }
}
```

Slide 44
Two-dimensional Arrays: Matrix

```c
int twodims[3][4];
for(i=0; i<3; i++) {
    for(j=0; j<4; j++) {
        twodims[i][j] = i+j;
    }
}
```
Memory Layout

- Matrix: 3 rows, 4 columns

Row Major Order:
- all Row 0 buckets, followed by
- all Row 1 buckets

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf260</td>
<td>0</td>
<td>twodim[0][0]</td>
<td></td>
</tr>
<tr>
<td>0xf264</td>
<td>1</td>
<td>twodim[0][1]</td>
<td></td>
</tr>
<tr>
<td>0xf268</td>
<td>2</td>
<td>twodim[0][2]</td>
<td></td>
</tr>
<tr>
<td>0xf26c</td>
<td>3</td>
<td>twodim[0][3]</td>
<td></td>
</tr>
<tr>
<td>0xf270</td>
<td>1</td>
<td>twodim[1][0]</td>
<td></td>
</tr>
<tr>
<td>0xf274</td>
<td>2</td>
<td>twodim[1][1]</td>
<td></td>
</tr>
<tr>
<td>0xf278</td>
<td>3</td>
<td>twodim[1][2]</td>
<td></td>
</tr>
<tr>
<td>0xf27c</td>
<td>4</td>
<td>twodim[1][3]</td>
<td></td>
</tr>
<tr>
<td>0xf280</td>
<td>2</td>
<td>twodim[2][0]</td>
<td></td>
</tr>
<tr>
<td>0xf284</td>
<td>3</td>
<td>twodim[2][1]</td>
<td></td>
</tr>
<tr>
<td>0xf288</td>
<td>4</td>
<td>twodim[2][2]</td>
<td></td>
</tr>
<tr>
<td>0xf28c</td>
<td>5</td>
<td>twodim[2][3]</td>
<td></td>
</tr>
</tbody>
</table>
Memory Layout

• Matrix: 3 rows, 4 columns

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
```

`twodim[1][3]:`

base addr + row offset + col offset

```
0xf260 + 16 + 12 = 0xf27c
```

```
<table>
<thead>
<tr>
<th>0x260</th>
<th>0</th>
<th>twodim[0][0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x264</td>
<td>1</td>
<td>twodim[0][1]</td>
</tr>
<tr>
<td>0x268</td>
<td>2</td>
<td>twodim[0][2]</td>
</tr>
<tr>
<td>0x26c</td>
<td>3</td>
<td>twodim[0][3]</td>
</tr>
<tr>
<td>0x270</td>
<td>1</td>
<td>twodim[1][0]</td>
</tr>
<tr>
<td>0x274</td>
<td>2</td>
<td>twodim[1][1]</td>
</tr>
<tr>
<td>0x278</td>
<td>3</td>
<td>twodim[1][2]</td>
</tr>
<tr>
<td>0x27c</td>
<td>4</td>
<td>twodim[1][3]</td>
</tr>
<tr>
<td>0x280</td>
<td>2</td>
<td>twodim[2][0]</td>
</tr>
<tr>
<td>0x284</td>
<td>3</td>
<td>twodim[2][1]</td>
</tr>
<tr>
<td>0x288</td>
<td>4</td>
<td>twodim[2][2]</td>
</tr>
<tr>
<td>0x28c</td>
<td>5</td>
<td>twodim[2][3]</td>
</tr>
</tbody>
</table>
```
Memory Layout

- **Matrix**: 3 rows, 4 columns

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
```

```plaintext
twodim[1][3]:

base addr + row offset + col offset

twodim + 1*ROWSIZE*4 + 3*4

0xf260 + 16 + 12 = 0xf27c
```
Memory Layout

- Matrix: 3 rows, 4 columns

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
1 & 2 & 3 & 4 \\
2 & 3 & 4 & 5 \\
\end{array}
\]

twodim[1][3]:

\[
\text{base addr + row offset + col offset}
\]

\[
\text{twodim + 1*ROWSIZE*4 + 3*4}
\]

\[
0xf260 + 16 + 12 = 0xf27c
\]
If we declared `int matrix[5][3];`, and the base of matrix is 0x3420, what is the address of `matrix[3][2]`?

A. 0x3438  
B. 0x3440  
C. 0x3444  
D. 0x344C  
E. None of these

(base addr + row offset + col offset)
If we declared `int matrix[5][3];`, and the base of matrix is 0x3420, what is the address of `matrix[3][2]`?

A. 0x3438  
B. 0x3440  
C. 0x3444  
D. 0x344C  
E. None of these

\[
0x3420 + 3 \times \text{ROWSIZE} \times 4 \text{ (int data type)} + 2 \times 2 \text{ (2 ints forward)} \times 4 \text{ (int data type)}
\]
If we declared `int matrix[5][3];`, and the base of matrix is `0x3420`, what is the address of `matrix[3][2]`?

A. 0x3438  
B. 0x3440  
C. 0x3444  
D. 0x344C  
E. None of these

\[
0x3420 + 3 \times \text{ROWSIZE} \times 4 \text{(int data type)} + 2 \times (2 \text{ ints forward}) \times 4 \text{(int data type)} \\
0x3420 + [36 + 8 = (44) = 0x2C] = 0x344C
\]
Composite Data Types

• Combination of one or more existing types into a new type. (e.g., an array of multiple ints, or a struct)

• Example: a queue
  – Might need a value (int) plus a link to the next item (pointer)

```c
struct queue_node{
    int value;
    struct queue_node *next;
};
```
Structs

• Laid out contiguously by field
  – In order of field declaration.

```c
struct student{
    int age;
    float gpa;
    int id;
};

struct student s;
```
Structs

- Struct fields accessible as a base + displacement
  - Compiler knows (constant) displacement of each field

```c
struct student{
    int age;
    float gpa;
    int id;
};
```

```c
struct student s;
```

```
0x1234
s.age

0x1238
s.gpa

0x123c
s.id
```

Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.

```c
struct student{
    int age;
    float gpa;
    int id;
};

struct student s;
```

Memory

```
0x1234  s.age
0x1238  s.gpa
0x123c  s.id
...     ...```

Slide 56
Data Alignment:

• Where (which address) can a field be located?

• **char (1 byte):** can be allocated at any address:
  0x1230, 0x1231, 0x1232, 0x1233, 0x1234, ...

• **short (2 bytes):** must be aligned on 2-byte addresses:
  0x1230, 0x1232, 0x1234, 0x1236, 0x1238, ...

• **int (4 bytes):** must be aligned on 4-byte addresses:
  0x1230, 0x1234, 0x1238, 0x123c, 0x1240, ...
Why do we want to align data on multiples of the data size?

A. It makes the hardware faster.

B. It makes the hardware simpler.

C. It makes more efficient use of memory space.

D. It makes implementing the OS easier.

E. Some other reason.
Why do we want to align data on multiples of the data size?

A. It makes the hardware faster.

B. It makes the hardware simpler.

C. It makes more efficient use of memory space.

D. It makes implementing the OS easier.

E. Some other reason.
Data Alignment: Why?

• Simplify hardware
  – e.g., only read ints from multiples of 4
  – Don’t need to build wiring to access 4-byte chunks at any arbitrary location in hardware

• Inefficient to load/store single value across alignment boundary (1 vs. 2 loads)

• Simplify OS:
  – Prevents data from spanning virtual pages
  – Atomicity issues with load/store across boundary
Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.

```c
struct student{
    int age;
    float gpa;
    int id;
};
```

```c
struct student s;
```

Slide 61
struct student {
    char name[11];
    short age;
    int id;
};
How much space do we need to store one of these structures?

```
struct student{
    char name[11];
    short age;
    int id;
};
```

A. 17 bytes  
B. 18 bytes  
C. 20 bytes  
D. 22 bytes  
E. 24 bytes
**Structs**

```c
struct student{
    char name[11];
    short age;
    int id;
};
```

- Size of data: 17 bytes
- Size of struct: 20 bytes

Use `sizeof()` when allocating structs with `malloc()`!
Alternative Layout

```c
struct student{
    int id;
    short age;
    char name[11];
};
```

Same fields, declared in a different order.
struct student{
    int id;
    short age;
    char name[11];
};

• Size of data: 17 bytes
• Size of struct: 17 bytes!

In general, this isn’t a big deal on a day-to-day basis. Don’t go out and rearrange all your struct declarations.
Cool, so we can get rid of this padding by being smart about declarations?

A. Yes (why?)

B. No (why not?)
Cool, so we can get rid of this padding by being smart about declarations?

A. Yes (why?)

B. No (why not?)
Cool, so we can get rid of this padding by being smart about declarations?

- **Answer:** Maybe.
- **Rearranging helps, but often padding after the struct can’t be eliminated.**

```c
struct T1 {
    char c1;
    int x;
    char c2;
    int x;
};
```

```c
struct T2 {
    int x;
    char c1;
    char c2;
};
```
“External” Padding

• Array of Structs

Field values in each bucket must be properly aligned:

```c
struct T2 arr[3];
```

Buckets must be on a 4-byte aligned address
A note on struct syntax...

```c
struct student {
    int id;
    short age;
    char name[11];
};

struct student s;

s.id = 406432;
s.age = 20;
strcpy(s.name, "Alice");
```
A note on struct syntax…

struct student {
    int id;
    short age;
    char name[11];
};

struct student *s = malloc(sizeof(struct student));

(*s).id = 406432;
(*s).age = 20;
strcpy((*s).name, "Alice");

s->id = 406432;
s->age = 20;
strcpy(s->name, "Alice");

Not a struct, but a pointer to a struct!

This works, but is very ugly.

Access the struct field from a pointer with ->
Does a dereference and gets the field.
Stack Padding

• Memory alignment applies elsewhere too.

```c
int x;         vs.     double y;
char ch[5];
short s;
double y;
```

```c
int x;
short s;
char ch[5];
```
Unions

• Declared like a struct, but only contains one field, rather than all of them.

  • Struct: field 1 and field 2 and field 3 ...
  • Union: field 1 or field 2 or field 3 ...

• Intuition: you know you only need to store one of N things, don’t waste space.
Unions

struct my_struct {
    char ch[2];
    int i;
    short s;
}

union my_union {
    char ch[2];
    int i;
    short s;
}

my_struct in memory

my_union in memory

Same memory used for all fields!
Unions

```c
my_union u;
u.i = 7;
```

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!
Unions

my_union u;

u.i = 7;

u.s = 2;

union my_union {
  char ch[2];
  int i;
  short s;
}

Same memory used for all fields!

my_union in memory
Unions

my_union u;

u.i = 7;

u.s = 2;

u.ch[0] = ‘a’;

Reading i or s here would be bad!

union my_union {
    char ch[2];
    int i;
    short s;
}

Same memory used for all fields!

my_union in memory
Unions

```c
union my_union {
    char ch[2];
    int i;
    short s;
}

my_union u;

u.i = 7;
u.s = 2;
u.ch[0] = 'a';

Reading i or s here would be bad!

u.i = 5;
```

Same memory used for all fields!

my_union in memory
Unions

• You probably won’t use these often.

• Use when you need mutually exclusive types.

• Can save memory.

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!

my_union in memory
Recall: Characters and Strings

- A character (type `char`) is a numerical value that holds one letter.
  
  ```c
  char my_letter = 'w'; // Note: single quotes
  ```

- What is the numerical value?
  - `printf("%d   %c", my_letter, my_letter);`
  - Would print: 119 w

- Why is ‘w’ equal to 119?
  - ASCII Standard says so.
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Null</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Start of heading</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>Start of text</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>End of text</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>End of transmit</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>Enquiry</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>Acknowledge</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>Audible bell</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>Backspace</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>Horizontal tab</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>Line feed</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>Vertical tab</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>Form feed</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>Carriage return</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>Shift out</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>Shift in</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Data link escape</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Device control 1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Device control 2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Device control 3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Device control 4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Neg. acknowledge</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>Synchronous idle</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>End trans. block</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>Cancel</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>End of medium</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Substitution</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>Escape</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>\</td>
<td>123</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>File separator</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td></td>
<td></td>
<td>124</td>
<td>7C</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Group separator</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>}</td>
<td>125</td>
<td>7D</td>
<td>}</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>Record separator</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>Unit separator</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
<td>127</td>
<td>7F</td>
<td>□</td>
</tr>
</tbody>
</table>
Recall: Characters and Strings

- A character (type `char`) is a numerical value that holds one letter.
- A string is a memory block containing characters, one after another...

- Examples:
  ```
  char name[6] = “Pizza”;
  ```

Hm, suppose we used `printf` and `%s` to print `name`.

How does it know where the string ends and other memory begins?
Recall: How can we tell where a string ends?

A. Mark the end of the string with a special character.

B. Associate a length value with the string, and use that to store its current length.

C. A string is always the full length of the array it’s contained within (e.g., char name[20] must be of length 20).

D. All of these could work (which is best?).

E. Some other mechanism (such as?).
Recall: How can we tell where a string ends?

A. Mark the end of the string with a special character (what C does).

B. Associate a length value with the string, and use that to store its current length.

C. A string is always the full length of the array it’s contained within (e.g., char name[20] must be of length 20).

D. All of these could work (which is best?).

E. Some other mechanism (such as?).
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Null</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>!</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>&quot;</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>#</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>c</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>$</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>d</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>%</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>e</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>&amp;</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>f</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>'</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>g</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>(</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>h</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>)</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>i</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>Line feed</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>Vertical tab</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>Form feed</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>Carriage return</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>Shift out</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>Shift in</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Data link escape</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Device control 1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Device control 2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Device control 3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Device control 4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Neg. acknowledge</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>Synchronous idle</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>End trans. block</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>Cancel</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>End of medium</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Substitution</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>Escape</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
<td>123</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>File separator</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Group separator</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
<td>125</td>
<td>7D</td>
<td>}</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>Record separator</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>Unit separator</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>`</td>
<td>127</td>
<td>7F</td>
<td>□</td>
</tr>
</tbody>
</table>

**Special stuff over here in the lower values.**
Recall: Characters and Strings

- A character (type `char`) is a numerical value that holds one letter.
- A string is a memory block containing characters, one after another, with a null terminator (numerical 0) at the end.
- Examples:
  ```
  char name[20] = "Pizza";
  ```
Recall: Strings in C

- C String library functions: `#include <string.h>`
  - Common functions (strlen, strcpy, etc.) make strings easier
  - Less friendly than Python strings

- More on strings later, in labs.

- For now, remember about strings:
  - Allocate enough space for null terminator!
  - If you’re modifying a character array (string), don’t forget to set the null terminator!
  - If you see crazy, unpredictable behavior with strings, check these two things!
Strings

• Strings are *character arrays*

• Layout is the same as:
  – char name[10];

• Often accessed as (char *)
String Functions

- C library has many built-in functions that operate on char *’s:
  - strcpy, strdup, strlen, strcat, strcmp, strstr

```c
char name[10];
strcpy(name, "CS 31");
```
String Functions

- C library has many built-in functions that operate on char *’s:
  - strcpy, strdup, strlen, strcat, strcmp, strstr

```c
char name[10];
strcpy(name, "CS 31");
```

- Null terminator (\0) ends string.
  - We don’t know/care what comes after
String Functions

• C library has many built-in functions that operate on char *’s:
  – strcpy, strdup, strlen, strcat, strcmp, strstr

• Seems simple on the surface.
  – That null terminator is tricky, strings error-prone.
  – Strings used everywhere!

• You will implement these functions in a future lab.