CS 31: Introduction to Computer Systems

13-14: Arrays and Pointers
March 5
Reading Quiz
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)
**structs**

- Treat a collection of values as a single type:
  - C is not an object oriented language, no classes
  - A struct is like just the data part of a class

- **Rules:**
  1. Define a new **struct** type outside of any function
  2. Declare variables of the new struct type
  3. **Use dot notation to access the different field values** of the struct variable
Struct Example

Suppose we want to represent a `student` type.

```c
struct student {
    char name[20];
    int grad_year;
    float gpa;
};

// Variable bob is of type struct student
struct student bob;

// Set name (string) with strcpy()
strcpy(bob.name, "Robert Paulson");
bob.grad_year = 2019;
bob.gpa = 3.1;

printf("Name: %s, year: %d, GPA: %f", bob.name, bob.grad_year, bob.gpa);
```
Recall: Arrays

• C’s support for **collections of values**
  – Array buckets store a single type of value
  – **Specify max capacity** (num buckets) when you declare an array variable (single memory chunk)
Recall: Arrays

Static Allocation:

\[\text{<type> <var_name>[<num_buckets>]}
\]

\[\text{int arr[5];}
// an array of 5 integers
\]

\[\text{float rates[40];}
// an array of 40 floats
\]

Dynamic Allocation:

\[\text{<type> <var_name>[<num_buckets>]}
\]

\[\text{int * arr = malloc(sizeof(int)*5);}\n// an array of 5 integers
\]

\[\text{//initialize array}
\]

\[\text{//free array}
\]

\[\text{free(arr);}\]
Recall: Pointers as Arrays

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

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
iptr[2] = 7;
```

1. Start from the base of iptr.
Pointers as Arrays

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

1. Start from the base of iptr.
2. Skip forward by the size of two ints.
iptr[2] = 7;
Pointers as Arrays

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

1. Start from the base of iptr.

iptr[2] = 7;

2. Skip forward by the size of two ints.

3. Treat the result as an int. (Access the memory location like a typical dereference.)
Pointer Arithmetic

• Addition and subtraction work on pointers.

• C automatically increments by the size of the type that’s pointed to.
What is the memory address stored in iptr2?

```plaintext
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
int *iptr2 = iptr + 3;
```

A. Mem. address in iptr + 12 bytes  
B. Mem. address in iptr + 3 bytes  
C. Mem. address in iptr + 4 bytes  
D. None of the above
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
Pointer Arithmetic

• Addition and subtraction work on pointers.

• C automatically increments by the size of the type that’s pointed to.
While Loop in C

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

- Moves `+1` by the size of the data type!
Let’s translate the while loop to assembly

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

Assume %ecx = base address of array
%eax = sum
%edx = loop index

```
movl $0 eax
movl $0 edx
loop:
    [fill instructions here]
cmpl $5, %edx
jne loop
```
While Loop in C

```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?**

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

cannot call free on `iptr` since it no longer references the base address of the array!
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!
• We know that arrays act as a pointer to the first element. For bucket \([N]\), we just skip forward \(N\).

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

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

- \(\text{val[0]}\)
- \(\text{val[1]}\)
- \(\text{val[2]}\)
- \(\text{val[3]}\)
- \(\text{val[4]}\)
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.”
We know that arrays act as a pointer to the first element. For bucket \([N]\), we just skip forward \(N\).

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
Indexed Addressing Mode

• We want 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:
  displacement(%base, %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.
Example

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

User says:

$$\text{iptr}[i] = 9;$$

Translates to:

$$\text{movl } -8(\%ebp), \%edx$$

Registers:

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

Heap

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824</td>
<td>iptr[0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0828</td>
<td>iptr[1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x082C</td>
<td>iptr[2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0830</td>
<td>iptr[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Suppose i is at %ebp - 8, and equals 2.

User says:

\[ \text{iptr}[i] = 9; \]

Translates to:

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

<table>
<thead>
<tr>
<th>Stack Frame</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824</td>
<td>ecx</td>
<td>0x824</td>
</tr>
<tr>
<td></td>
<td>edx</td>
<td>2</td>
</tr>
</tbody>
</table>

Heap:

<table>
<thead>
<tr>
<th>Address</th>
<th>iptr[i]</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>
Suppose \( i \) is at \( %\text{ebp} - 8 \), and equals 2.

User says:

\[
\text{iptr}[i] = 9;
\]

Translates to:

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

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

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

User says:

\[ \text{iptr}[i] = 9; \]

Translates to:

\[
\begin{align*}
\text{movl} & \ -8(\%ebp), \ %edx \\
\text{movl} & \ $9, (\%ecx, %edx, 4)
\end{align*}
\]

\[0x0824 + (2 \times 4) + 0 \]

\[0x0824 + 8 = 0x082C\]
Example:

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

User says:

\[
iptr[i] = 9;
\]

Translates to:

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

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

Allowed us to preserve \( %\text{ecx} \), and compute an offset without changing the pointer to the base of our array.
What is the final state after this code?

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

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

(Initial state)

<table>
<thead>
<tr>
<th>Registers:</th>
<th>Memory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax: 0x2464</td>
<td>0x2464: 5</td>
</tr>
<tr>
<td>%ecx: 0x246C</td>
<td>0x2468: 1</td>
</tr>
<tr>
<td>%edx: 7</td>
<td>0x246C: 42</td>
</tr>
<tr>
<td>0x2470: 3</td>
<td>0x2474: 9</td>
</tr>
</tbody>
</table>

Slide 46
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:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>0x2464:</td>
</tr>
<tr>
<td>0x2468:</td>
</tr>
<tr>
<td>0x246C:</td>
</tr>
<tr>
<td>0x2470:</td>
</tr>
<tr>
<td>0x2474:</td>
</tr>
</tbody>
</table>
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?

int twodims[3][4];

• “Give me three sets of four integers.”

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

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

The code initializes a 2D array `twodims` with three rows and four columns. It then fills each element `twodims[i][j]` with the sum of its indices `i+j`. The diagram illustrates the assignment of values to the array elements.
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;
    }
}
```

```
twodims[0]  | 0  | 1  | 2  | 3  
twodims[1]  | 1  | 2  | 3  | 4  
twodims[2]  | 2  | 3  | 4  | 5  
```
Memory Layout

- Matrix: 3 rows, 4 columns

<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>

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>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

<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>

$\text{twodim}[1][3]$: 

- base addr + row offset + col offset

$\text{twodim} + 1 \cdot \text{ROWSIZE} \cdot 4 + 3 \cdot 4$

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

- Matrix: 3 rows, 4 columns

```
0  1  2  3
1  2  3  4
2  3  4  5
```

twodim[1][3]:

```
0xf260  0  twodim[0][0]
0xf264  1  twodim[0][1]
0xf268  2  twodim[0][2]
0xf26c  3  twodim[0][3]
0xf270  1  twodim[1][0]
0xf274  2  twodim[1][1]
0xf278  3  twodim[1][2]
0xf27c  4  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

<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>

\[
twodim[1][3]:
\]

- base addr + row offset + col offset

\[
twodim + 1\times ROWSIZE\times 4 + 3\times 4
\]

\[
0xf260 + 16 + 12 = 0xf27c
\]
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

\[
\text{base addr + row offset + col offset} =\]
\[
\text{or}
\]
\[
\text{base addr + num cols * data size + col offset}
\]