CS 31: Intro to Systems
Arrays, Structs and Pointers

Martin Gagne
Swarthmore College
February 28, 2016
Announcements

• No reading quiz today.

• Midterm in class on Thursday.

• Lab05 checkpoint deadline extended.
  • Checkpoint due Friday 11:59pm.
  • Complete lab due in two weeks (w000… fun break…).
Overview

• 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 list_cell {
    int value;
    struct list_cell *next;
}
```
Recall: Arrays in Memory

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```
Recall: Assembly While Loop

```assembly
movl $0, %eax
movl $0, %edx
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.

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;
}
```

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>iptr[0]</td>
</tr>
<tr>
<td>iptr[1]</td>
</tr>
<tr>
<td>iptr[2]</td>
</tr>
<tr>
<td>iptr[3]</td>
</tr>
</tbody>
</table>
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.
Problem: \texttt{iptr} is changing!

What if we wanted to free it?

What if we wanted something like this:

\begin{verbatim}
iptr = malloc(...);
sum = 0;
i = 0;
while (i < 4) {
    sum += iptr[i];
    i += 1;
}
\end{verbatim}

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];
```

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
Recall: Indexed Addressing Mode

• General form:
  \[ \text{offset}(\%\text{base}, \%\text{index}, \text{scale}) \]

• Translation: Access the memory at address...
  \[ \text{base} + (\text{index} \times \text{scale}) + \text{offset} \]

• Example:
  \[ -0x8(\%\text{ebp}, \%\text{ecx}, 0x4) \]
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:

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

Translates to:

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

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

User says:

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

Translates to:

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

\[
0x0824 + (2 \times 4) + 0 \\
0x0824 + 8 = 0x082C
\]
What is the final state after this code?

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

(Initial state)

Registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x2464</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x246C</td>
</tr>
<tr>
<td>%edx</td>
<td>7</td>
</tr>
</tbody>
</table>

Memory:

<table>
<thead>
<tr>
<th>Offset</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>
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

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

![Diagram of two-dimensional array]

### Memory Layout

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

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

**Row Major Order:**
- all Row 0 buckets,
- followed by all Row 1 buckets
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*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
2D Arrays Another Way

char *arr;
arr = malloc(sizeof(char)*ROWS*COLS);
for(i=0; i< ROWS; i++) {
    for(j=0; j< COLS; j++) {
        arr[i*COLS+j] = i+j;
    }
}

Heap: all ROW*COLS buckets are contiguous
(allocated by a single malloc)
all buckets can be access from single base address (addr)
2D Arrays yet Another Way

char *arr[3]; // array of 3 char *’s
for(i=0; i<3; i++) {
    arr[i] = malloc(sizeof(char)*5);
    for(j=0; j<5; j++) {
        arr[i][j] = i+j;
    }
}

Heap: each malloc’ed array of 5 chars is contiguous, but three separately malloc’ed arrays, not necessarily → each has separate base address
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 (required by C standard).

```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;
};

struct student s;
```

![Memory diagram]

<table>
<thead>
<tr>
<th>Memory</th>
<th>Value</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234</td>
<td>s.age</td>
<td></td>
</tr>
<tr>
<td>0x1238</td>
<td>s.gpa</td>
<td></td>
</tr>
<tr>
<td>0x123c</td>
<td>s.id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Structs

• Laid out contiguously by field
  – In order of field declaration (required by C standard).
  – May require some padding, for alignment.

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

struct student s;
```

Memory

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234</td>
<td>s.age</td>
</tr>
<tr>
<td>0x1238</td>
<td>s.gpa</td>
</tr>
<tr>
<td>0x123c</td>
<td>s.id</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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.
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
struct student{
    char name[11];
    short age;
    int id;
};
How much space do we need to store one of these structures?

```c
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.
Alternative Layout

```
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?

• Answer: Maybe.

• Rearranging helps, but often padding after the struct can’t be eliminated.

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

struct T2 {
    int x;
    char c1;
    char c2;
};
```

<table>
<thead>
<tr>
<th>T1: c1</th>
<th>c2</th>
<th>2bytes</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>x</td>
<td>c1</td>
<td>c2</td>
</tr>
</tbody>
</table>
“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
Which instructions would you use to access the age field of students[8]?

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

struct student students[20];

students[8].age = 21;
```

Assume the base of students is stored in register %edx.
Stack Padding

• Memory alignment applies elsewhere too.

```c
int x;         vs.     double y;
char ch[5];    int x;
short s;       short s;
double y;      char ch[5];
```
What We’ve Learned

CS31: First Half
The Hardware Level

- Basic Hardware Units:
  - Processor
  - Memory
  - I/O devices
- Connected by buses.
Foundational Concepts

• Von Neumann architecture
  • Programs are data.
  • Programs and other data are stored in main memory.

• Binary data representation
  • Data is encoded in binary.
    • Two’s complement
    • ASCII
    • etc.
  • Instructions are encoded in binary.
    • Opcode
    • Source and destination addresses
Architecture and Digital Circuits

• Circuits are built from logic gates.
  • Basic gates: AND, OR, NOT, …

• Three types of circuits:
  • Arithmetic/Logic
  • Storage
  • Control

• The CPU uses all three types of circuits.

• Clock cycle drives the system.
  • One instruction per clock cycle.

• ISA defines which operations are available.
Assembly Language

• Assembly instructions correspond closely to CPU operations.

• Compiler converts C code to assembly instructions.

• Types of instructions:
  • Arithmetic/logic: ADD, OR, …
  • Control Flow: JMP, CALL
  • Data Movement: MOV, (and fake data mvmt: LEAL)
  • Stack & Functions: PUSH, POP, CALL, LEAVE, RET

• Many ways to compile the same program.
  • Conventions govern choices that need to be consistent.
    • Location of function arguments, return address, etc.
C Programming Concepts

• Arrays, structs, and memory layout.

• Pointers and addresses.

• Function calls and stack memory.

• Dynamic memory on the heap.
Some of the (many) things we’ve left out...

• EE level: wires and transistors.
• Optimizing circuits: time and area.
  • Example: a ripple carry adder has a long critical path; can we shorten it?
• Architecture support for complex instructions.
  • Often an assembly instruction requires multiple CPU operations.
• Compiler design.
  • The compiler automates C →IA32 translation. How does this work? How can it be made efficient?
Midterm Info

• Arrive early on Thursday. We will start right at 11:20.
• Bring a pencil.
  • Please don’t use a pen unless you’re REALLY certain of your answer.
• Closed notes, but you may bring the following:
  • IA32 cheat sheet
  • IA32 stack diagram
• Q&A-style review session in lab tomorrow.
  • I will not prepare slides for this.
  • You need to prepare questions to make this useful.
Midterm Tips

• Don’t leave questions blank: a partial answer is better than none.

• If you don’t understand a question, ask for clarification during exam.

• If you’re not sure how to do problem, move on and come back later.

• Use a question’s point value as rough guide for how much time to spend on it.

• Review your answers before turning in the exam.

• Show your work for partial credit.