Structs & Alignment
CSE 351 Winter 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Jonathan Chen
Justin Johnson
Porter Jones
Josie Lee
Jeffery Tian
Callum Walker
Eddy (Tianyi) Zhou

http://xkcd.com/163/
Administrivia

- Lab 2 due tonight (2/07)
  - Lab 3 coming soon!

- **Midterm** Monday (2/10), during lecture

- Mid-quarter survey due Thursday (2/13)

- hw13 due *next* Friday (2/14)
Roadmap

C:

```c
#include <stdlib.h>

Car *c = malloc(sizeof(Car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

Assembly language:

```
get_mpg:
pushq %rbp
movq %rsp, %rbp
...  
popq %rbp
ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111111011000011111
```

OS:

- Windows 10
- OS X Yosemite

Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs

Memory & caches
- Processes
- Virtual memory
- Memory allocation

Java vs. C
Polling Question

- Which of the following statements is FALSE?
  - Vote at http://pollev.com/rea

\[
\text{int } \text{sea}[4][5];
\]

A. \text{sea}[4][–2] is a valid array reference
B. \text{sea}[1][1] makes two memory accesses
C. \text{sea}[2][1] will always be a higher address than \text{sea}[1][2]
D. \text{sea}[2] is calculated using only \text{lea}
E. We’re lost…

\[
\begin{array}{cccccc}
9 & 8 & 1 & 9 & 5 & 9 \\
9 & 8 & 1 & 0 & 5 & 9 \\
8 & 1 & 0 & 3 & 9 & 8 \\
1 & 1 & 1 & 1 & 1 & 5 \\
\end{array}
\]
Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structs**
  - Alignment

- **Unions**
Structs in C

- A structured group of variables, possibly including other structs
  - Way of defining compound data types

```c
struct song {  
    char *title;  
    int lengthInSeconds;  
    int yearReleased;  
};

struct song song1;  
song1.title = "Señorita";  
song1.lengthInSeconds = 191;  
song1.yearReleased = 2019;

struct song song2;  
song2.title = "Call Me Maybe";  
song2.lengthInSeconds = 193;  
song2.yearReleased = 2011;
```
Struct Definitions

- Structure definition:
  - Does NOT declare a variable
  - Variable type is “struct name”

- Variable declarations like any other data type:

```
struct name name1, *pn, name_ar[3];
```

- Can also combine struct and instance definitions:
  - This syntax can be difficult to read, though

```
struct name { /* fields */ } st, *p = &st;
```
Typedef in C

- A way to create an *alias* for another data type:
  ```c
  typedef <data type> <alias>;
  ```
  - After typedef, the alias can be used interchangeably with the original data type
  - *e.g.* `typedef unsigned long int uli;`

- Joint struct definition and typedef
  - Don’t need to give struct a name in this case

```c
struct nm {
    /* fields */
};
typedef struct nm name;
name n1;
```
Scope of Struct Definition

- Why is the placement of struct definition important?
  - What actually happens when you declare a variable?
    - Creating space for it somewhere!
  - Without definition, program doesn’t know how much space

```c
struct data {  
  int ar[4];  
  long d;  
};  
```

Size = 24 bytes

```c
struct rec {  
  int a[4];  
  long i;  
  struct rec* next;  
};  
```

Size = 32 bytes

- Almost always define structs in global scope near the top of your C file
  - Struct definitions follow normal rules of scope
Accessing Structure Members

- Given a struct instance, access member using the \( . \) operator:
  ```c
  struct rec r1;
  r1.i = val;
  ```

- Given a `pointer` to a struct:
  ```c
  struct rec *r;
  r = &r1;  // or malloc space for r to point to
  ```

  We have two options:
  - Use `*` and `.` operators: \((r).i = val\);
  - Use `->` operator for short: \(r->i = val\);

- **In assembly:** register holds address of the first byte
  - Access members with offsets
    ```c
    D(Rb, Ri, S)
    ```

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```
Java side-note

- An instance of a class is like a *pointer to* a struct containing the fields
  - (Ignoring methods and subclassing for now)
  - So Java’s `x.f` is like C’s `x->f` or `(*x).f`

- In Java, almost everything is a pointer ("reference") to an object
  - Cannot declare variables or fields that are structs or arrays
  - Always a *pointer* to a struct or array
  - So every Java variable or field is $\leq 8$ bytes (but can point to lots of data)

```java
class Record { ... }
Record x = new Record();
```
Structure Representation

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} st, *r = &st;
```

- **Characteristics**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Fields may be of different types
Structure Representation

- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering would be more compact
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
} st, *r = &st;
```
Accessing a Structure Member

Compiler knows the *offset* of each member within a struct

- Compute as *(r+offset)*
  - Referring to absolute offset, so no pointer arithmetic

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} st, *r = &st;
```

```c
long get_i(struct rec *r)
{
    return r->i;
}
```

```
# r in %rdi, index in %rsi
movq 16(%rdi), %rax
ret
```
Exercise: Pointer to Structure Member

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} st, *r = &st;

long* addr_of_i(struct rec *r) {
    return &(r->i);
}

struct rec** addr_of_next(struct rec *r) {
    return &(r->next);
}
```
Generating Pointer to Array Element

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as: \( r + 4 \times \text{index} \)

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
} st, *r = &st;
```

```c
int* find_addr_of_array_elem (struct rec *r, long index) {
    return &r->a[index];
}
```

```asm
# r in %rdi, index in %rsi
leaq (%rdi, %rsi, 4), %rax
ret
```
Review: Memory Alignment in x86-64

- **Aligned** means that any primitive object of $K$ bytes must have an address that is a multiple of $K$

- Aligned addresses for data types:

<table>
<thead>
<tr>
<th>$K$</th>
<th>Type</th>
<th>Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
<td>No restrictions</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
<td>Lowest bit must be zero: $\ldots0_2$</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
<td>Lowest 2 bits zero: $\ldots00_2$</td>
</tr>
<tr>
<td>8</td>
<td>long, double, *</td>
<td>Lowest 3 bits zero: $\ldots000_2$</td>
</tr>
<tr>
<td>16</td>
<td>long double</td>
<td>Lowest 4 bits zero: $\ldots0000_2$</td>
</tr>
</tbody>
</table>

"multiple of" means no remainder when you divide by; since $K$ is a power of 2, dividing by $K$ is equivalent to $\gg \log_2(K)$. No remainder means no weight is "lost" during the shift $\rightarrow$ all zeros in lowest $\log_2(K)$ bits.
Alignment Principles

- Aligned Data
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
  - Required on some machines; advised on x86-64

- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of bytes (width is system dependent)
    - Inefficient to load or store value that spans quad word boundaries
    - Virtual memory trickier when value spans 2 pages (more on this later)
  - Though x86-64 hardware will work regardless of alignment of data
Structures & Alignment

- **Unaligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

```c
struct S1 {
    char c;
    int i[2];
    double v;
} st, *p = &st;
```
Satisfying Alignment with Structures (1)

- **Within** structure:
  - Must satisfy each element’s alignment requirement

- **Overall** structure placement
  - Each structure has alignment requirement $K_{max}$
    - $K_{max} =$ Largest alignment of any element
    - Counts array elements individually as elements

- Example:
  - $K_{max} =$ 8, due to `double` element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} st, *p = &st;
```
Satisfying Alignment with Structures (2)

- Can find offset of individual fields using `offsetof()`
  - Need to `#include <stddef.h>`
  - Example: `offsetof(struct S2, c)` returns 16

- For largest alignment requirement $K_{\text{max}}$, overall structure size must be multiple of $K_{\text{max}} = 8$
  - Compiler will add padding at end of structure to meet overall structure alignment requirement

```c
struct S2 {
    double v;
    int i[2];
    char c;
} st, *p = &st;
```
Arrays of Structures

- Overall structure length multiple of $K_{max}$
- Satisfy alignment requirement for every element in array

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Alignment of Structs

- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each *field* must be aligned *within* the struct
    (may insert padding)
    - `offsetof` can be used to get actual field offset
  - Overall struct must be *aligned* according to largest field
  - Total struct *size* must be multiple of its alignment
    (may insert padding)
    - `sizeof` should be used to get true size of structs
How the Programmer Can Save Space

- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

```c
struct S4 {
    char c;
    int i;
    char d;
} st;
```

```c
struct S5 {
    int i;
    char c;
    char d;
} st;
```

- 12 bytes vs. 8 bytes

**same data, but more efficient!**
Polling Question

- Minimize the size of the struct by re-ordering the vars

```
struct old {
    int i;
    short s[3];
    char *c;
    float f;
};
```

```
struct new {
    int i;
    float f;  // could also switch these (internal vs. external frag)
    char *c;
    short s[3];
};
```

- What are the old and new sizes of the struct?

  sizeof(struct old) = ______  sizeof(struct new) = ______

A. 16 bytes
B. 22 bytes
C. 28 bytes
D. 32 bytes
E. We’re lost...

Vote on `sizeof(struct old)`: [http://pollev.com/rea](http://pollev.com/rea)
Summary

- **Arrays in C**
  - Aligned to satisfy every element’s alignment requirement

- **Structures**
  - Allocate bytes for fields in order declared by programmer
  - Pad in middle to satisfy individual element alignment requirements
  - Pad at end to satisfy overall struct alignment requirement