Structs & Alignment
CSE 351 Autumn 2022

Instructor: Justin Hsia
Teaching Assistants:
Angela Xu
Assaf Vayner
David Dai
James Froelich
Paul Stevans
Arjun Narendra
Carrie Hu
Dominick Ta
Jenny Peng
Renee Ruan
Armin Magness
Clare Edmonds
Effie Zheng
Kristina Lansang
Vincent Xiao

http://xkcd.com/804/
Relevant Course Information

- Lab 2 due tonight
- Lab 3 released next Monday (10/31)
  - A shorter lab, due Friday, 11/11
- hw13 due next Wednesday (11/2)

- **Take-home Midterm (11/3 – 11/5)**
  - Instructions will be posted on Ed Discussion
  - **Gilligan’s Island Rule**: discuss high-level concepts and give hints, but not solving the problems together
  - We will be available on Ed Discussion (private posts only) and office hours to answer clarifying questions
Reading Review

❖ Terminology:
   ▪ Structs: tags and fields, . and → operators
   ▪ Typedef
   ▪ Alignment, internal fragmentation, external fragmentation

❖ Questions from the Reading?
Review Questions

```c
struct ll_node {
    long data;
    struct ll_node* next;
} n1, n2;
```

❖ How much space does (in bytes) does an instance of `struct ll_node` take?

❖ Which of the following statements are syntactically valid?

- `n1.next = &n2;`
- `n2->data = 351;`
- `n1.next->data = 333;`
- `(&n2)->next->next.data = 451;`
Data Structures in C

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

- **Structs**
  - **Alignment**

- **Unions**
Structs in C (Review)

- 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 = "drivers license";
song1.lengthInSeconds = 242;
song1.yearReleased = 2021;

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

❖ 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 (Review)

❖ A way to create an alias for another data type:
  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

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

❖ Why is the placement of struct definition important?
  ▪ Declaring a variable creates space for it somewhere
  ▪ Without definition, program doesn’t know how much space

```
struct data {
  int ar[4];
  long d;
};
```
Size = 24 bytes

```
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 (Review)

❖ 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 (shorter): `r->i = val;`

❖ In assembly: register holds address of the first byte
  - Access members with offsets
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 $(\ast x).f$

- 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 ≤ 8 bytes (but can point to lots of data)
Structure Representation (Review)

```c
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 (Review)

- 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

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

❖ Compiler knows the offset of each member
   ▪ No pointer arithmetic; compute as *(r+offset)

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

.int get_a3(struct rec* r) {
   return r->a[3];
}

# r in %rdi
movq 16(%rdi), %rax
ret

# r in %rdi
movl 12(%rdi), %eax
ret
### Pointer to Structure Member

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

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

```c
struct rec** addr_of_next(struct rec* r) {
    return &(r->next);
}
```

```c
# r in %rdi
leaq 16(%rdi), %rax
ret
```

```c
# r in %rdi
leaq 24(%rdi), %rax
ret
```
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
```
Struct Pointers

- Pointers store addresses, which all “look” the same
  - Lab 0 Example: struct instance Scores could be treated as array of ints of size 4 via pointer casting
  - A struct pointer doesn’t have to point to a declared instance of that struct type

- Different struct fields may or may not be meaningful, depending on what the pointer points to
  - This will be important for Lab 5!

```c
long get_a3(struct rec* r) {
    return r->a[3];
}
```

```
  movl 12(%rdi), %rax
  ret
```

Memory:
```
  r
  r+12
```

"r->a[3]"
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
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: ...0₂</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
<td>Lowest 2 bits zero: ...00₂</td>
</tr>
<tr>
<td>8</td>
<td>long, double, *</td>
<td>Lowest 3 bits zero: ...000₂</td>
</tr>
<tr>
<td>16</td>
<td>long double</td>
<td>Lowest 4 bits zero: ...0000₂</td>
</tr>
</tbody>
</table>
Structures & Alignment (Review)

❖ Unaligned Data

```
struct S1 {
    char c;
    int i[2];
    double v;
} st, *p = &st;
```

❖ Aligned Data

- Primitive data type requires \( K \) bytes
- Address must be multiple of \( K \)

```
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 $K_{\text{max}}$
    - $K_{\text{max}}$ = Largest alignment of any element
    - Counts array elements individually as elements

- **Example:**
  - $K_{\text{max}} = 8$, 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 $K_{\text{max}}$
  - 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

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

- 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;
```

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>c</th>
<th>d</th>
<th>2 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Practice Question

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

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

What is the new size of the struct?

```c
struct new {
    int i;
    ______  ______;
    ______  ______;
    ______  ______;
};
```

- Minimize the size of the struct by re-ordering the vars
- What is the new size of the struct?
- `sizeof(struct old) = 32 B`
- `sizeof(struct new) = _____`

A. 22 bytes
B. 24 bytes
C. 28 bytes
D. 32 bytes
E. We’re lost...
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