Structs & Alignment
CSE 351 Summer 2020

Instructor:
Porter Jones

Teaching Assistants:
Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

http://xkcd.com/163/
Administrivia

- Questions doc: [https://tinyurl.com/CSE351-7-24](https://tinyurl.com/CSE351-7-24)

- hw13 due Monday (7/27) – 10:30am

- hw14 due Wednesday (7/29) – 10:30am
  - This one is especially long, please start early

- Lab 3 due next Friday (7/31) – 11:59pm
  - You get to write some buffer overflow exploits!
Roadmap

C:

```c
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:

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

Machine code:

```
0111010000011000
1000110100000100
1000100111000010
1100000111111010
0100001111101000
```

Computer system:

Memory & data
Integers & floats
Java vs. C

Memory & caches
x86 assembly
Executables
Proc. & stacks
Java vs. C

Arrays & structs
Virtual memory
Executables
Memory allocation
Java vs. C
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 = "Respect";  
song1.lengthInSeconds = 148;  
song1.yearReleased = 1967;

struct song song2;  
song2.title = "Purple Haze";  
song2.lengthInSeconds = 171;  
song2.yearReleased = 1970;
```
Struct Definitions

- **Structure definition:**
  - Does NOT declare a variable
  - Variable type is "`struct name`"

- **Variable declarations like any other data type:**

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

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

Size = 16 + 8 = 24 bytes

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

Size = 16 + 8 + 8 = 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;`  `*r->i = val;`
  - Use -> operator for short:  `r->i = val;`  `better style`

- In assembly: register holds address of the first byte
  - Access members with offsets
Java connection

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

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

- 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;
};
struct rec st;
struct rec *r = &st;
```
Accessing a Structure Member

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

long get_i(struct rec *r) {
    return r->i; *(r + 16)
}
```

- Compiler knows the offset of each member within a struct
  - Compute as *(r+offset)
    - Referring to absolute offset, so no pointer arithmetic

```asm
movq 16(%rdi), %rax
ret
```

# r in %rdi, index in %rsi

---

1. Accessing a Structure Member
2. Compiler knows the offset of each member within a struct
   - Compute as *(r+offset)
     - Referring to absolute offset, so no pointer arithmetic
Exercise: Pointer to Structure Member

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

```c
long* addr_of_i(struct rec *r)  
{  
    return &(r->i); // r + 16
}
```

```c
struct rec** addr_of_next(struct rec *r)  
{  
    return &(r->next); // r + 24
}
```
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;
};
struct rec st;  
struct rec *r = &st;
```

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

```c
# 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**
  
  - Position of variables:
    - `c` at $p+0$
    - `i[0]` at $p+4$
    - `i[1]` at $p+8$
    - `v` at $p+16$

  - Address must be multiple of 4
  - Internal fragmentation

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

- Fragmentation occurs when there are unused portions of a struct

- Internal Fragmentation
  - Unused portion(s) occur *between* fields

- External Fragmentation
  - Unused portion at the end of the struct
Satisfying Alignment with Structures (1)

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

- **Overall structure placement**
  - Each structure has alignment requirement $K_{\text{max}}$
    - $K_{\text{max}} = \text{Largest alignment of any element}$
    - 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;
};
struct S1 st;
struct S1 *p = &st;
```
Satisfying Alignment with Structures (2)

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

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

```
struct S2 {  
double v;  
int i[2];  
char c;  
};
```

```
struct S2 st;  
struct S2 *p = &st;
```
Arrays of Structures

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

```c
struct S2 {
    double v;
    int i[2];
    char c;
};
struct S2 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

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

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

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

```
<table>
<thead>
<tr>
<th>8 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
</tbody>
</table>
```

Multiple of 4 □
Polling Question [Structs]

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

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

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

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

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

- Size comparison:
  ```c
  sizeof(struct old) = ___  sizeof(struct new) = ___
  ```

  - A. 16 bytes
  - B. 22 bytes
  - C. 28 bytes
  - D. 32 bytes
  - E. We’re lost...
Aside: More Struct Definitions

Can combine struct and instance definitions:

```c
struct name {  
    /* fields */  
};

struct name st;
struct name *p = &st;
```

Define a struct type (`struct name`), an instance of that type (`st`), and a pointer to that type (`p`)

This syntax is difficult to read

- Porter doesn’t like it in most situations because it conflates a type definition with an instance definition. But that’s just his opinion...
- We are showing it because you may see it in code in the future (and on the homework 😊)
Aside: Typedef in C

- 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 uni;

- Joint struct definition and typedef
  - Don’t need to give struct a name in this case
  - *typedef* alone doesn’t create an instance of the struct!

```
struct nm {
    /* fields */
};
typedef struct nm name;
name n1;
```

```
typedef struct {
    /* fields */
} name;
name n1;
```
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
Data Structures in Assembly

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structs
  - Alignment
- Unions

This is extra (non-testable) material
Unions

- Only allocates enough space for the largest element in union
- Can only use one member at a time

```c
struct S {
    char c;
    int i[2];
    double v;
} s, *sp = &s;

union U {
    char c;
    int i[2];
    double v;
} u, *up = &u;
```

This is extra (non-testable) material
Accessing Array Elements

- Compute start of array element as: $12 * \text{index}$
  - `sizeof(S3) = 12`, including alignment padding
- Element $j$ is at offset 8 within structure
- Assembler gives offset $a + 8$

```c
short get_j(int index) {
  return a[index].j;
}
```

```assembly
short get_j(int index) {
  return a[index].j;
}
```

```c
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

```assembly
# %rdi = index
leaq (%rdi,%rdi,2),%rax  # 3*index
movzwl a+8 (%rax,4),%eax
```
C:

```c
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
110000011111110100001111
```

**Computer system:**

**OS:**

- Windows 10
- OS X Yosemite

**Memory & data**
- Integers & floats

**x86 assembly**
- Procedures & stacks

**Executables**
- Arrays & structs

**Processes**
- Memory & caches

**Virtual memory**
- Memory allocation

**Java vs. C**
Assembly Programmer’s View

- **Programmer-visible state**
  - **PC**: the Program Counter (%rip in x86-64)
    - Address of next instruction
  - Named registers
    - Together in “register file”
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- **Memory**
  - Byte-addressable array
  - Code and user data
  - Includes *the Stack* (for supporting procedures)
x86-64 Instructions

1. Data movement
   - mov, movs, movz, ...
   - Operand types: Imm $
   - Reg %
   - Mem ()

2. Arithmetic
   - add, sub, shl, sar, lea, ...
   - Labels are addresses

3. Control flow
   - cmp, test, j*, set*, ...

4. Stack/procedures
   - push, pop, call, ret, ...
Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`)
  - Put resulting machine code in file `p`
Assembling

- Executable has **addresses** (no more labels)

```
00000000004004f6 <pcount_r>:
4004f6:  b8 00 00 00 00 00  mov $0x0,%eax
4004fb:  48 85 ff       test %rdi,%rdi
4004fe:  74 13  je 400513 <pcount_r+0x1d>
400500:  53  push %rbx
400501:  48 89 fb      mov %rdi,%rbx
400504:  48 d1 ef      shr %rdi
400507:  e8 ea ff ff ff callq 4004f6 <pcount_r>
40050c:  83 e3 01     and $0x1,%ebx
40050f:  48 01 d8     add %rbx,%rax
400512:  5b          pop %rbx
400513:  f3 c3      rep ret
```

- `gcc -g pcount.c -o pcount`
- `objdump -d pcount`
### A Picture of Memory (64-bit view)

<table>
<thead>
<tr>
<th>Instruction Address</th>
<th>Stored Bytes</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000004004f6 &lt;pcount_r&gt;:</td>
<td></td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>4004f6: b8 00 00 00 00</td>
<td></td>
<td>test %rdi,%rdi</td>
</tr>
<tr>
<td>4004fb: 48 85 ff</td>
<td></td>
<td>je 400513 &lt;pcount_r+0x1d&gt;</td>
</tr>
<tr>
<td>4004fe: 74 13</td>
<td></td>
<td>push %rbx</td>
</tr>
<tr>
<td>400500: 53</td>
<td></td>
<td>mov %rdi,%rbx</td>
</tr>
<tr>
<td>400501: 48 89 fb</td>
<td></td>
<td>shr %rdi</td>
</tr>
<tr>
<td>400504: 48 d1 ef</td>
<td></td>
<td>callq 4004f6 &lt;pcount_r&gt;</td>
</tr>
<tr>
<td>400507: e8 ea ff ff ff</td>
<td></td>
<td>and $0x1,%ebx</td>
</tr>
<tr>
<td>40050c: 83 e3 01</td>
<td></td>
<td>add %rbx,%rax</td>
</tr>
<tr>
<td>40050f: 48 01 d8</td>
<td></td>
<td>pop %rbx</td>
</tr>
<tr>
<td>400512: 5b</td>
<td></td>
<td>rep ret</td>
</tr>
<tr>
<td>400513: f3 c3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Memory Address Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x4004f0</td>
<td>b8 00</td>
</tr>
<tr>
<td>0x4004f8</td>
<td>00</td>
</tr>
<tr>
<td>0x400500</td>
<td>53 48</td>
</tr>
<tr>
<td>0x400508</td>
<td>ea ff</td>
</tr>
<tr>
<td>0x400510</td>
<td>01 d8</td>
</tr>
</tbody>
</table>

Unaligned, but more compact.
CSE351, Summer 2020

L14: Structs & Alignment

[Diagram showing memory layout and alignment issues]

- `c`: 3 bytes
- `i[0]`, `i[1]`, `i[2]`: aligned to multiple of 4
- `v`: aligned to multiple of 8
- `s`: aligned to multiple of 8
- `fp+0`, `fp+4`, `fp+16`, `fp+24`, `fp+26`, `fp+32`: aligned to multiple of 8

- Multiple of 4 fragmentation
- Multiple of 8 fragmentation
- Internal fragmentation
- External fragmentation