Structs and Alignment

CSE 351 Autumn 2017

Instructor:
Justin Hsia

Teaching Assistants:
Lucas Wotton Michael Zhang Parker DeWilde Ryan Wong
Sam Gehman Sam Wolfson Savanna Yee Vinny Palaniappan

Administrivia

- Lab 2 due tonight
- Homework 3 due next Friday (11/3)
- Lab 3 released next Wednesday (11/1)

- **Midterm** (10/30, 5:10-6:20pm, KNE 120)
  - Come early for ID check and time to settle in
  - Make a cheat sheet! – two-sided letter page, *handwritten*
  - Midterm details Piazza post: @163
- Extra office hours
  - Sun 10/29, 6:30-8:30pm, CSE 438
  - Mon 10/30, normal lecture times, CSE 438

Roadmap

C:

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

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:

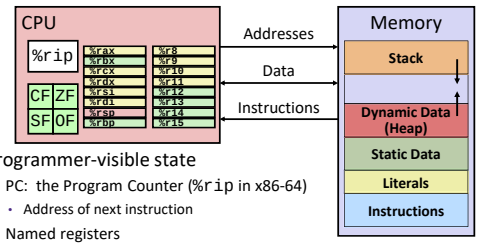
```
0111010000011000
100011010000010000000010
1000100111000010
1100000111110100011111
```

OS: Windows 10, OS X Yosemite

Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Assembly Programmer's View



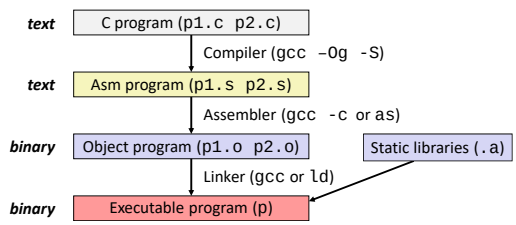
- Programmer-visible state
  - PC: the Program Counter (`%rip` in x86-64)
    - Address of next instruction
  - Named registers
    - Together in "register file"
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- Memory
  - Byte-addressable array
  - Code and user data
  - Includes *the Stack* (for supporting procedures)

x86-64 Instructions

- Data movement
  - `mov, movs, movz, ...`
- Arithmetic
  - `add, sub, shl, sar, lea, ...`
- Control flow
  - `cmp, test, j*, set*, ...`
- 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`) [New to recent versions of GCC]
  - Put resulting machine code in file `p`



Assembling

- Executable has **addresses**

assembler

```
00000000004004f6 <pcount_r>:
  4004f6:  b8 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)

```
00000000004004f6 <pcount_r>:
  4004f6:  b8 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
```

0|8	1|9	2|a	3|b	4|c	5|d	6|e	7|f	
								0x00
								0x08
								0x10
...								...
						b8	00	0x4004f0
00	00	00	48	85	ff	74	13	0x4004f8
53	48	89	fb	48	d1	ef	e8	0x400500
ea	ff	ff	ff	83	e3	01	48	0x400508
01	d8	5b	f3	c3				0x400510

Roadmap

C:

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

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
110000011111101000011111
```

OS: Windows 10, OS X Yosemite

Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Data Structures in Assembly

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

Peer Instruction Question

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

`int sea[4][5];`

9	8	1	9	5	9	8	1	0	5	9	8	1	0	3	9	8	1	1	5

76 96 116 136 156

A. `sea[4][-2]` is a *valid* array reference
B. `sea[1][1]` makes *two* memory accesses
C. `sea[2][1]` will *always* be a higher address than `sea[1][2]`
D. `sea[2]` is calculated using *only* `lea`
E. We're lost…

Structs in C

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

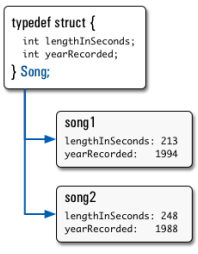
```
typedef struct {
  int lengthInSeconds;
  int yearRecorded;
} Song;

Song song1;

song1.lengthInSeconds =  213;
song1.yearRecorded    = 1994;

Song song2;

song2.lengthInSeconds =  248;
song2.yearRecorded    = 1988;
```



Struct Definitions

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

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

Easy to forget semicolon!

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

pointer

array

- 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;
```

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


Scope of Struct Definition

- Why is 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 = _____ bytes

Size = _____ bytes

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

- 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:

```
struct rec r1;
r1.i = val;
```

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

- Given a *pointer* to a struct:

```
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

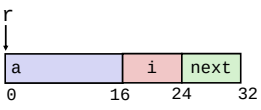
Java side-note

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

- 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

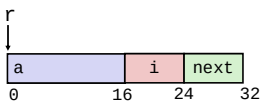
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



- Characteristics
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

Structure Representation

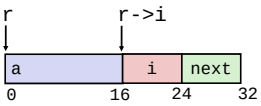
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



- 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

Accessing a Structure Member

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



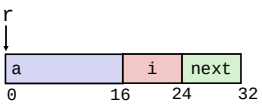
- Compiler knows the *offset* of each member within a struct
  - Compute as `*(r+offset)`
    - Referring to absolute offset, so no pointer arithmetic

```
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;
} *r;
```




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

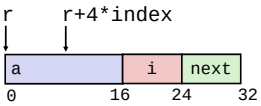
```
  # r in %rdi
  _____  ___,%rax
  ret
```

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

```
  # r in %rdi
  _____  ___,%rax
  ret
```


Generating Pointer to Array Element

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as: `r+4*index`

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

`&(r->a[index])`

```
  # r in %rdi, index in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```


Review: Memory Alignment in x86-64

- For good memory system performance, Intel recommends data be aligned
  - However the x86-64 hardware will work correctly regardless of alignment of data
- *Aligned* means that any primitive object of K bytes must have an address that is a multiple of K
- Aligned addresses for data types:

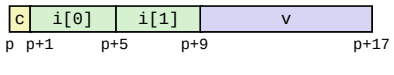
K	Type	Addresses
1	char	No restrictions
2	short	Lowest bit must be zero: $...0_2$
4	int, float	Lowest 2 bits zero: $...00_2$
8	long, double, *	Lowest 3 bits zero: $...000_2$
16	long double	Lowest 4 bits zero: $...0000_2$

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 4 or 8 bytes (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)

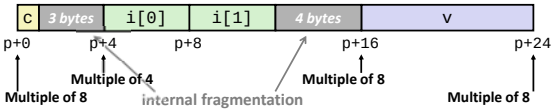
Structures & Alignment

- Unaligned Data




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

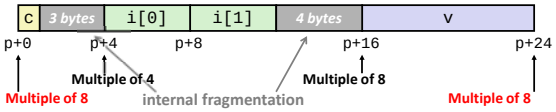
- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



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
  - **Address of structure & structure length must be multiples of $K_{\max}$**
- Example:
  - $K_{\max}$ = 8, due to `double` element

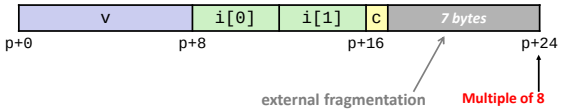
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



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 $K_{\max}$, 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;
} *p;
```



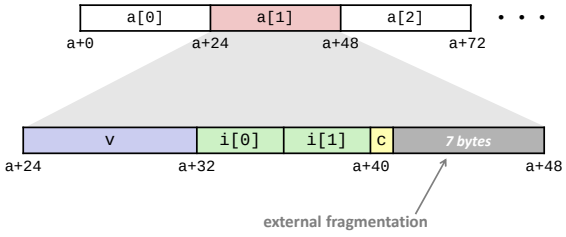
Alignment of Structs

- 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

Arrays of Structures

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

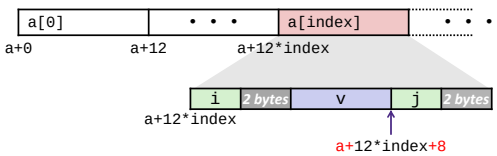
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

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




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

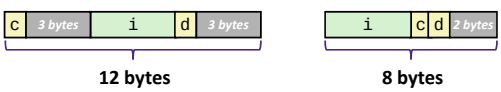
```
# %rdi = index
leaq (%rdi,%rdi,2),%rax  # 3*index
movzwl a+8(,%rax,4),%eax
```


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;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```



Peer Instruction Question

Vote on sizeof(**struct old**): http://PollEv.com/justinh

- 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;
  _____ _____;
  _____ _____;
  _____ _____;
};
```

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

  sizeof(struct old) = _____ sizeof(struct new) = _____

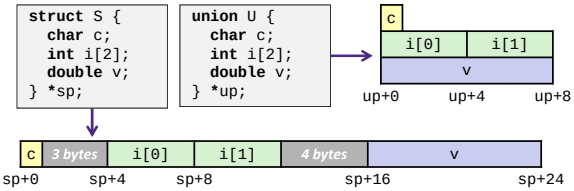
  A. 16 bytes
  B. 22 bytes
  C. 28 bytes
  D. 32 bytes
  E. We're lost…

Unions

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

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

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



Summary

- Arrays in C
  - Aligned to satisfy every element's alignment requirement
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Provide different views of the same memory location