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();

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

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000001111

OS:
Windows 8
Mac

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
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**
Review: Structs in Lab 0

```c
// Use typedef to create a type: FourInts
typedef struct {
    int a, b, c, d;
} FourInts;  // Name of type is "FourInts"

int main(int argc, char* argv[]) {

    FourInts f1;  // Allocates memory to hold a FourInts
                   // (16 bytes) on stack (local variable)
    f1.a = 0;     // Assign the first field in f1 to be zero

    FourInts* f2; // Declare f2 as a pointer to a FourInts

    // Allocate space for a FourInts on the heap,
    // f2 is a "pointer to"/"address of" this space.
    f2 = (FourInts*)malloc(sizeof(FourInts));
    f2->b = 17;  // Assign the second field to be 17
...
}
```
Syntax for structs without typedef

```c
struct rec {    // Declares the type "struct rec"
    int a[4];    // Total size = _______ bytes
    long i;
    struct rec *next;
};
struct rec r1; // Allocates memory to hold a struct rec
    // named r1, on stack or globally,
    // depending on where this code appears
struct rec *r;  // Allocates memory for a pointer
r = &r1;        // Initializes r to "point to" r1
```

Minor syntax note: Need that semicolon after a struct declaration (easy to forget)
Syntax for structs with typedef

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
struct rec r1; // Allocates memory to hold a struct rec
                // named r1, on stack or globally,
                // depending on where this code appears
struct rec *r; // Allocates memory for a pointer
r = &r1;       // Initializes r to “point to” r1
```

typedef struct rec {
    int a[4];
    long i;
    struct rec *next;
} Record; // typedef creates new name for ‘struct rec’
           // (that doesn’t need ‘struct’ in front of it)
Record r2; // Declare variable of type ‘Record’
           // (really a ‘struct rec’)
More Structs Syntax

```c
struct rec { // Declares the type “struct rec"
    int a[4];
    long i;
    struct rec *next;
};
struct rec r1; // Declares r1 as a struct rec
```

Equivalent to:

```c
struct rec { // Declares the type “struct rec"
    int a[4];
    long i;
    struct rec *next;
} r1; // Declares r1 as a struct rec
```
More Structs Syntax: Pointers

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
struct rec *r; // Declares r as pointer to a struct rec
```

Equivalent to:

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r; // Declares r as pointer to a struct rec
```
Accessing Structure Members

- Given an instance of the struct, we can use 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:
- Using * and . operators: `(*r).i = val;`
- Or, use `->` operator for short: `r->i = val;`

- The pointer is the address of the first byte of the structure
  - 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`, i.e., `(*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;
} *r;
```

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members 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 could yield a more compact representation
- 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;
} *r;
```
Accessing a Structure Member

```c
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)`

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

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

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

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

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

```c
define
    # r in %rdi
    _____ _____,%rax
ret
```
Exercise: Generating Pointer to Structure Member

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

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

```c
struct rec* address_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 Structure Member**

```
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: \( r + 4 \times \text{index} \)

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

# \( r \) in \( %rdi \), \( \text{index} \) 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:**
- Any primitive object of K bytes must have an address that is a multiple of K.

This means we could expect these types to have starting addresses that are the following multiples:

<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, pointers</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>
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**

  - 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 $K$

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

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

- **Overall structure placement**
  - Each `structure` has alignment requirement `K`
    - `K` = Largest alignment of any element
  - Initial address of structure & structure length must be multiples of `K`

- **Example:**
  - `K = 8`, due to `double` element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Satisfying Alignment Requirements: Another Example

- For largest alignment requirement $K$
- Overall structure size must be multiple of $K$
- Compiler will add padding at end of structure to meet overall structure alignment requirement

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

- Compiler:
  - Maintains declared `ordering` of fields in struct
  - Each `field` must be aligned `within` the struct *(may insert padding)*
    - `offsetof` can be used to find the actual offset of a field
  - 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
- 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
  - \(\text{sizeof}(S3) = 12\), 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
# %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

```c
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

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

**3 bytes**  
**3 bytes**  
**2 bytes**  
**12 bytes**  
**8 bytes**
Unions

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

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

struct S {
  char c;
  int i[2];
  double v;
} *sp;
```
What Are Unions Good For?

- Unions allow the same region of memory to be referenced as different types
  - Different “views” of the same memory location
  - Can be used to circumvent C’s type system (bad idea and technically not guaranteed to work)

- Better idea: use a struct inside a union to access some memory location either as a whole or by its parts

- But watch out for endianness at a small scale...

- Layout details are implementation/machine-specific...

```c
union int_or_bytes {
    int i;
    struct bytes {
        char b0, b1, b2, b3;
    }
}
```
typedef union
{
    unsigned char byte;
    struct {
        unsigned char reserved:4;
        unsigned char b3:1;
        unsigned char b2:1;
        unsigned char b1:1;
        unsigned char b0:1;
    } bits;
} hw_register;

hw_register reg;
reg.byte = 0x3F;       // 00111111_2
reg.bits.b2 = 0;       // 00111011_2
reg.bits.b3 = 0;       // 00110011_2
unsigned short a = reg.byte;
printf("0x%X\n", a);  // output: 0x33

(Note: the placement of these fields and other parts of this example are implementation-dependent)
Summary

- **Arrays in C**
  - Contiguous allocations of memory
  - No bounds checking
  - Can usually be treated like a pointer to first element
  - 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