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
110000011111101000001111

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

Autumn 2015
Data Structures in Assembly

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

- Structs
  - Alignment

- Unions
Array Allocation

- Basic Principle
  - T A[N];
  - Array of data type T and length N
  - *Contiguously* allocated region of N * sizeof(T) bytes

```plaintext
char mesg[12];
int val[5];
double a[3];
char* p[3];
```

(or char *p[3];)
Array Access

- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - Identifier A can be used as a pointer to array element 0: Type T*

```
int val[5];
```

```
9 8 1 9 5
```

```
x  x + 4  x + 8  x + 12 x + 16 x + 20
```

- **Reference Type Value**
  - val[4]
  - val
  - val+1
  - &val[2]
  - val[5]
  - *(val+1)
  - val + i
Array Access

- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - Identifier A can be used as a pointer to array element 0: Type T*

```
int val[5];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>x + 4</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>x + 8</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>?? (whatever is in memory at address x + 20)</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>8</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>x + 4*i</td>
</tr>
</tbody>
</table>

```
Array Example

typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

int uw[5] ...
Array Example

typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

- Declaration “zip_dig uw” equivalent to “int uw[5]”
- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Accessing Example

```c
typedef int zip_dig[5];

int get_digit(zip_dig z, int dig)
{
    return z[dig];
}
```

Assembly

```assembly
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference (%rdi,%rsi,4)
Referencing Examples

\[
\text{typedef int \( zip\_dig[5] \);} \\
\]

\[
\begin{array}{c}
\text{zip\_dig cmu;} \\
\begin{array}{cccccc}
1 & 5 & 2 & 1 & 3 \\
16 & 20 & 24 & 28 & 32 & 36
\end{array} \\
\end{array}
\]

\[
\begin{array}{c}
\text{zip\_dig uw;} \\
\begin{array}{cccccc}
9 & 8 & 1 & 9 & 5 \\
36 & 40 & 44 & 48 & 52 & 56
\end{array} \\
\end{array}
\]

\[
\begin{array}{c}
\text{zip\_dig ucb;} \\
\begin{array}{cccccc}
9 & 4 & 7 & 2 & 0 \\
56 & 60 & 64 & 68 & 72 & 76
\end{array} \\
\end{array}
\]

- **Reference**
- **Address**
- **Value**
- **Guaranteed?**

- \text{uw[3]}
- \text{uw[6]}
- \text{uw[-1]}
- \text{cmu[15]}
Referencing Examples

```c
typedef int zip_dig[5];
```

```
<table>
<thead>
<tr>
<th></th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>uw[3]</td>
<td>36 + 4* 3 = 48</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>uw[6]</td>
<td>36 + 4* 6 = 60</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>uw[-1]</td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>
```

- No bounds checking
- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Loop Example

### Function `zd2int` Example

```c
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

### Example Calculations

- **zi** = 10*0 + 9 = 9
- **zi** = 10*9 + 8 = 98
- **zi** = 10*98 + 1 = 981
- **zi** = 10*981 + 9 = 9819
- **zi** = 10*9819 + 5 = 98195

### Typedef

typedef int zip_dig[5];
Array Loop Example

- Original

- Transformed
  - Eliminate loop variable i, use pointer zend instead
  - Convert array code to pointer code
    - Pointer arithmetic on z
  - Express in do-while form (no test at entrance)
Array Loop Implementation

- **Registers**
  - %rdi  z
  - %rax  zi
  - %rcx  zend

- **Computations**
  - 10*zi + *z implemented as: *z + 2*(5*zi)
  - z++ increments by 4 (size of int=4)

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

```assembly
# %ecx = z
movl  $0,%eax
leaq 20(%rdi),%rcx
.L17:
    leal (%rax,%rax,4),%edx # zi + 4*zi = 5*zi
    movl (%rdi),%eax # eax = *z
    leal (%rax,%rdx,2),%eax # zi = *z + 2*(5*zi)
    addq $4,%rdi # z++
    cmpq %rcx,%rdi # z : zend
    jne .L17 # if != goto loop
```

gcc with –O1
Nested Array Example

```c
typedef int zip_dig[5];

  { 9, 8, 1, 9, 5 },
  { 9, 8, 1, 0, 5 },
  { 9, 8, 1, 0, 3 },
  { 9, 8, 1, 1, 5 }
};
```

Remember, \( T \ A[N] \) is an array with elements of type \( T \), with length \( N \)

What is the layout in memory?

```c
int sea[4][5];
```
Nested Array Example

```c
typedef int zip_dig[5];

zip_dig sea[4] =
    {{ 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 }};
```

- "Row-major" ordering of all elements
- Elements in the same row are contiguous
- Guaranteed (in C)

Remember, T A[N] is an array with elements of type T, with length N.
Two-Dimensional (Nested) Arrays

**Declaration**
- T \( A[R][C] \);
- 2D array of data type T
- R rows, C columns
- Type T element requires K bytes

**Array size?**
Two-Dimensional (Nested) Arrays

- **Declaration**
  - $T \ A[R][C]$;
  - 2D array of data type $T$
  - $R$ rows, $C$ columns
  - Type $T$ element requires $K$ bytes

- **Array size:**
  - $R \times C \times K$ bytes

- **Arrangement**
  - **Row-major** ordering

```plaintext
int A[R][C];
```

```
\begin{array}{cccc}
A[0][0] & \cdots & A[0][C-1] \\
A[R-1][0] & \cdots & A[R-1][C-1] \\
\end{array}
```

4*R*C Bytes
Nested Array **Row Access**

- **Row vectors**
  - Given: \( T \ A[R][C] \):
    - \( A[i] \) is an array of \( C \) elements, “row \( i \)”
    - Each element of type \( T \) requires \( K \) bytes
    - \( A \) is starting address of array
    - Starting address of row \( i = A + i \times (C \times K) \)

```c
int A[R][C];
```

![Diagram showing row access](image)
Nested Array Row Access Code

```c
int* get_sea_zip(int index)
{
    return sea[index];
}
```

```c
int sea[4][5] =
    {{ 9, 8, 1, 9, 5 },
     { 9, 8, 1, 0, 5 },
     { 9, 8, 1, 0, 3 },
     { 9, 8, 1, 1, 5 }};
```

- What data type is `sea[index]`?
- What is its starting address?
Nested Array **Row Access** Code

```c
int* get_sea_zip(int index)
{
    return sea[index];
}
```

- **What data type is** `sea[index]`?*
- **What is its starting address?**

```c
int sea[4][5] =
{
    { 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 }
};
```

```c
# %rdi = index
    leaq (%rdi,%rdi,4),%rax
    leaq sea(,%rax,4),%rax
```

**Translation?**
Nested Array **Row Access Code**

```c
int* get_sea_zip(int index) {
    return sea[index];
}
```

```c
int sea[4][5] = {
    { 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 }};
```

- **Row Vector**
  - `sea[index]` is array of 5 `int`
  - Starting address = `sea+20*index`

- **Assembly Code**
  - Computes and returns address
  - Compute as: `sea+4*(index+4*index)= sea+20*index`
Nested Array Element Access

```c
int A[R][C];
```

**Arrays & structs**
Nested Array Element Access

- **Array Elements**
  - $A[i][j]$ is element of type $T$, which requires $K$ bytes
  - Address of $A[i][j]$ is
    \[
    A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K
    \]

```c
int A[R][C];
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>C-1</td>
</tr>
</tbody>
</table>

```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>i</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>i</td>
<td>C-1</td>
</tr>
</tbody>
</table>

```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>R-1</td>
<td>R-1</td>
<td>C-1</td>
</tr>
</tbody>
</table>

```

$A + i \times C \times 4 + j \times 4$
Nested Array **Element Access Code**

```c
int get_sea_digit
  (int index, int dig)
{
  return sea[index][dig];
}

int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```assembler
leaq (%rdi,%rdi,4), %rax # 5*index
addl %rax, %rsi # 5*index+dig
movl sea(,%rsi,4), %eax # *(sea + 4*(5*index+dig))
```

### Array Elements
- `sea[index][dig]` is an `int`
- Address = `sea + 20*index + 4*dig`

### Assembly Code
- Computes address as: `sea + 4*dig + 4*(index+4*index)`
- `movl` performs memory reference
### Strange Referencing Examples

```c
typedef int zip_dig[5];
zip_dig sea[4];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sea[3][3]</code></td>
<td>9 8 1 9 5</td>
<td>9 8 1 0 5</td>
<td></td>
</tr>
<tr>
<td><code>sea[2][5]</code></td>
<td>9 8 1 0 5</td>
<td>9 8 1 0 3</td>
<td></td>
</tr>
<tr>
<td><code>sea[2][-1]</code></td>
<td>9 8 1 0 3</td>
<td>9 8 1 1 5</td>
<td></td>
</tr>
<tr>
<td><code>sea[4][-1]</code></td>
<td>9 8 1 1 5</td>
<td>9 8 1 1 5</td>
<td></td>
</tr>
<tr>
<td><code>sea[0][19]</code></td>
<td>9 8 1 1 5</td>
<td>9 8 1 1 5</td>
<td></td>
</tr>
<tr>
<td><code>sea[0][-1]</code></td>
<td>9 8 1 1 5</td>
<td>9 8 1 1 5</td>
<td></td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
Strange Referencing Examples

```c
typedef int zip_dig[5];
```

```c
zip_dig sea[4];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sea[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
Multi-Level Array Example

Multi-Level Array Declaration(s):

```c
int cmu[5] = { 1, 5, 2, 1, 3 };
int uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };

int* univ[3] = {uw, cmu, ucb};
```

Is a multi-level array the same thing as a 2D array? NO

2D Array Declaration:

```c
zip_dig univ2D[3] = {
    { 9, 8, 1, 9, 5 },
    { 1, 5, 2, 1, 3 },
    { 9, 4, 7, 2, 0 }\n};
```

One array declaration = one contiguous block of memory
**Multi-Level Array Example**

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes each
- Each pointer points to array of `int`

```java
int cmu[5] = { 1, 5, 2, 1, 3 };
int uw[5] = { 9, 8, 1, 9, 5 };  
int ucb[5] = { 9, 4, 7, 2, 0 };

int* univ[3] = {uw, cmu, ucb};
```

Note: this is how Java represents multi-dimensional arrays.
Element Access in Multi-Level Array

```c
int get_univ_digit(int index, int dig)
{
    return univ[index][dig];
}
```

```assembly
salq  $2, %rsi          # rsi = 4*digit
addq  univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl  (%rsi), %eax      # return *p
ret
```

### Computation

- Element access `Mem[Mem[univ+8*index]+4*dig]`
- Must do **two memory reads**
  - First get pointer to row array
  - Then access element within array
Array Element Accesses

Nested array

```c
int get_sea_digit (int index, int dig)
{
    return sea[index][dig];
}
```

Multi-level array

```c
int get_univ_digit (int index, int dig)
{
    return univ[index][dig];
}
```

Access looks similar, but it isn’t:

\[
\text{Mem}[\text{sea}+20*\text{index}+4*\text{dig}] \quad \text{vs.} \quad \text{Mem}[\text{Mem}[\text{univ}+8*\text{index}]+4*\text{dig}]
\]
Strange Referencing Examples

- C Code does not do any bounds checking
- Location of each lower-level array in memory is not guaranteed
Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>univ[2][3]</code></td>
<td><code>60+4*3 = 72</code></td>
<td><code>2</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td><code>16+4*5 = 36</code></td>
<td><code>9</code></td>
<td>No</td>
</tr>
<tr>
<td><code>univ[2][-2]</code></td>
<td><code>60+4*-2 = 52</code></td>
<td><code>5</code></td>
<td>No</td>
</tr>
<tr>
<td><code>univ[3][-1]</code></td>
<td><code>#%@!^??</code></td>
<td><code>??</code></td>
<td>No</td>
</tr>
<tr>
<td><code>univ[1][12]</code></td>
<td><code>16+4*12 = 64</code></td>
<td><code>4</code></td>
<td>No</td>
</tr>
</tbody>
</table>

- C Code does not do any bounds checking
- Location of each lower-level array in memory is not guaranteed
Summary: Arrays in C

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

**int a[4][5] => array of arrays**
- All levels in one contiguous block of memory

**int* b[4] => array of pointers to arrays**
- First level in one contiguous block of memory
- Each element in the first level points to another “sub” array
- Parts anywhere in memory
Data Structures in Assembly

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

- **Structs**
  - Alignment

- **Unions**
Review: Structs in Lab 0

// 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

    ...
}

Aside: Syntax for structs without typedef

```
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
```
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 Pointer Syntax

```c
struct rec {      // Declares the type “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 {      // Declares the type “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, (just like Java):
  ```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
**Structure Representation**

```c
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

- 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;
```
Generating **Pointer to Structure Member**

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

### 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];
}
```

```assembly
# r in %rdi, index in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```
Exercise: Generating Pointer to Structure Member

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

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

```
struct rec* address_of_next(struct rec *r)  
{  
    return &(r->next);  
}
```
Exercise: Generating Pointer to Structure Member

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

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

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

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

```
# r in %rdi  
leaq 24(%rdi), %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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
</tr>
<tr>
<td>8</td>
<td>long, double, pointers</td>
</tr>
</tbody>
</table>
Structures & Alignment

- **Unaligned Data**

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p+1</td>
<td>p+5</td>
<td>p+9</td>
</tr>
</tbody>
</table>

  - 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;
```
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 datum that spans quad word boundaries
    ▪ Virtual memory trickier when datum spans 2 pages (more on this later)

■ Compiler
  ▪ Maintains declared *ordering* of fields in struct
  ▪ Inserts padding in structure to ensure correct *alignment* of fields
  ▪ `sizeof()` should be used to get true size of structs
  ▪ `offsetof(struct, field)` can be used to find the actual offset of a field
Specific Cases of Alignment (x86-64)

- **1 byte: char, ...**
  - no restrictions on address
- **2 bytes: short, ...**
  - lowest 1 bit of address must be 0\textsubscript{2}
- **4 bytes: int, float, ...**
  - lowest 2 bits of address must be 00\textsubscript{2}
- **8 bytes: double, long, pointers, ...**
  - lowest 3 bits of address must be 000\textsubscript{2}
- **16 bytes: long double (GCC on Linux)**
  - lowest 4 bits of address must be 0000\textsubscript{2}
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

```
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;
```

Multiple of $K=8$
Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element in array

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

Create an array of ten S2 structs called “a”

```
a[0] a[1] a[2] ...
```

```
a+0 a+24 a+48 a+72
```

```
v i[0] i[1] c
```

```
a+24 a+32 a+40 a+48
```

7 bytes

external fragmentation
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
```

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

```c
short get_j(int index) {
    return a[index].j;
}
```
How the Programmer Can Save Space

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

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

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

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

12 bytes

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

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)
- 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;
    }
}
```
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