CS201- Lecture 10
IA32 Data Access

RAOUL RIVAS

PORTLAND STATE UNIVERSITY

Announcements

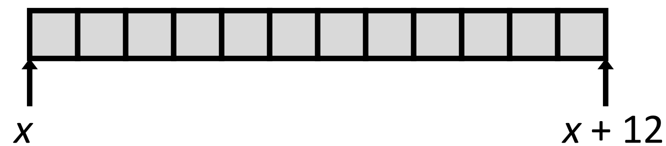
Array Allocation

- Basic Principle

  T **A[*L*]**;

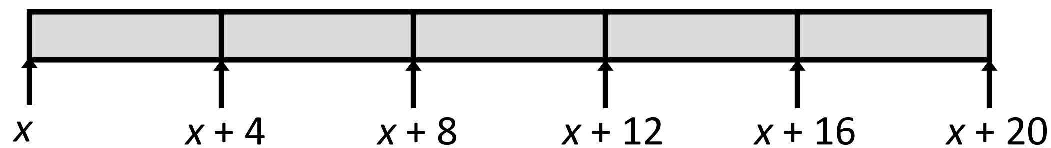
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof**(*T*) bytes in memory

`char string[12];`

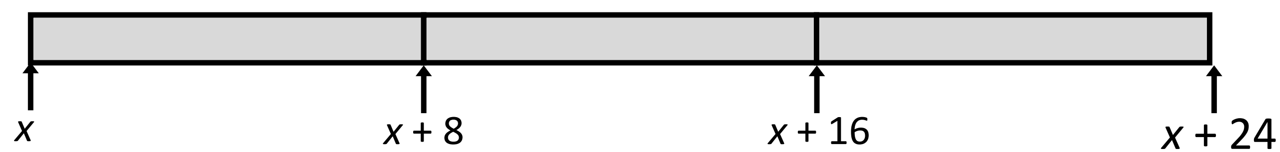


Remember x86 and x64 is byte addressable!

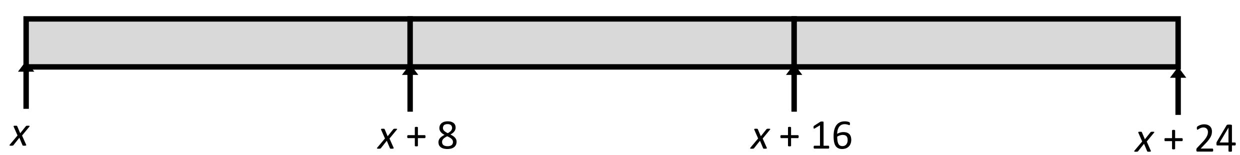
`int val[5];`



`double a[3];`

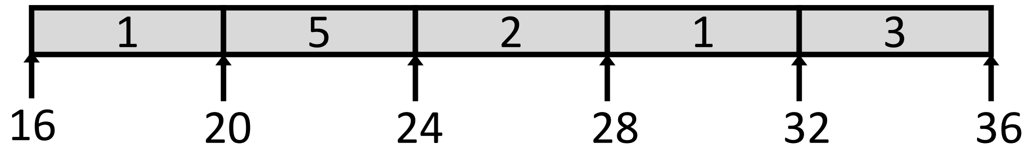


`char *p[3];`



Array Access Example

```
zip_dig cmu;
```




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

IA32

```
# %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)`

Array Loop Example

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

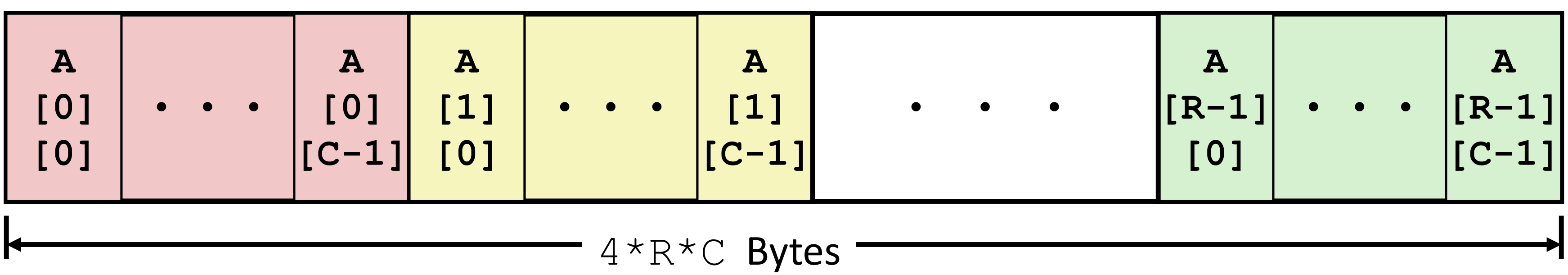
```
  # %rdi = z
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $4, %rax            #   i:4
  jbe     .L4                 #   if <=, goto loop
  rep; ret
```

Static Multidimensional 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* * *C* * *K* bytes
- Arrangement
  - Row-Major Ordering

$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

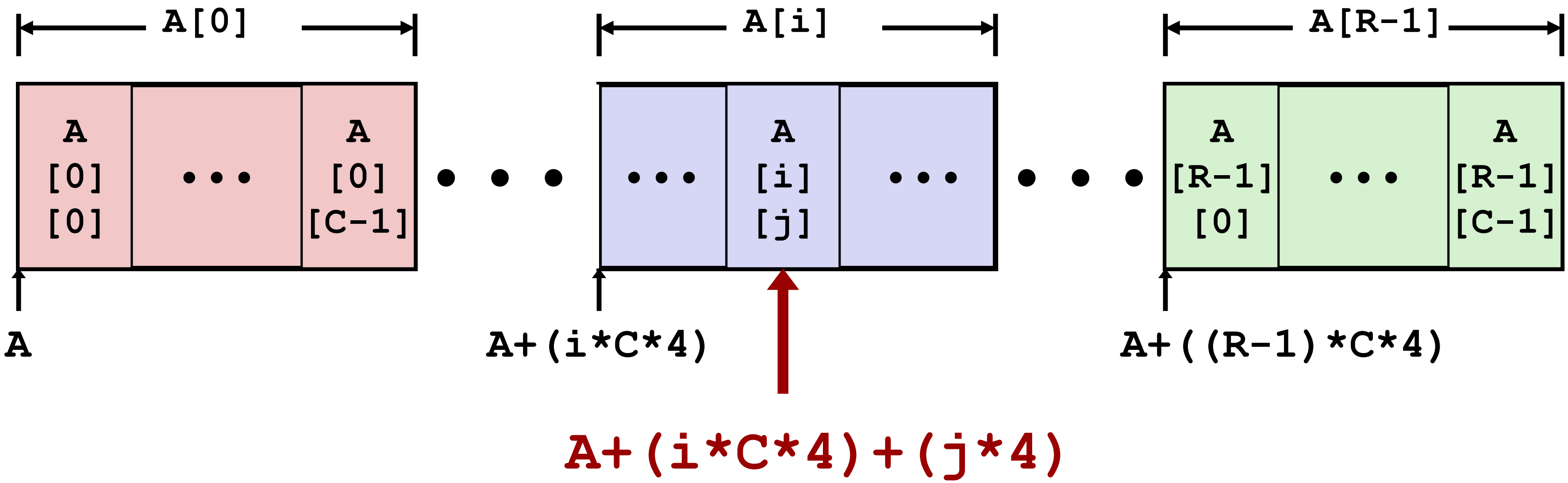
```
int A[R][C];
```



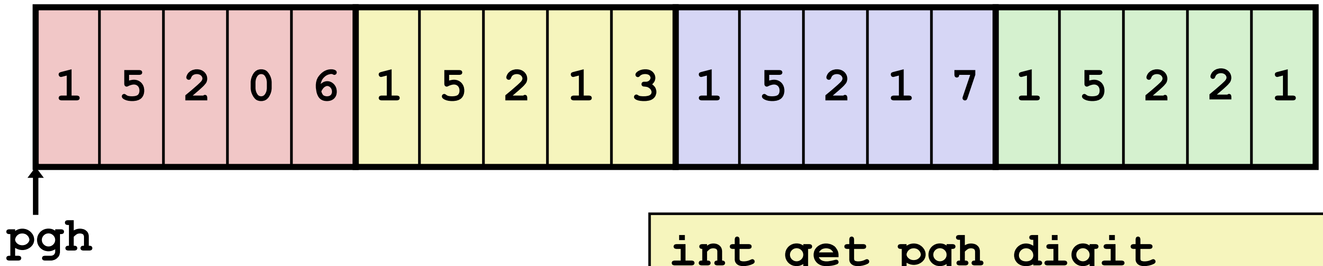
Element Access (Static Array)

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

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



Element Access (Static Array)




```
int get_pgh_digit
   (int index, int dig)
{
   return pgh[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

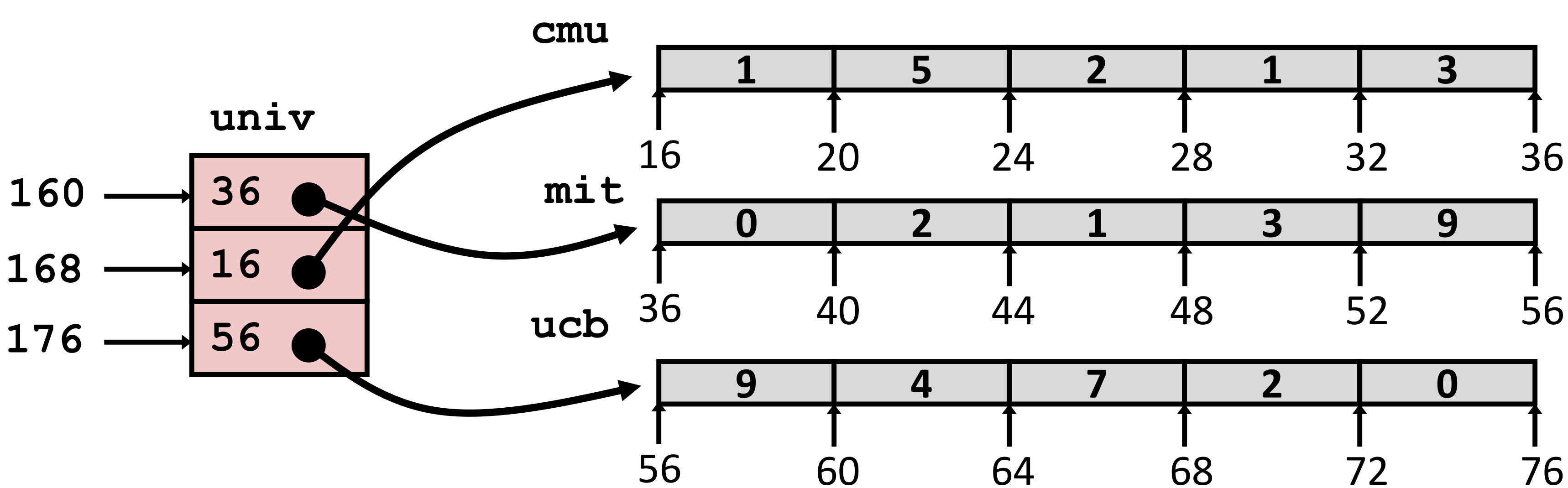
- **Array Elements**
  - `pgh[index][dig]` is `int`
  - Address: `pgh + 20*index + 4*dig`
    - = `pgh + 4*(5*index + dig)`

Array of Pointer (Dynamic Array)

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

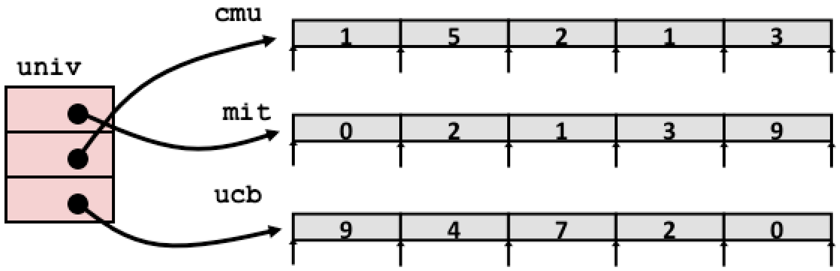
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

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



Element Access (Dynamic Array)

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```




```
  salq    $2, %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*digit]`**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

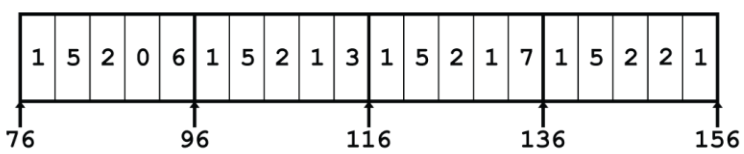
Dynamic vs Static Array

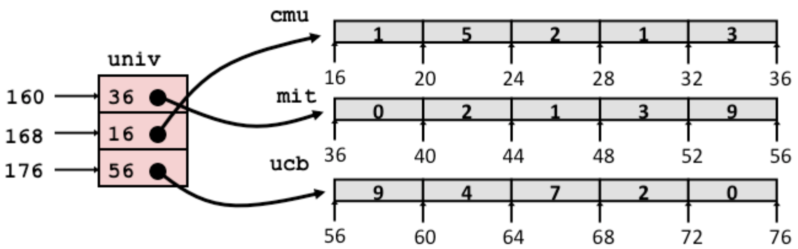
Static array

```
int get_pgh_digit
  (size_t index, size_t digit)
{
  return pgh[index][digit];
}
```

Dynamic array (Array of Pointers)

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```



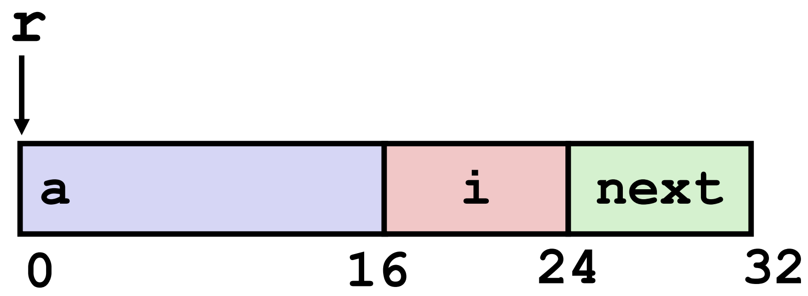


Accesses looks similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]`   `Mem[Mem[univ+8*index]+4*digit]`

Structures

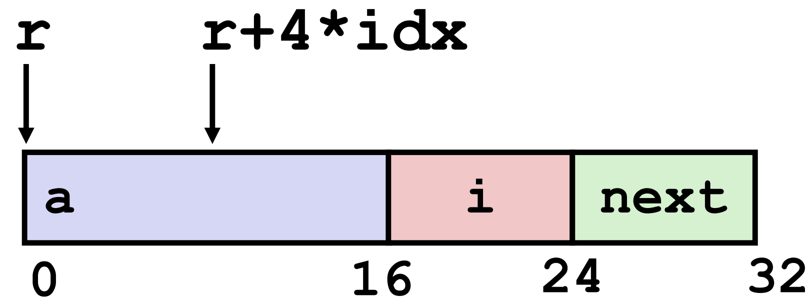
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- **Structure represented as block of memory**
  - **Big enough to hold all of the fields**
- **Fields ordered according to declaration**
  - **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**

Member Access

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



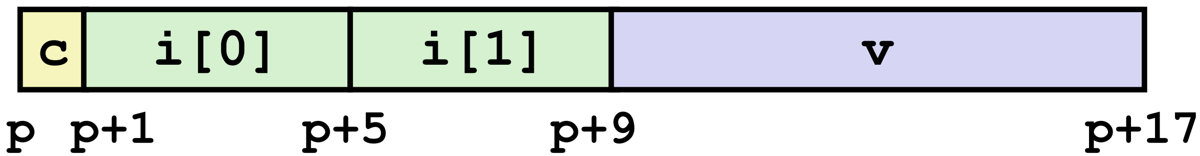
- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time
  - Compute as **`r + 4*idx`**

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

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

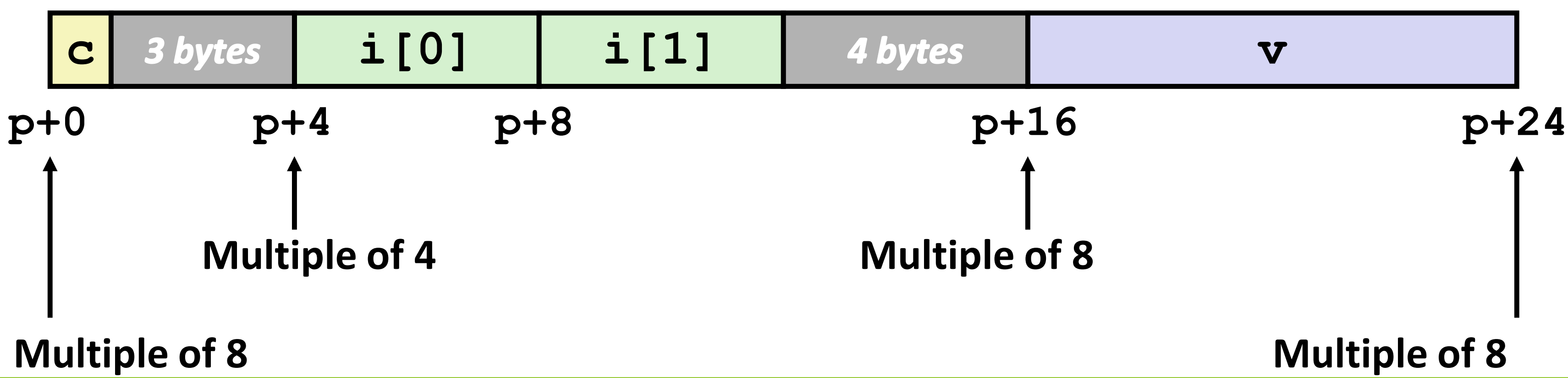
Data Alignment

- **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***



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
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

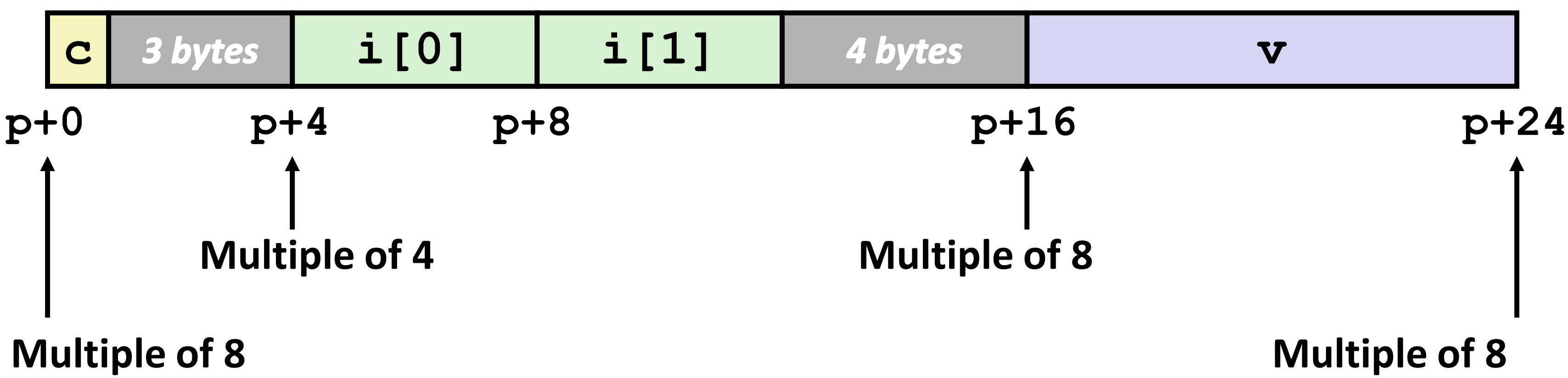
X64 Alignment

- 1 byte: **`char`**
  - no restrictions on address
- 2 bytes: **`short`**
  - lowest 1 bit of address must be 0_2
- 4 bytes: **`int`**, **`float`**
  - lowest 2 bits of address must be 00_2
- 8 bytes: **`double`**, **`long`**, **`char *`**
  - lowest 3 bits of address must be 000_2
- 16 bytes: **`long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Structure Alignment

- **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 & structure length must be multiples of **K**
- **Example:**
  - K = 8, due to **`double`** element

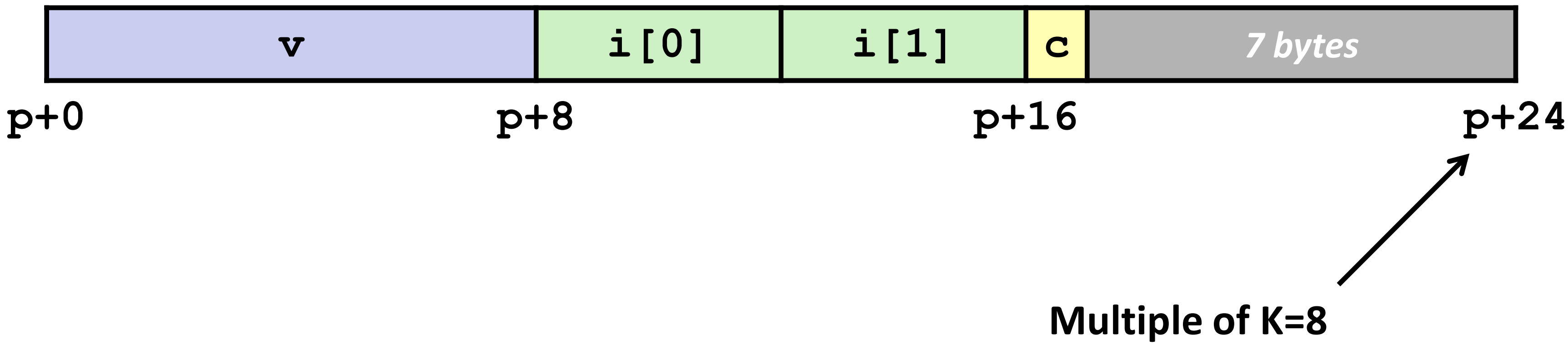
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Structure Alignment

- **For largest alignment requirement K**
- **Overall structure must be multiple of K**

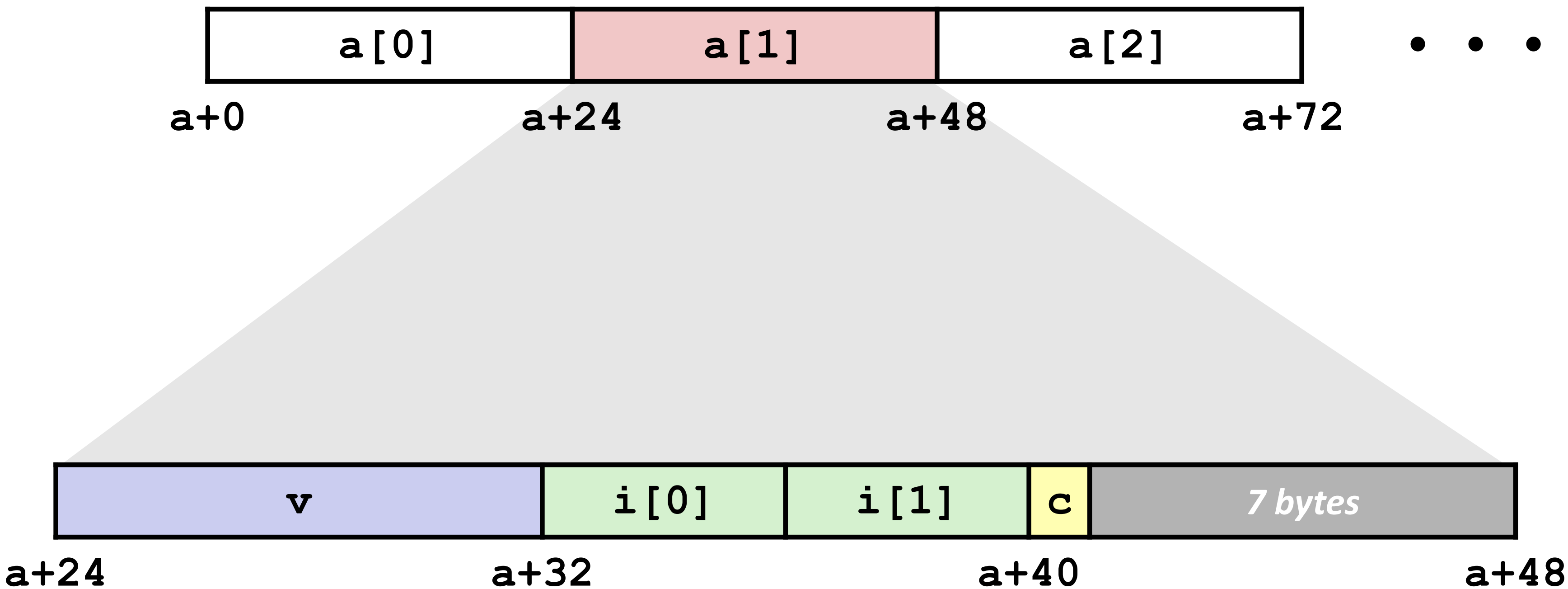
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

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

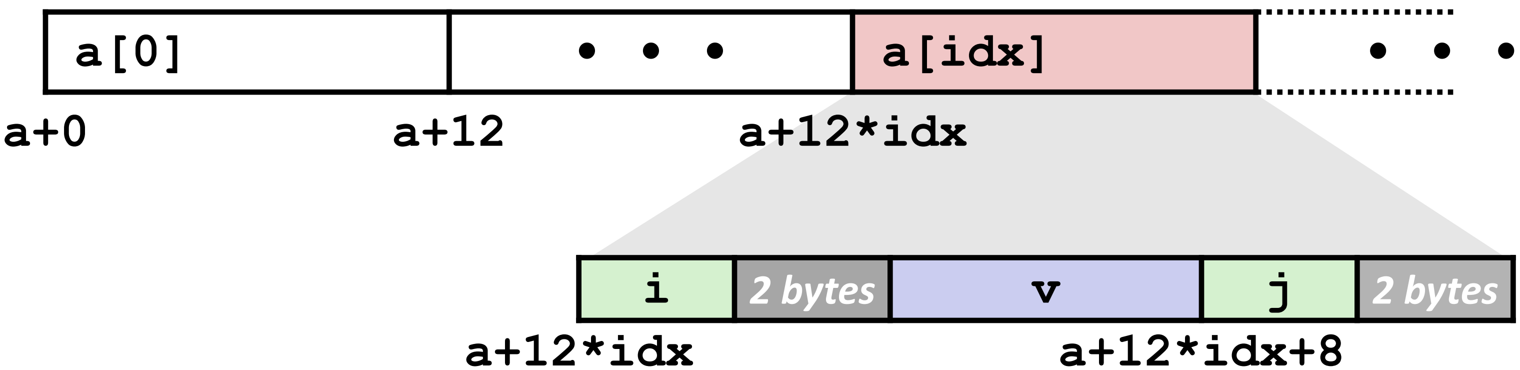
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Arrays of Structures

- **Compute array offset 12*idx**
  - `sizeof(S3)`, including alignment spacers
- **Element `j` is at offset 8 within structure**
- **Assembler gives offset `a+8`**
  - Resolved during linking

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




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

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

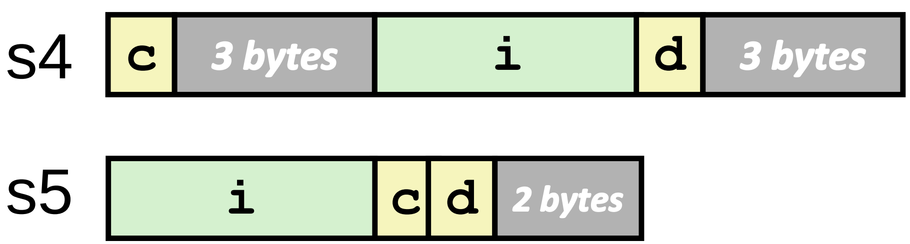
Structures Alignment Optimization

- **Put large data types first**

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

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

- **Result**



Data Serialization

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

Data Exchange

Internet
Flash Drives

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

Data Corruption

c	*6 bytes*	i	d

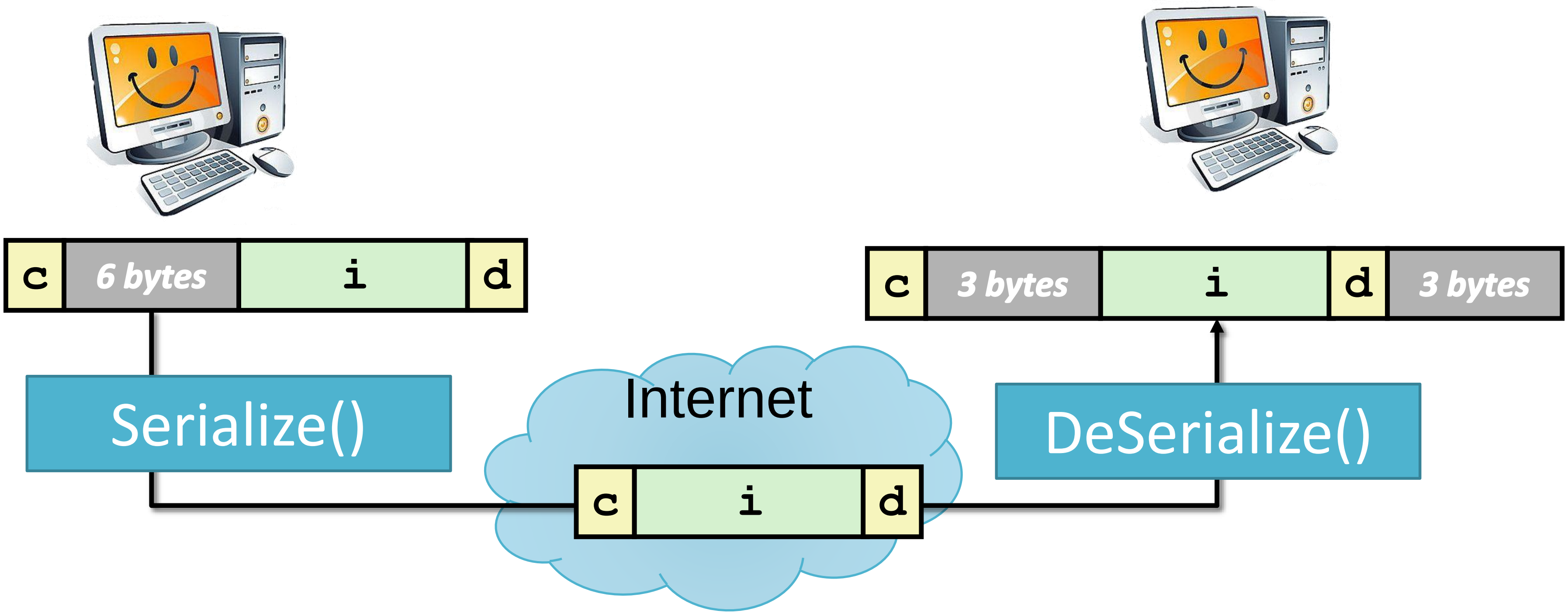
c	*3 bytes*	i	d	*3 bytes*

Never send a raw structure across the Network

Data Serialization

- If the language includes serialization routines use them
  - Java and C#
- If not copy structure member by member to a contingUous buffer before send
- Endianness – Use common agreement of formats
    - hton macros in C
  - Data sizes – Use independent data types
    - StdInt.h – uint16_t, uint32_t, etc
- Some libraries allow for data serialization trivially
  - Google Protocol Buffers Library

Data Serialization



Serialize.c

```
char b[sizeof(char)*2+sizeof(int)];
int off=0;
memcpy(b, &p->c, sizeof(char));
off = sizeof(char);
memcpy(b + off, &p->i, sizeof(int));
off += sizeof(int);
memcpy(b + off, &p->d, sizeof(char));
send(b, sizeof(b));
```

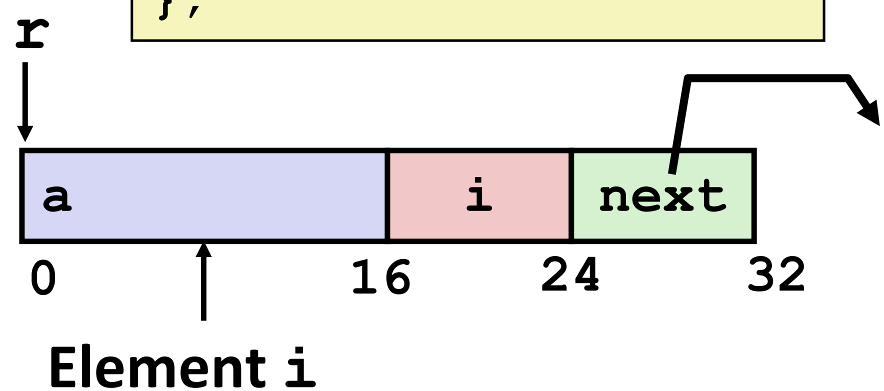
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

Linked Lists

- **C Code**

```
void set_val
   (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

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



Register	Value
%rdi	r
%rsi	val

```
.L11:                               # loop:
  movslq  16(%rdi), %rax            #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)       #   M[r+4*i] = val
  movq    24(%rdi), %rdi            #   r = M[r+24]
  testq   %rdi, %rdi                #   Test r
  jne     .L11                      #   if !=0 goto loop
```

Unions

- **Allocate according to largest element**
- **Can only use one field at a time**

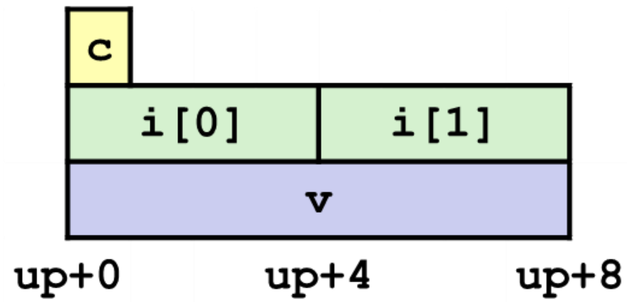
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

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

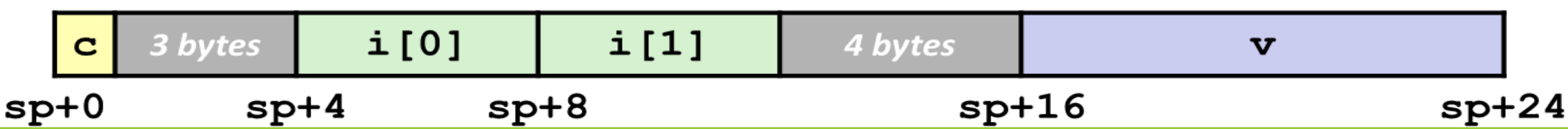
Any real life example?

Java Number class implementation

Union



Structure



Summary

- Static and Dynamic array access in C use the same semantics (operator []) but address computation is different
- Members of Structures must be properly aligned to avoid performance penalty
- Structures must be serialized before saving them to files or before sending them across the network
- Unions allow a single variable to represent multiple types