



Assembly IV: Complex Data Types

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
<http://csl.skku.edu>

Basic Data Types

- **Integer**

- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long (x86-64)

- **Floating point**

- Stored & operated on in floating point registers

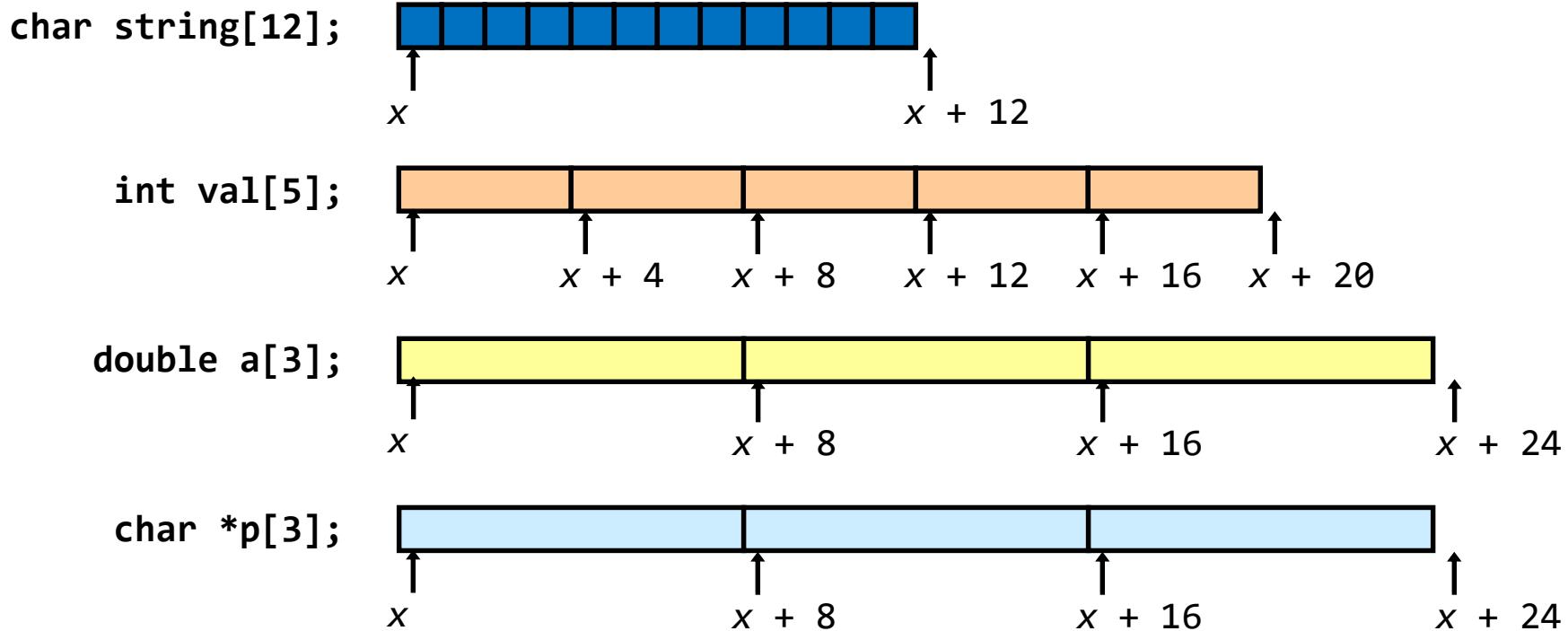
Intel	GAS	Bytes	C
single	s	4	float
double	l	8	double

Complex Data Types

- **Complex data types in C**
 - Pointers
 - Arrays
 - Structures
 - Unions
 - ...
- **Can be combined**
 - Pointer to pointer, pointer to array, ...
 - Array of array, array of structure, array of pointer, ...
 - Structure in structure, pointer in structure, array in structure, ...

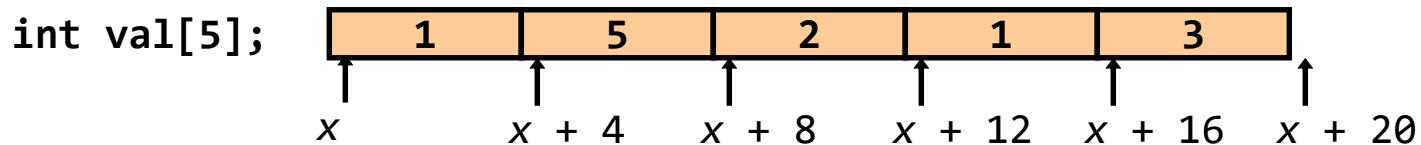
Array Allocation

- Basic principle: $T \ A[L];$
 - Array of data type T and length L
 - Contiguously allocated region of $L * \text{sizeof}(T)$ bytes



Array Access

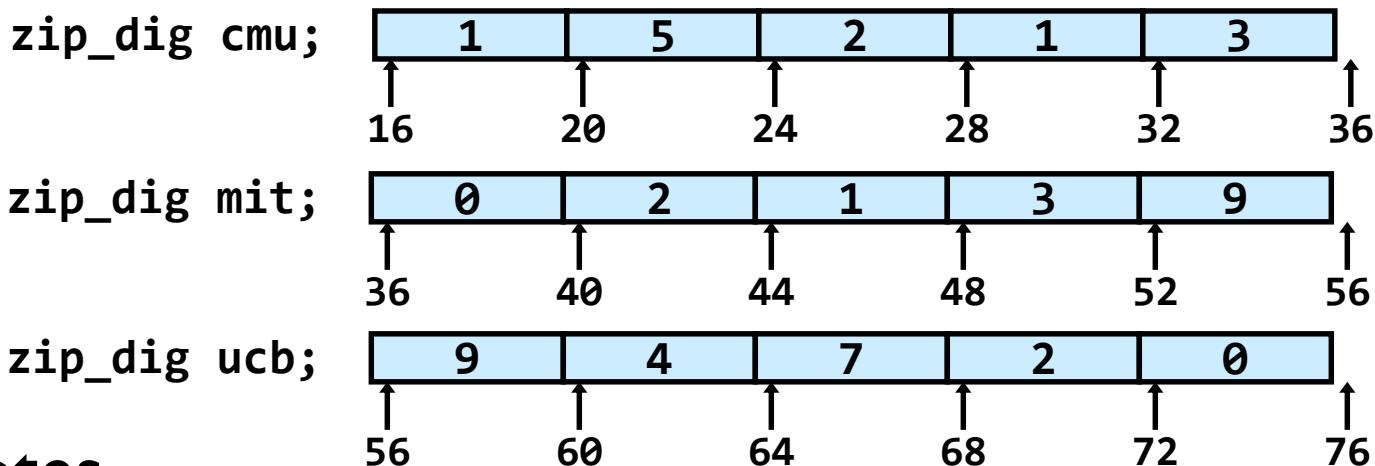
- Basic principle: $T \ A[L];$
 - Array of data type T and length L
 - Identifier A can be used as a pointer to array element 0



Reference	Type	Value
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	x
<code>val + 1</code>	<code>int *</code>	$x + 4$
<code>&val[2]</code>	<code>int *</code>	$x + 8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	$x + 4 * i$

Array Example

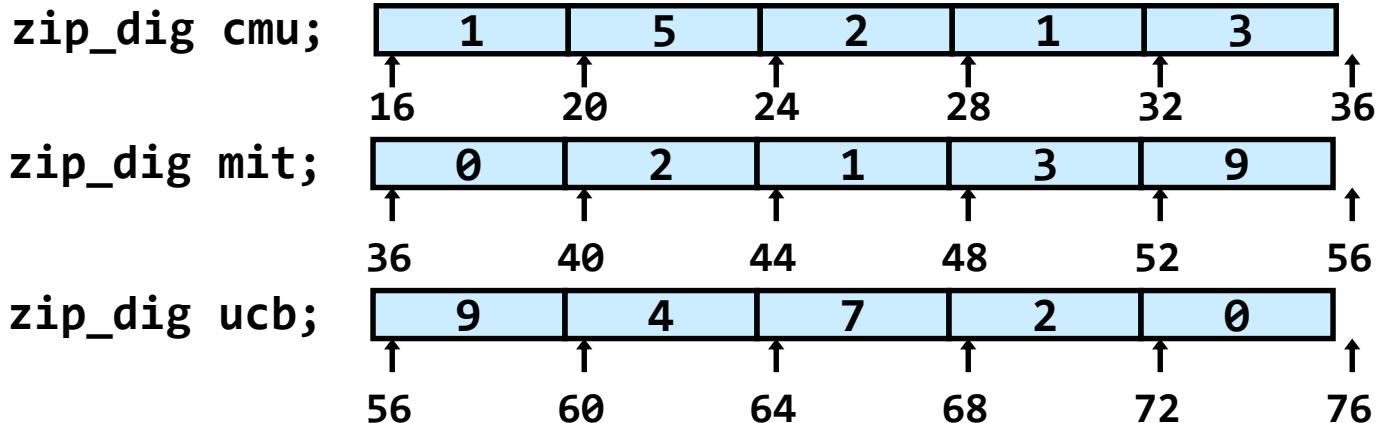
```
typedef int zip_dig[5];  
  
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig mit = { 0, 2, 1, 3, 9 };  
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- **Notes**

- Example arrays were allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

Array Access Example



- Code does not do any bounds checking!
 - Out of range behavior implementation-dependent
 - No guaranteed relative allocation of different arrays

Reference	Address	Value	Guaranteed?
<code>mit[3]</code>	$36 + 4 * 3 = 48$	3	Yes
<code>mit[5]</code>	$36 + 4 * 5 = 56$	9	No
<code>mit[-1]</code>	$36 * 4 * (-1) = 32$	3	No
<code>cmu[15]</code>	$16 + 4 * 15 = 76$??	No

Array Access Code

- Computation

- Register **%rdi** contains starting address of array
- Register **%esi** contains array index
- Desired digit at
%rdi + 4 * %esi

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

Memory Reference Code

```
# %rdi = z
# %esi = digit

movslq  %esi, %rsi
movl    (%rdi,%rsi,4),%eax
```

Array Loop Example

- Registers

- %rdi z
- %rax i

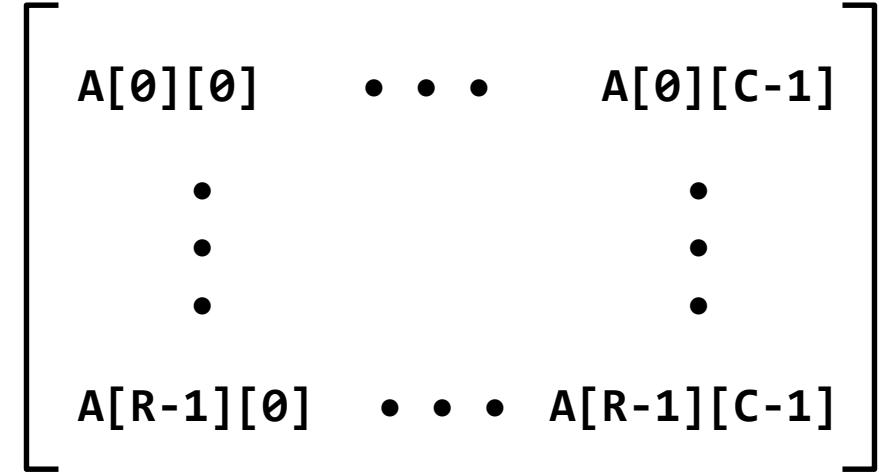
```
void zincr(zip_dig z) {  
    int i;  
    for (i = 0; i < 5; i++)  
        z[i]++;  
}
```

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

Nested Array

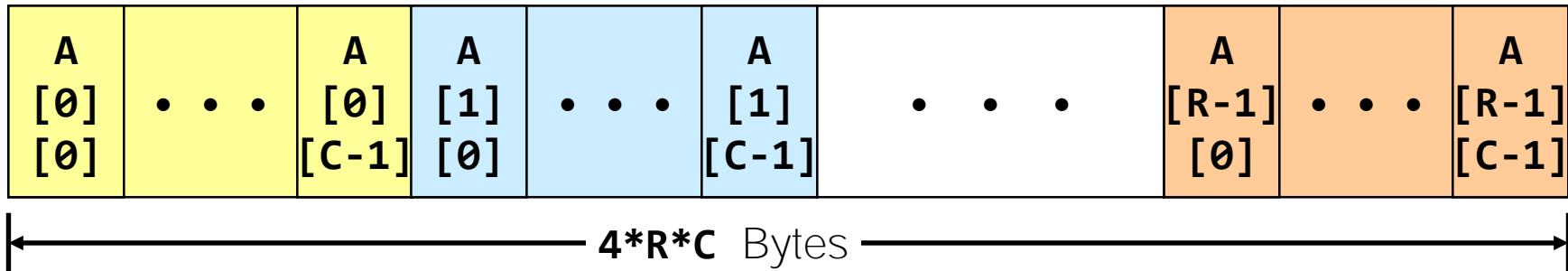
- Declaration: $T \ A[R][C];$

- 2D array of data type T
- R rows, C columns
- Array size =
$$R * C * \text{sizeof}(T)$$



- Arrangement

- Row-major ordering



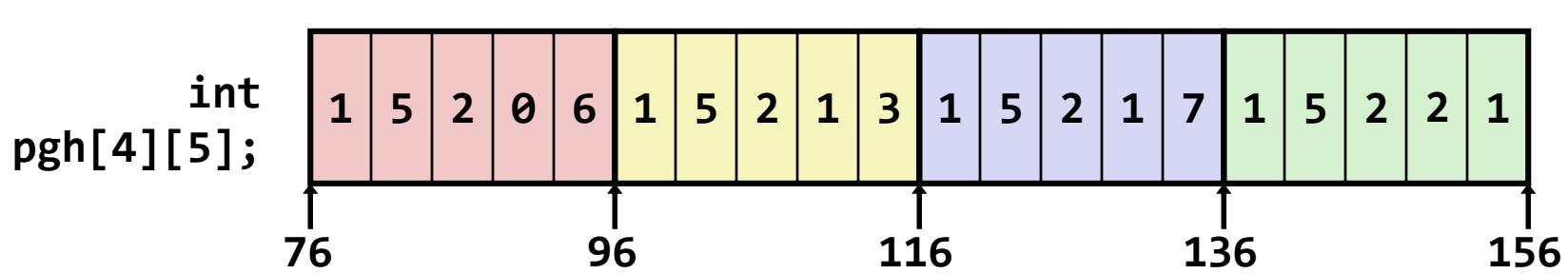
Nested Array Example

- C code

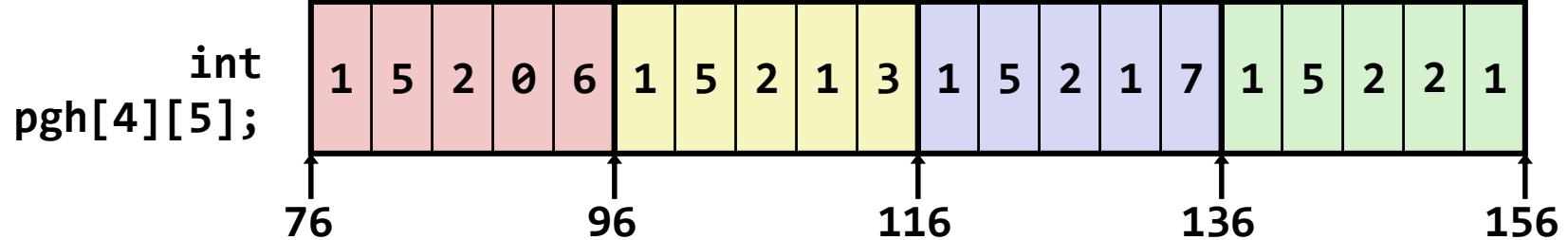
- Variable `pgh` denotes array of 4 elements
 - Allocated contiguously
- Each element is an array of 5 `int`'s
 - Allocated contiguously

```
int pgm[4][5] =  
{{1, 5, 2, 0, 6},  
 {1, 5, 2, 1, 3 },  
 {1, 5, 2, 1, 7 },  
 {1, 5, 2, 2, 1 }};
```

- Row-major ordering of all elements guaranteed



Nested Array Access Example



- Code does not do any bounds checking! (again)
 - Ordering of elements within array guaranteed

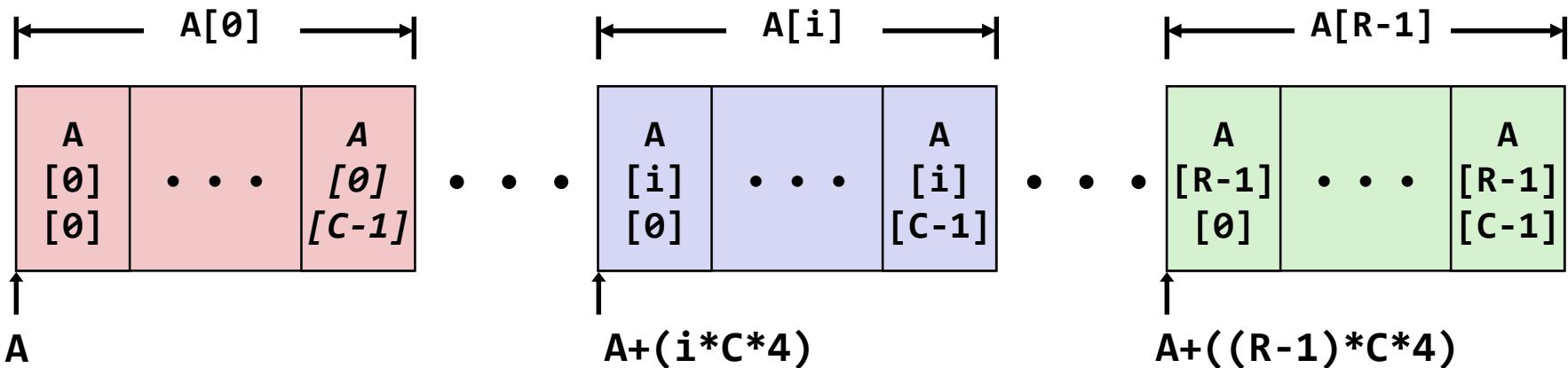
Reference	Address	Value	Guaranteed?
pgh[3][3]	$76+20*3+4*3 = 148$	2	Yes
pgh[2][5]	$76+20*2+4*5 = 136$	1	Yes
pgh[2][-1]	$76+20*2+4*(-1) = 112$	3	Yes
pgh[4][-1]	$76+20*4+4*(-1) = 152$	1	Yes
pgh[0][19]	$76+20*0+4*19 = 152$	1	Yes
pgh[0][-1]	$76+20*0+4*(-1) = 72$??	No

Nested Array Row Access

- **Row vectors**

- $A[i]$ is array of C elements
- Each element of type T requires K bytes
- Starting address: $A + i * (C * K)$

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



Nested Array Row Access Code

- **Row vector**

- `pgh[index]` is array of 5 int's
- Starting address: `pgh + 20 * index`

```
int *get_pgh_zip(long index) {  
    return pgh[index];  
}
```

- **Code**

- Compute and return address
- Compute as `pgh + 4 * (index + 4 * index)`

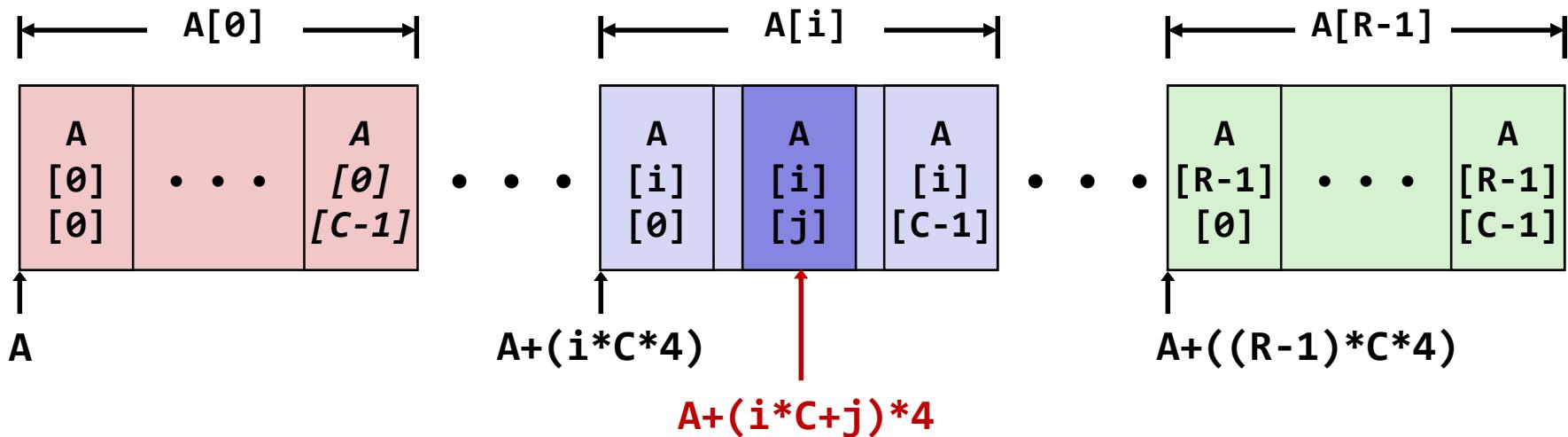
```
# %rdi = index  
leaq (%rdi,%rdi,4),%rax    # 5 * index  
leaq pgh(%rax,4),%rax      # pgh + (20 * index)
```

Nested Array Element Access

- **Array elements**

- $A[i][j]$ is element of type T
- Address: $A + i * (C * K) + j * K = A + (i * C + j) * K$

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



Nested Array Element Access Code

- Array elements

- `pgh[index][digit]` is `int`
 - Address: `pgh + 20 * index + 4 * digit`

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

- Code

- Compute address `pgh + 4 * ((index + 4 * index) + digit)`
 - `movl` performs memory reference

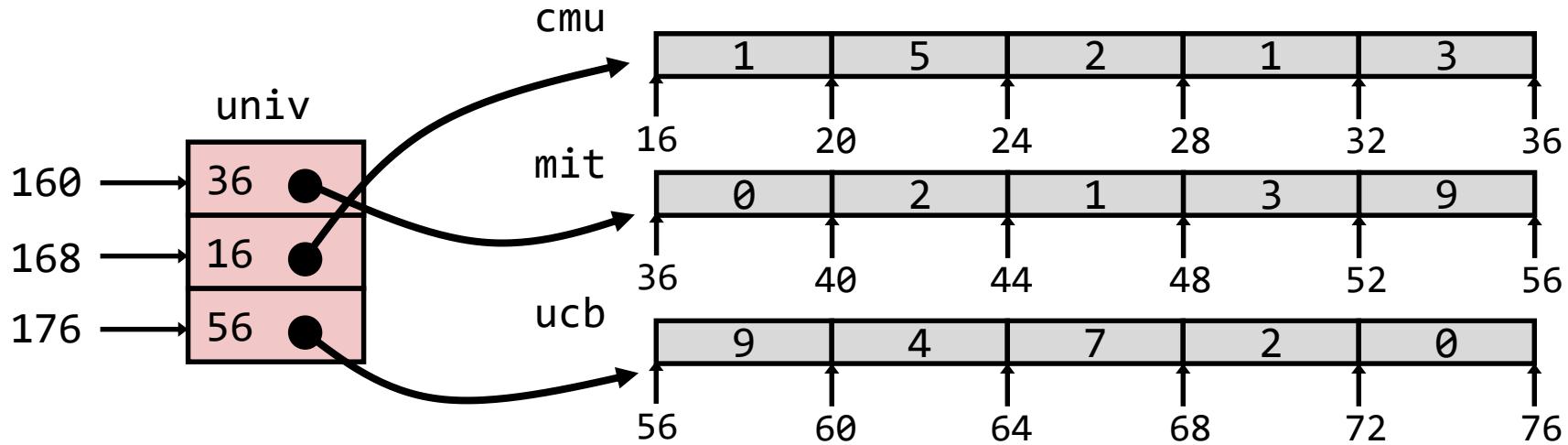
```
# %rdi = index
# %rsi = digit
leaq (%rdi,%rdi,4),%rax      # 5 * index
addl %rax, %rsi                # 5 * index + digit
movl pgh(%rsi,4),%eax        # M[pgh+4*(5*index+digit)]
```

Multi-Level Array

- Example

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

```
typedef int zip_dig[5];  
  
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig mit = { 0, 2, 1, 3, 9 };  
zip_dig ucb = { 9, 4, 7, 2, 0 };  
  
int *univ[3] = {mit, cmu, ucb};
```



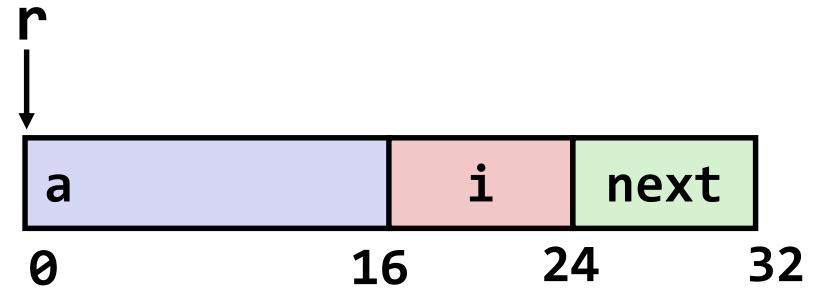
Summary

- **Arrays in C**
 - Contiguous allocation of memory
 - Pointer to first element
 - No bounds checking
- **Compiler optimizations**
 - Compiler often turns array code into pointer code
 - Uses addressing modes to scale array indices
 - Lots of tricks to improve array indexing in loops

Structure

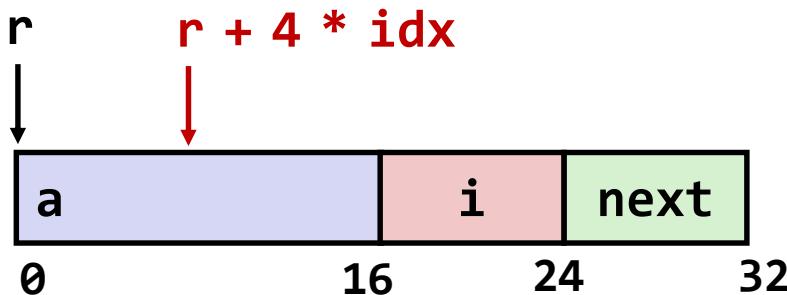
- Structure represented as block of memory
 - Refer to members within structure by names
 - Members may be of different type
- Fields ordered according to declaration
 - Even if another ordering could be more compact
- Compiler determines overall size + positions of fields
 - Machine-level program has no understanding of the structure

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



Structure Referencing Example

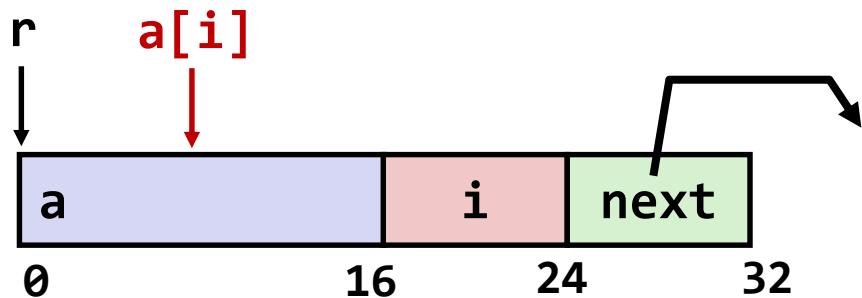
- Generating pointer to array element
 - Offset of each structure member determined at compile time
 - Compute as $r + 4 * idx$



```
struct rec {  
    int a[4];  
    long i;  
    struct rec *next;  
};  
  
int *get_ap  
    (struct rec *r, long idx)  
{  
    return &r->a[idx];  
}
```

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

Following Linked List Example



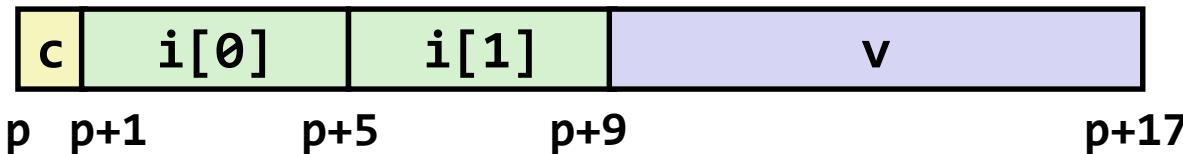
```
# rdi = r
# esi = val
.L11:
    movq    16(%rdi),%rax
    movl    %esi,(%rdi,%rax,4)
    movq    24(%rdi),%rdi
    testq   %rdi,%rdi
    jne     .L11
```

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

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

Structure & Alignment

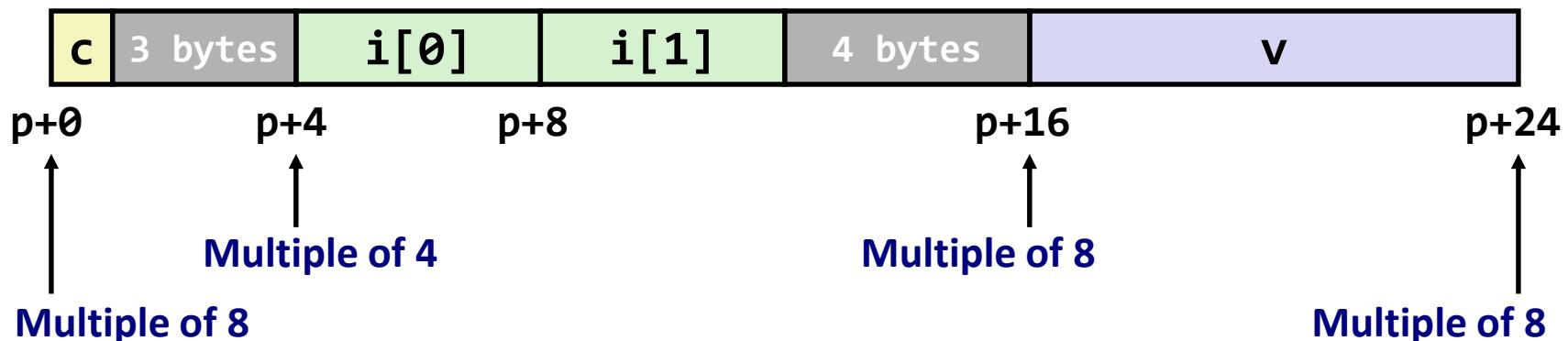
- Unaligned data



```
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

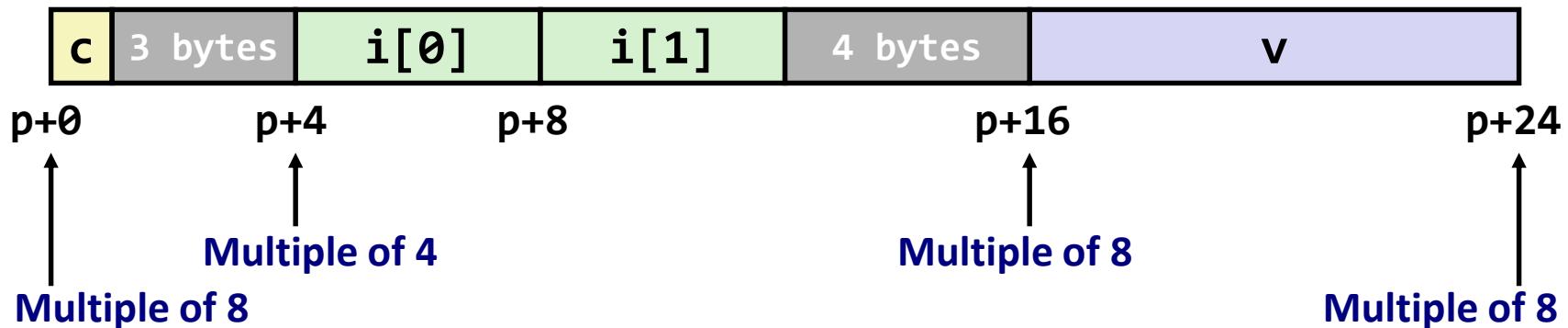
- Aligned data
 - Primitive data type requires **K** bytes
 - Address must be multiple of **K**
 - Required on some machines; advised on x86-64
- Motivation for aligning data
 - Memory accessed by (aligned) chunks of 4 or 8 bytes
 - Inefficient to load or store data that spans quad word boundaries
 - Virtual memory trickier when data spans 2 pages
- Compiler
 - Inserts gaps (or “pads”) in structure to ensure correct alignment of fields

Specific Alignment Cases (x86-64)

- 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: **long**, **double**, **char ***, ...
 - Lowest 3 bits of address must be 000_2

Satisfying Alignment in Structures

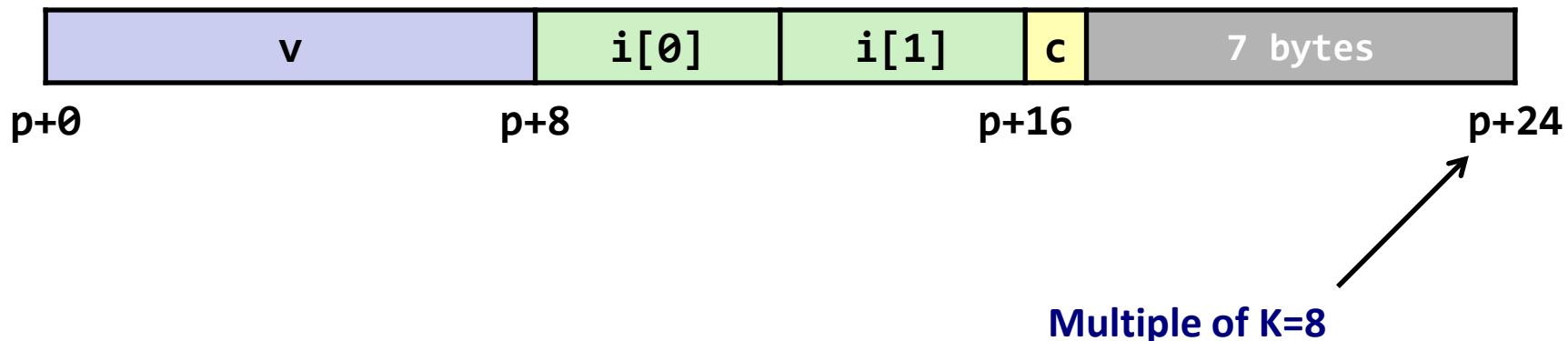
- Offsets within structure
 - Must satisfy each element's alignment requirement
- Overall structure placement
 - Each structure has alignment requirement K
 - $K = \text{Largest alignment of any element}$
 - Initial address & structure length must be multiple of K
- Example: $K = 8$ due to double element



Overall Alignment Requirement

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

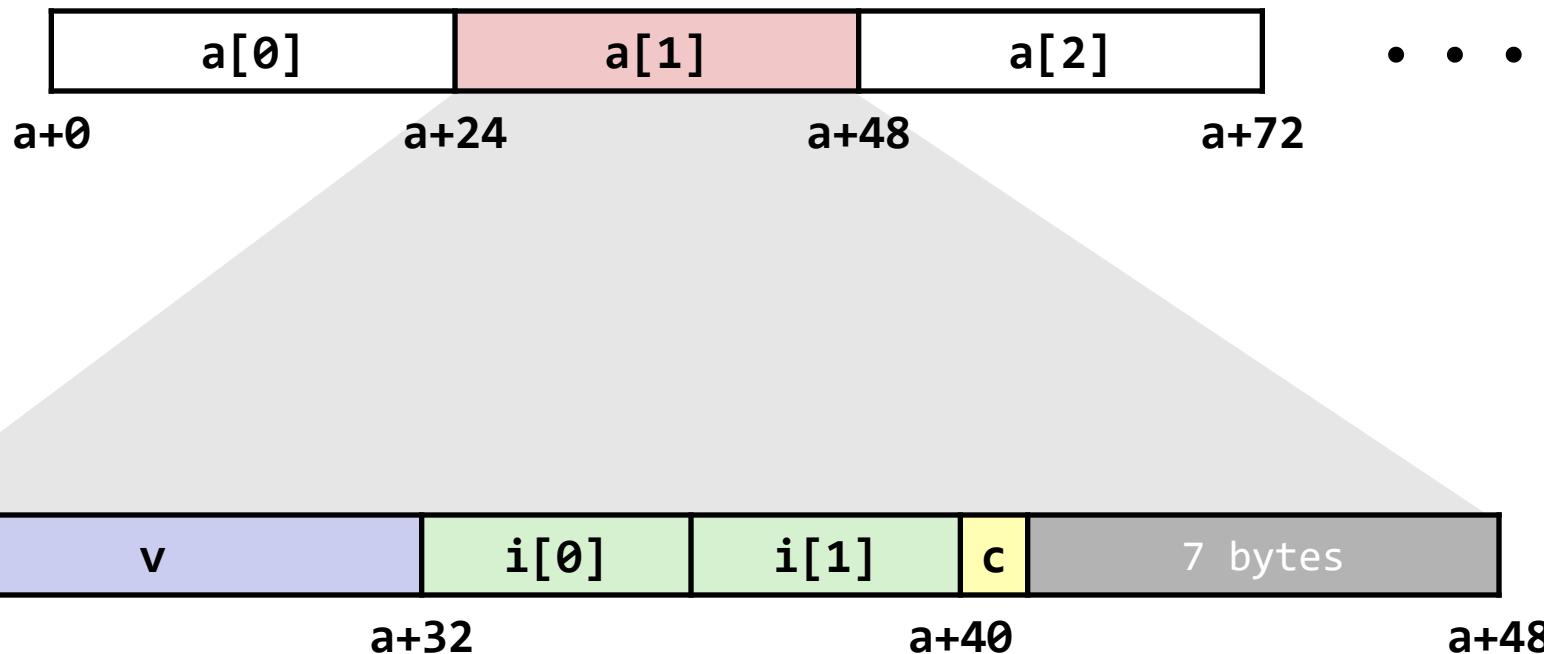
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



Arrays of Structures

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

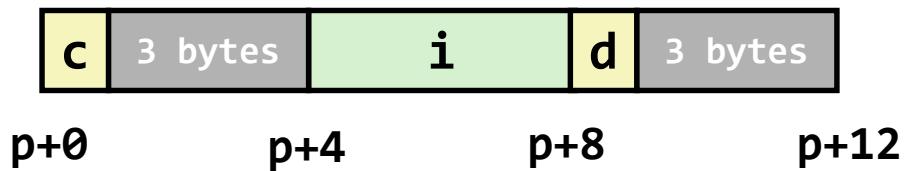
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



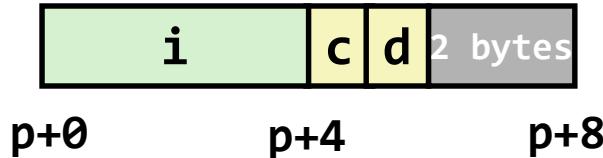
Saving Spaces

- Put large data types first

```
struct S3 {  
    char c;  
    int i;  
    char d;  
} *p;
```



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



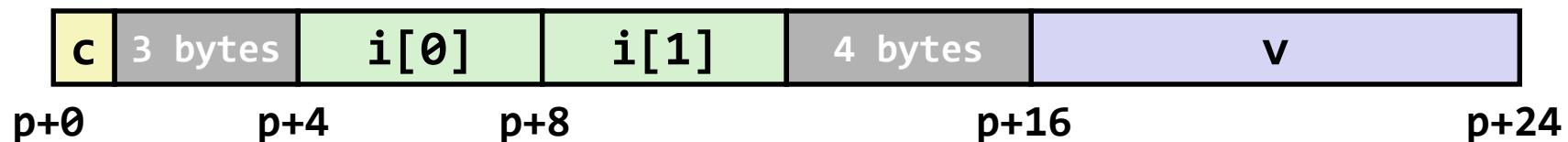
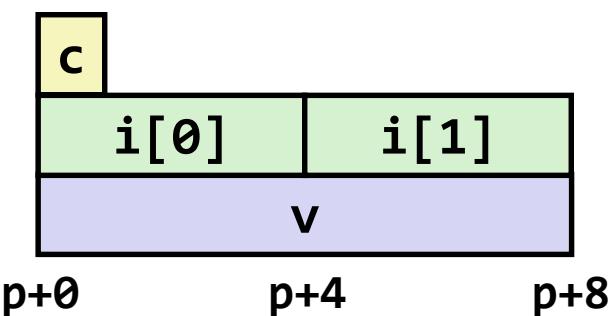
Union Allocation

- Principles

- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time

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

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



Summary

- **Structures**

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

- **Unions**

- Overlay declarations
- Way to circumvent type system