Machine-Level Programming: Data

Prof. Hyuk-Yoon Kwon

https://sites.google.com/view/seoultech-bigdata

Most parts are based on slides written by Brayant and O'Hallaon, CMU
(http://csapp.cs.cmu.edu/3e/instructors.html)

Today: Data

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Floating Point**

Array Allocation

Basic Principle

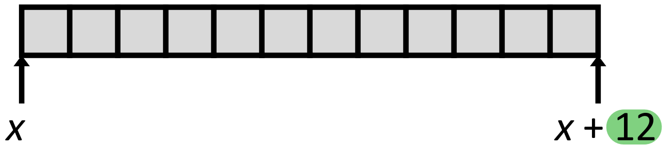
T **A[*L*]** ;

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

(end point) – (starting point)

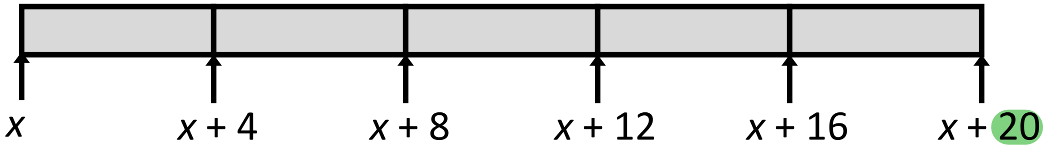
`char string[12];`

1 x 12



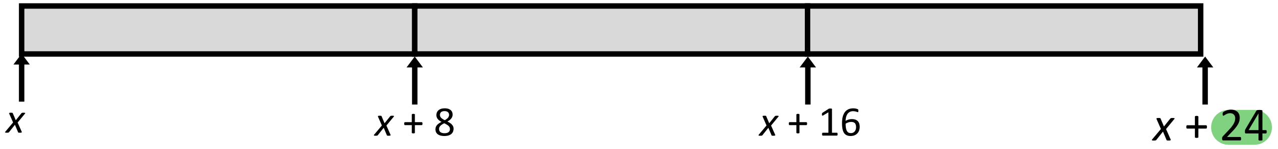
`int val[5];`

4 x 5



`double a[3];`

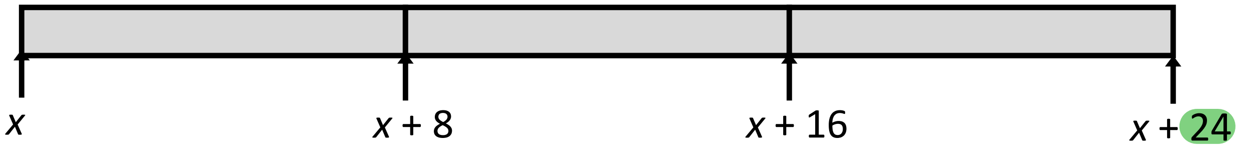
8 x 3



`char *p[3];`

8 x 3

in 64bit architecture.



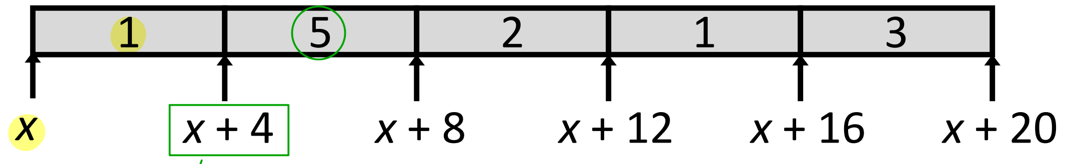
Array Access

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: Type *T**

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

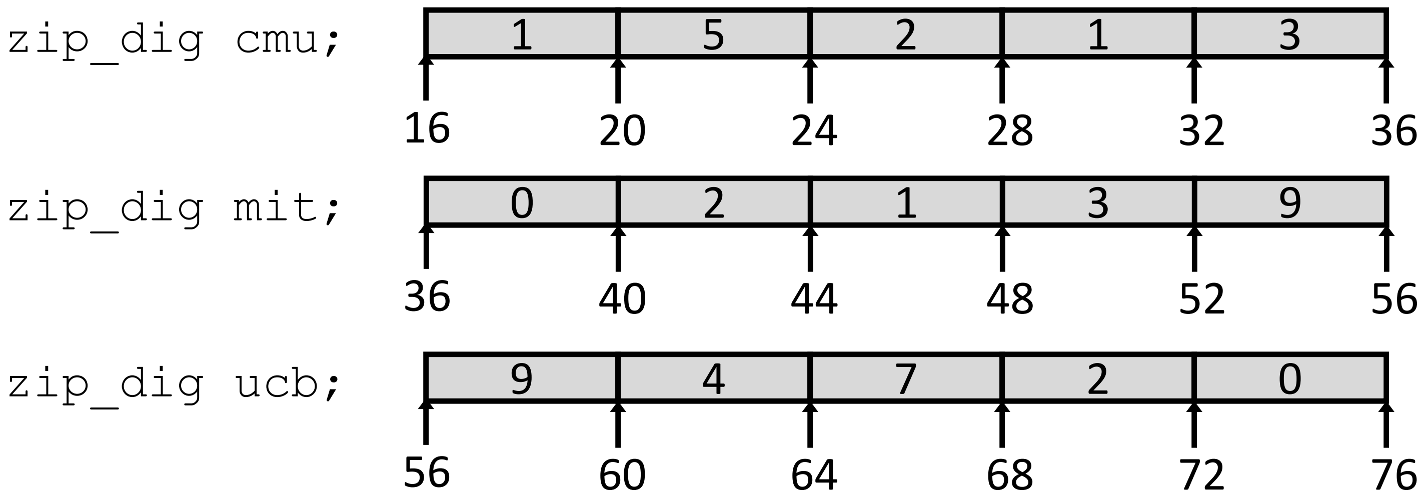


Reference

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	x
`val+1` (addition means next element in array identifier)	`int *`	$\boxed{x+4}$
`&val[2]`	`int *`	x + 8
`val[5]`	`int`	? … out of boundary of array? unknown value?
`*(val+1)`	`int`	5

Array Example

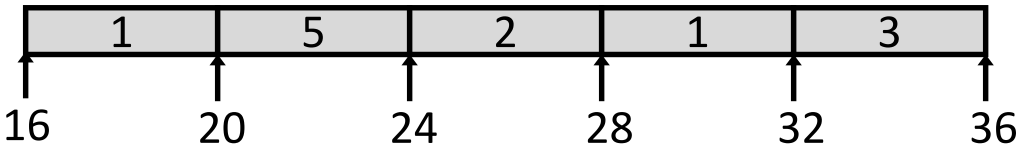
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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



- **Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"**
- **Example arrays were allocated in successive 20 byte blocks**
  - Not guaranteed to happen in general

Array Accessing Example

```
zip_dig cmu;
```




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

array identifier

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

due to sizeof(int)

(%rdi)+4(%rsi)

- **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
```

Multidimensional (Nested) Arrays

Declaration

`T A[R][C];`

- 2D array of data type *T*
- *R* rows, *C* columns
- Type *T* element requires *K* bytes = sizeof(T)

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

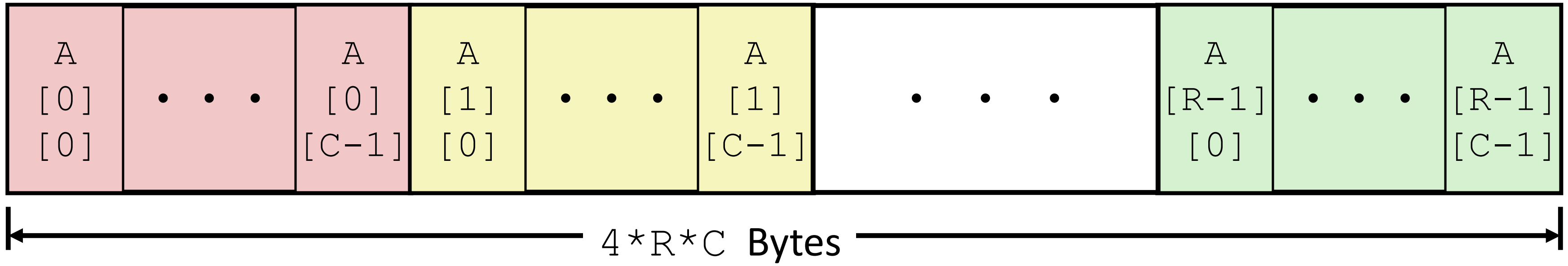
Array Size

- *R* * *C* * *K* bytes

Arrangement

- Row-Major Ordering

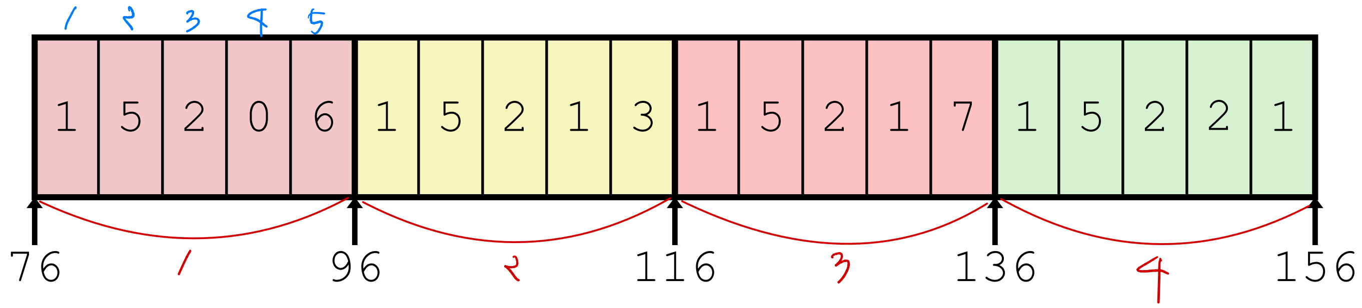
```
int A[R][C];
```



Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

`zip_dig pgh[4];`



- **“`zip_dig pgh[4]`” equivalent to “`int pgh[4][5]`”**
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**’s, allocated contiguously
- **“Row-Major” ordering of all elements in memory**

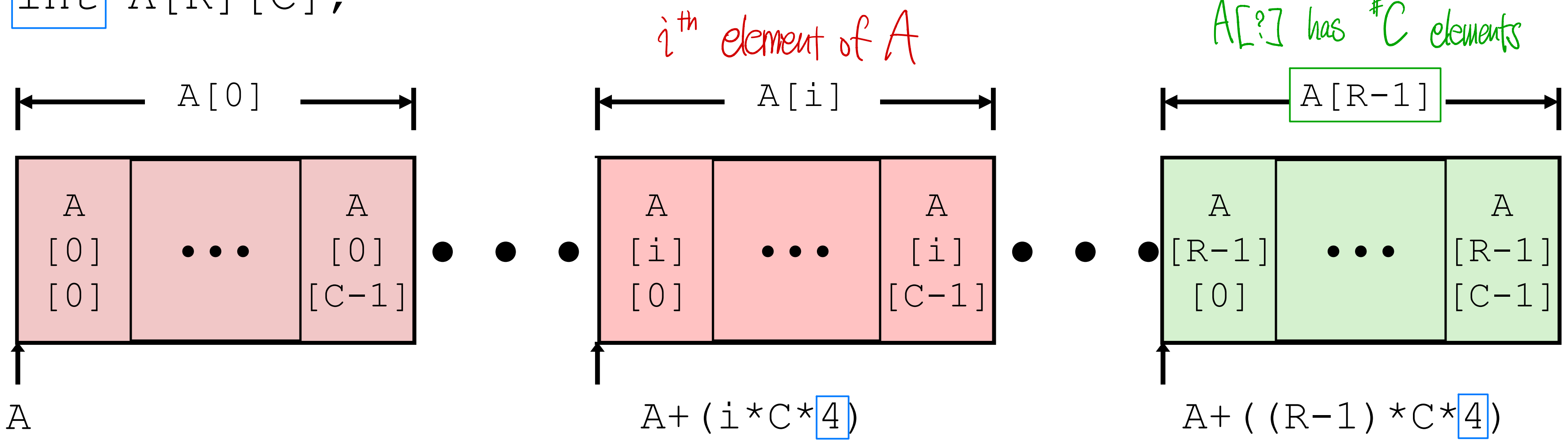
Nested Array Row Access

Row Vectors

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

$$\&A[i][j] = A + (i * C * 4) + (j * 4)$$

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



Nested Array Row Access Code

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1

pgh

20 bytes

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

(%rdi) + 4(%rdi) = 5(%rdi)

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

4(%rax) + pgh

Row Vector

- **`pgh[index]`** is array of **5 `int`**’s
  - 5 * sizeof(int)
- Starting address **`pgh+20*index`**

Machine Code

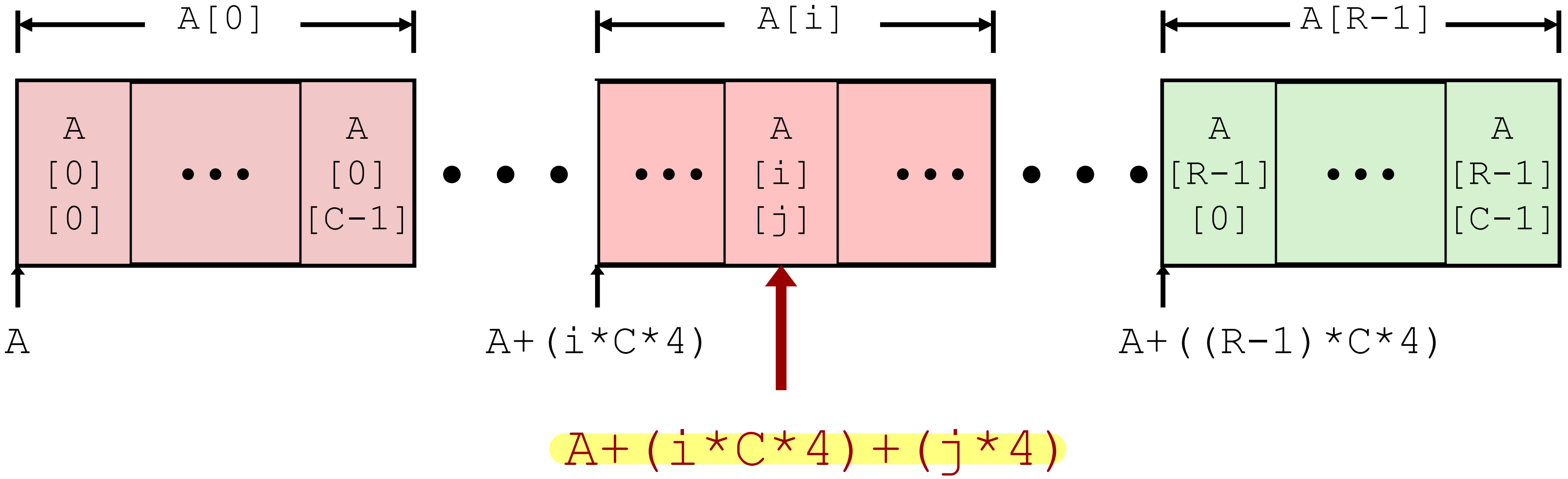
- Computes and returns address
- Compute as **`pgh + 4*(index+4*index)`**

Nested Array Element Access

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];
```



Nested Array Element Access Code

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1

pgh

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

%rsi = %rsi + %rax   %rax = 5(%rdi)

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

%rdi

%rsi

%eax = 4(%rsi)+pgh = 4(%rsi + 5(%rdi)) + pgh

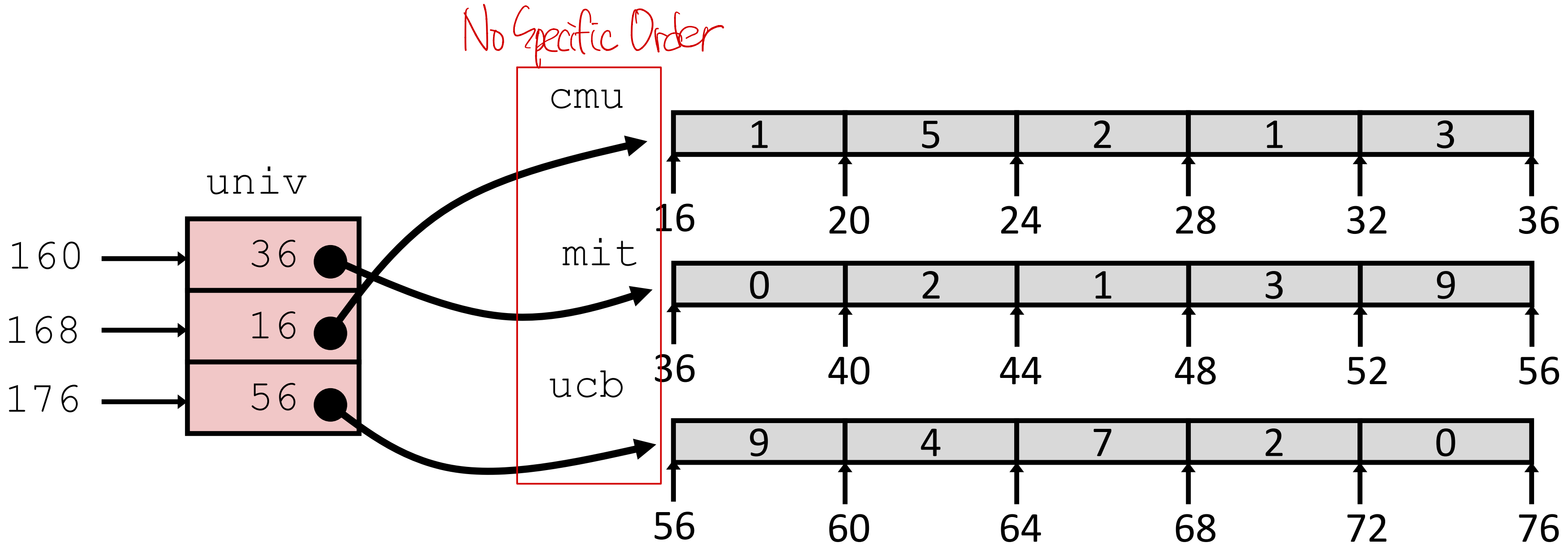
- **Array Elements**
  - **`pgh[index][dig]`** is **`int`**
  - Address: **`pgh + 20*index + 4*dig`**
    - = **`pgh + 4*(5*index + dig)`**

Multi-Level Array Example

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

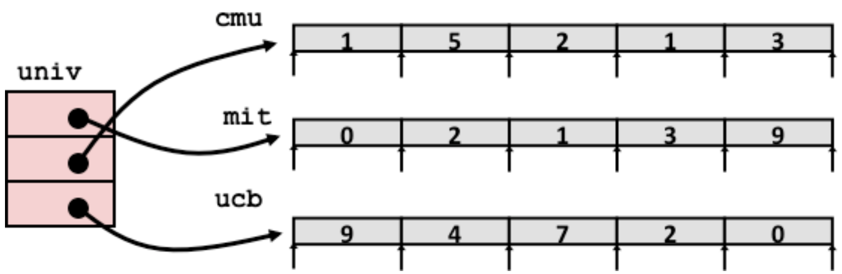
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

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



Element Access in Multi-Level Array

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



%rsi = 8(%rdi)+univ + %rsi   %rsi = 4(%rsi)

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

%eax = dereference of %rsi

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

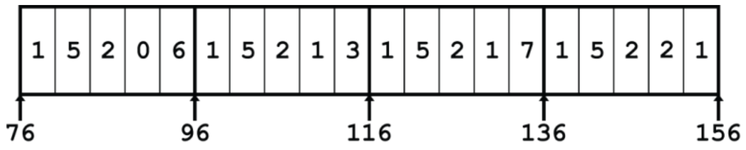
Array Element Accesses

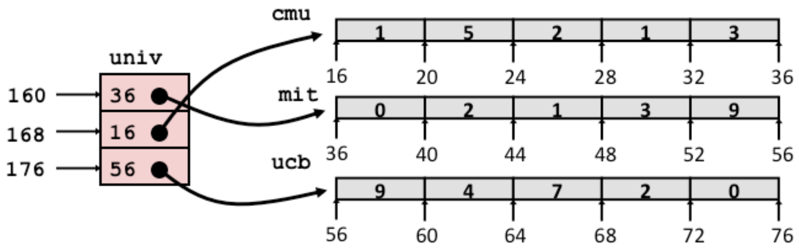
Nested array

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

Multi-level array

```
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]`

N X N Matrix Code

- **Fixed dimensions**
  - Know value of N at compile time
- **Variable dimensions, explicit indexing**
  - Traditional way to implement dynamic arrays
- **Variable dimensions, implicit indexing**
  - Now supported by gcc

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j)
{
  return a[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j)
{
  return a[IDX(n,i,j)];
}
```

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```

16 X 16 Matrix Access

- **Array Elements**
  - Address **A** + $i * (C * K) + j * K$
  - C = 16, K = 4

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}
```

```
  # a in %rdi, i in %rsi, j in %rdx
  salq    $6, %rsi              # 64*i
  addq    %rsi, %rdi            # a + 64*i
  movl    (%rdi,%rdx,4), %eax   # M[a + 64*i + 4*j]
  ret
```

n X n Matrix Access

- **Array Elements**
  - Address **A** + $i * (C * K) + j * K$
  - C = n, K = 4
  - Must perform integer multiplication

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j)
{
  return a[i][j];
}
```

```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi              # n*i
  leaq    (%rsi,%rdi,4), %rax     # a + 4*n*i
  movl    (%rax,%rcx,4), %eax     # a + 4*n*i + 4*j
  ret
```

Practice: Accessing Array

- **Consider the following declarations:**

```
int P[5];
short Q[2];
int **R[9];
double *S[10];
short *T[2];
```

Fill in the following table by programming actual C codes

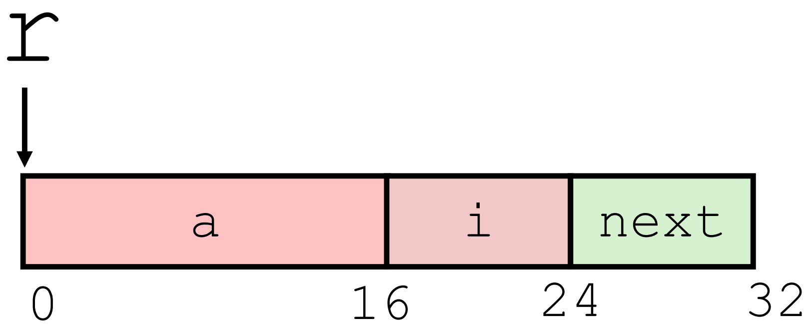
Array	Size of one element		Total size	Start Address	Address for element *i*
P	4	5	20	X_P =	$X_P + 4i$
Q	2	2	4	X_Q =	$X_Q + 2i$
R	8	9	72	X_R =	$X_R + 8i$
S	8	10	80	X_S =	$X_S + 8i$
T	8	2	16	X_T =	$X_T + 8i$

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Floating Point**

Structure Representation

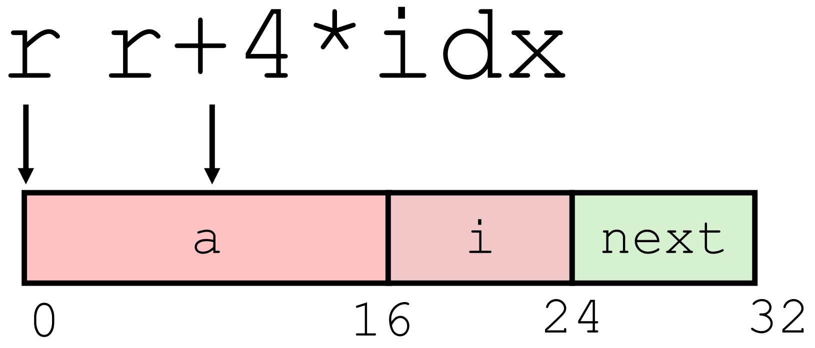
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- **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**

Generating Pointer to Structure Member

```
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];
}
```

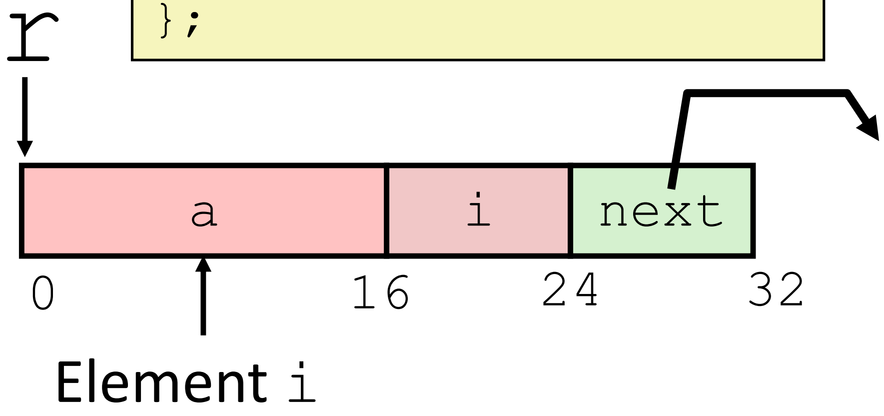
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Following Linked List

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

- **C Code**

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

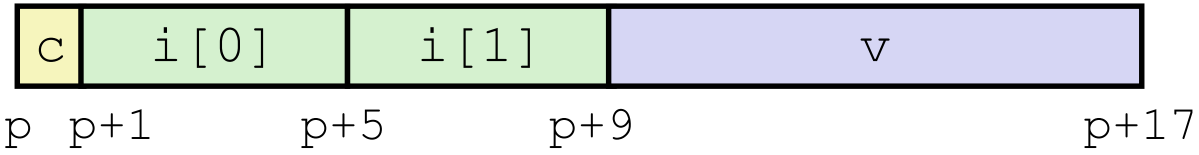


Register	Value
%rdi	r
%rsi	val

```
.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
```

Structures & Alignment

Unaligned Data

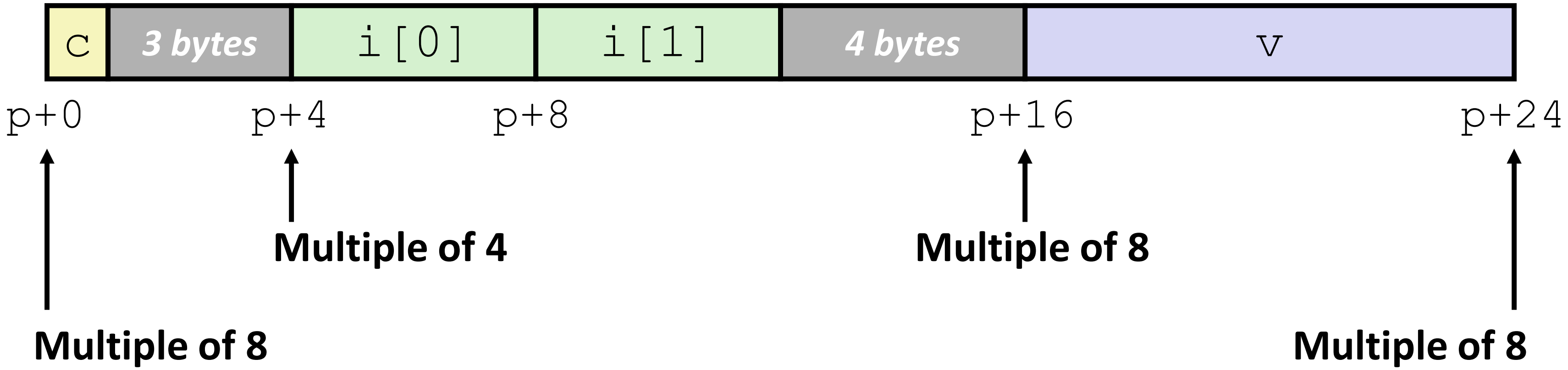



```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

단위 맞춰서 cost 감소시키려는 목적
메모리 access 횟수 줄여야 좋다!

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

32bit or 64 bit

- 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

Specific Cases of Alignment (x86-64)

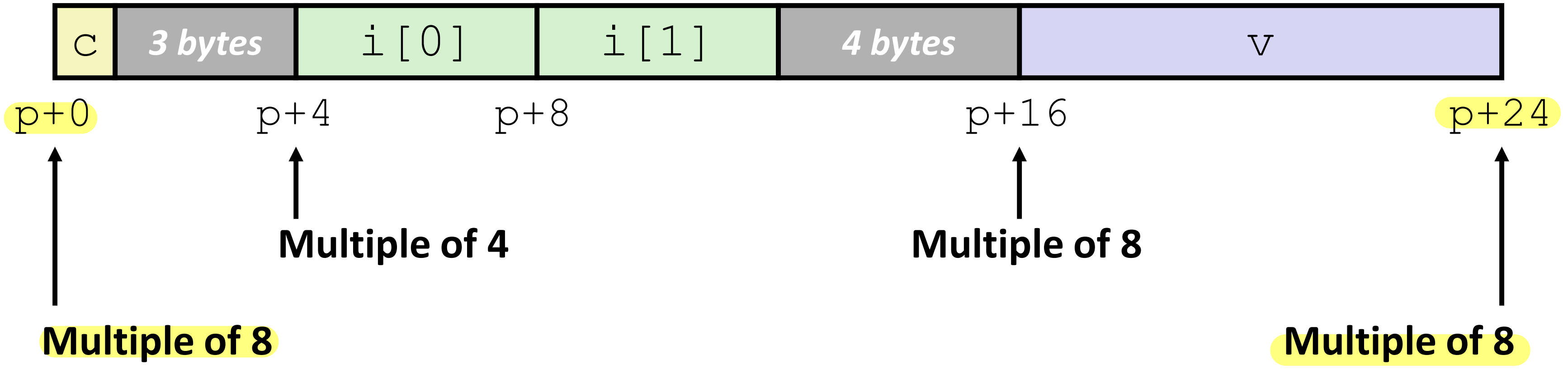
- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2 $2 = 10_2$
- **4 bytes: `int`, `float`, …**
  - lowest 2 bits of address must be 00_2 $4 = 100_2$
- **8 bytes: `double`, `long`, `char *`, …**
  - lowest 3 bits of address must be 000_2 $8 = 1000_2$
- **16 bytes: `long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2 $16 = 10000_2$

Satisfying Alignment with Structures

intra-struct alignment
inner-struct alignment

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

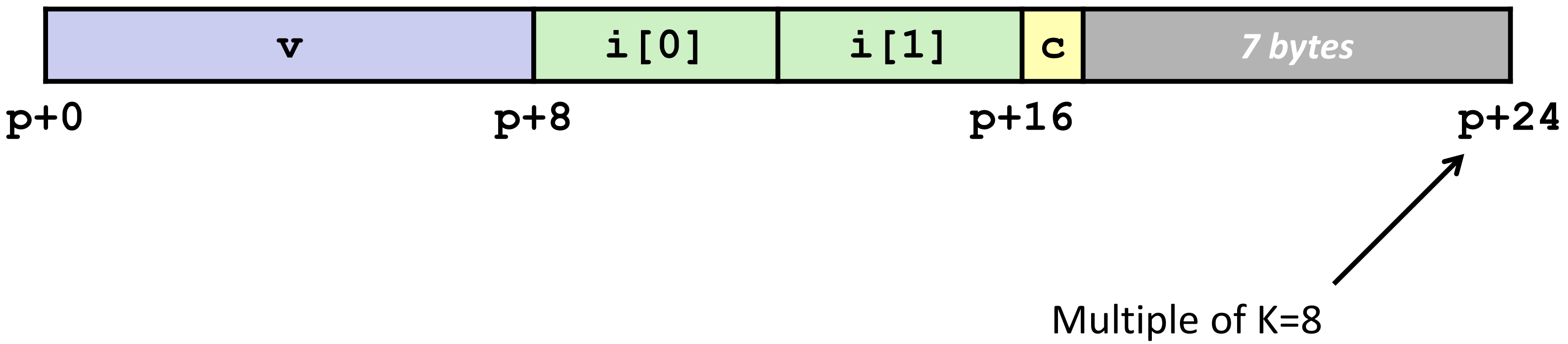
- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- **Example:**
  - K = 8, due to **`double`** element



Meeting Overall Alignment Requirement

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

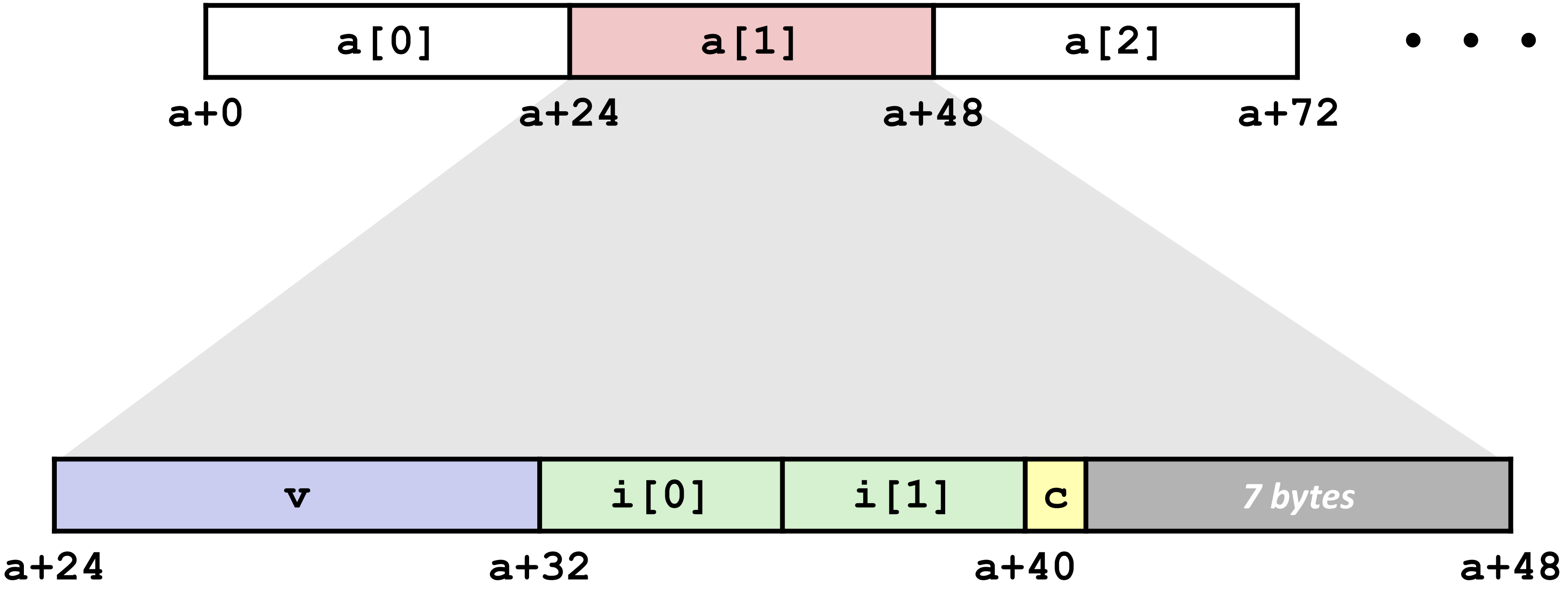
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

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

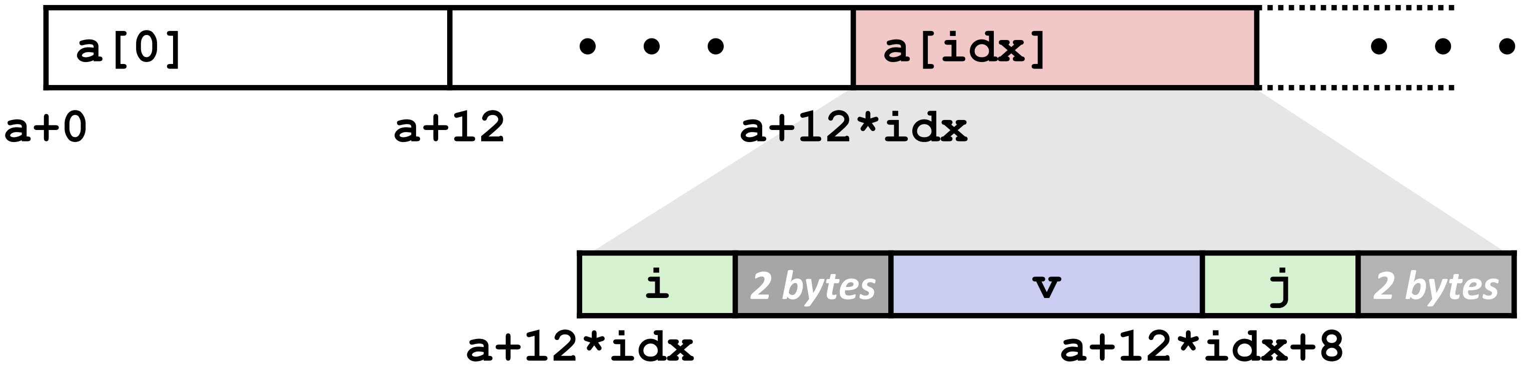
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

- **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
```

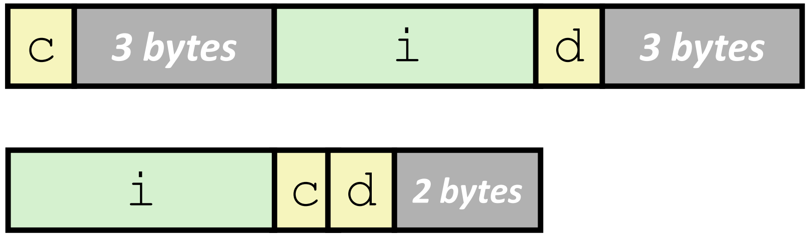
Saving Space

- **Put large data types first**

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

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

- **Effect (K=4)**



Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Floating Point**

FP Basics

- Arguments passed in `%xmm0`, `%xmm1`, ...
- Result returned in `%xmm0`
- All XMM registers caller-saved

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

```
  # x in %xmm0, y in %xmm1
  addss   %xmm1, %xmm0
  ret
```

```
 # x in %xmm0, y in %xmm1
  addsd   %xmm1, %xmm0
  ret
```

FP Memory Referencing

- **Integer (and pointer) arguments passed in regular registers**
- **FP values passed in XMM registers**
- **Different mov instructions to move between XMM registers, and between memory and XMM registers**

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
  # p in %rdi, v in %xmm0
  movapd  %xmm0, %xmm1    # Copy v
  movsd   (%rdi), %xmm0   # x = *p
  addsd   %xmm0, %xmm1    # t = x + v
  movsd   %xmm1, (%rdi)   # *p = t
  ret
```

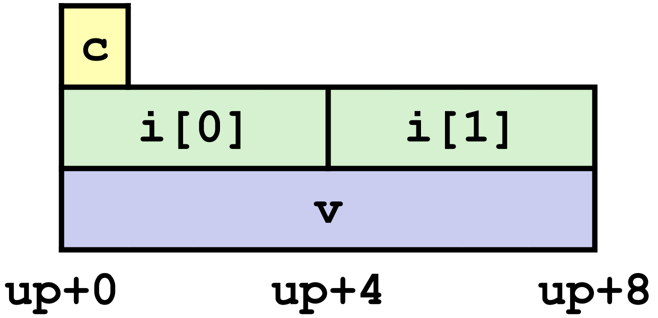
Machine-Level Programming: Data

- **Unions**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

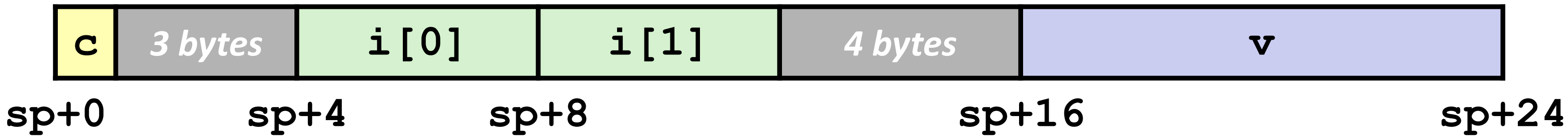
Union Allocation

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

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

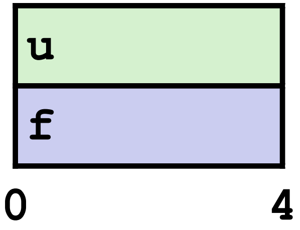



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(float) u`? yes!

Same as `(unsigned) f`? yes!

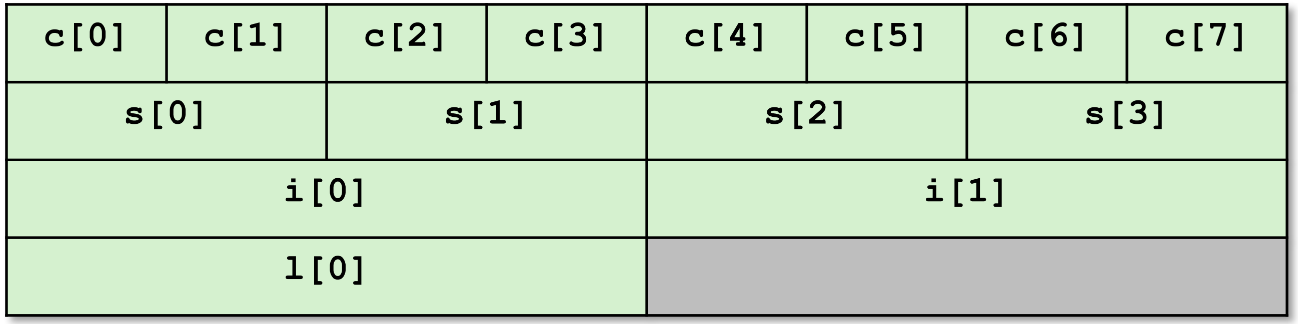
Byte Ordering Revisited

- **Idea**
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- **Big Endian**
  - Most significant byte has lowest address
  - Sparc
- **Little Endian**
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS

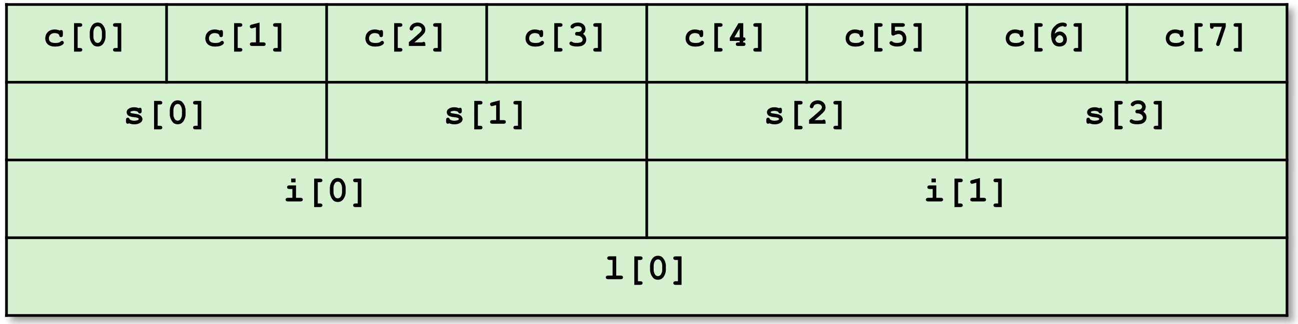
Byte Ordering Example

```c
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

32-bit



64-bit



Byte Ordering Example (Cont).

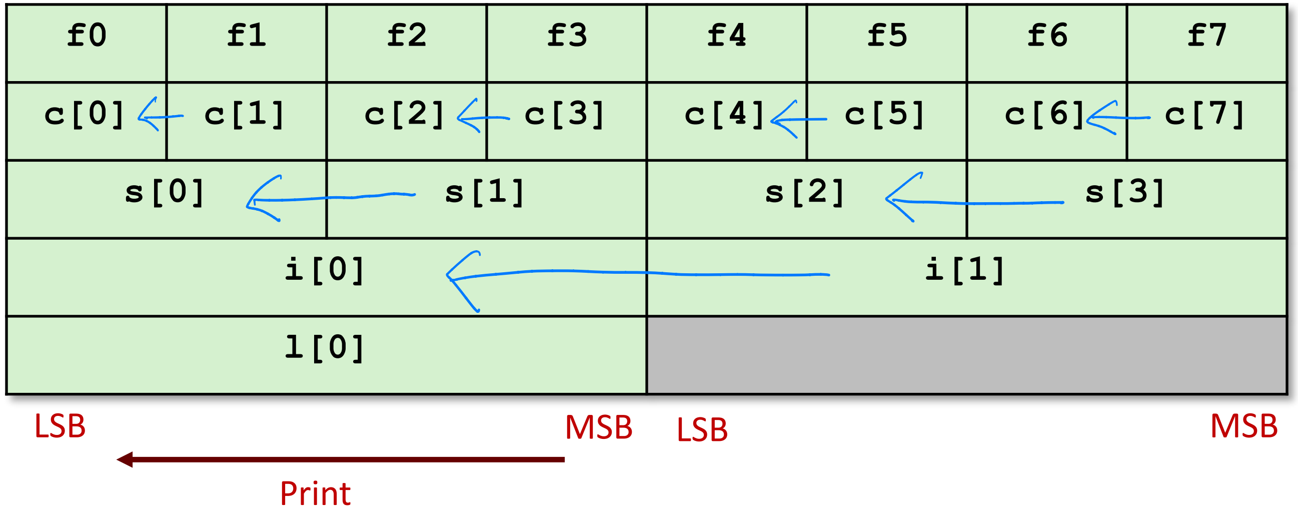
```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Read MSB First

Little Endian

LSB ⇒ Lowest Memory Address



Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Read MSB First

Big Endian

f0	f1	f2	f3	f4	f5	f6	f7
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

MSB LSB MSB LSB

Print →

Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Byte Ordering on x86-64

Little Endian

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

LSB ← MSB

Print

Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Practice: Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```