

# Chapter 6

## Machine-Level Programming IV: Data

---

SCS@TJU



# Outline

---

## Arrays

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

## Structures

- Allocation
- Access
- Alignment

## Union

Arrays : One dimensional

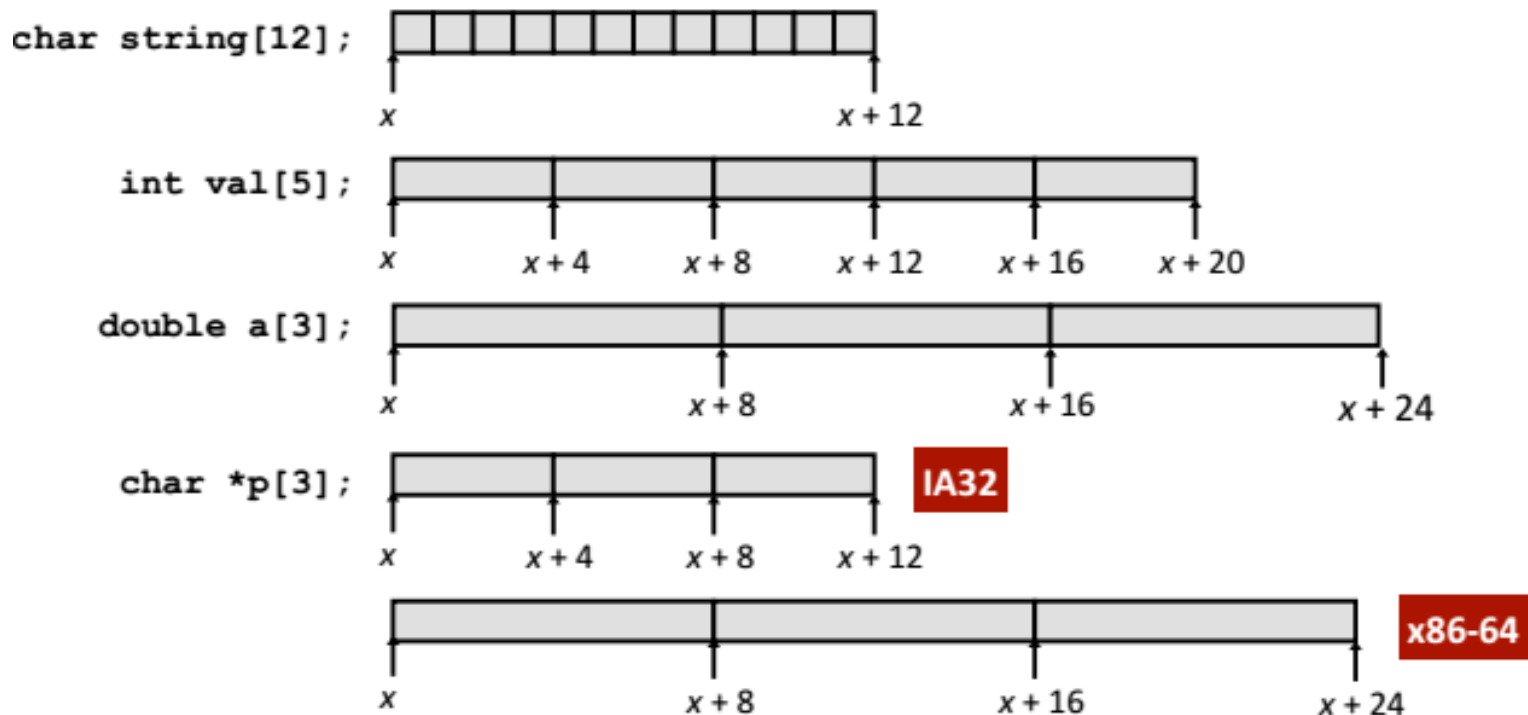
---

# Array Allocation

## Basic Principle

$T$   $A[L]$ ;

- Array of data type  $T$  and length  $L$
- Contiguously allocated region of  $L * \text{sizeof}(T)$  bytes

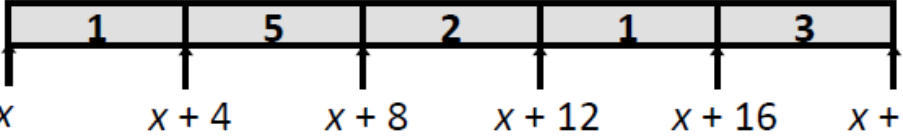


# Array Access

## Basic Principle

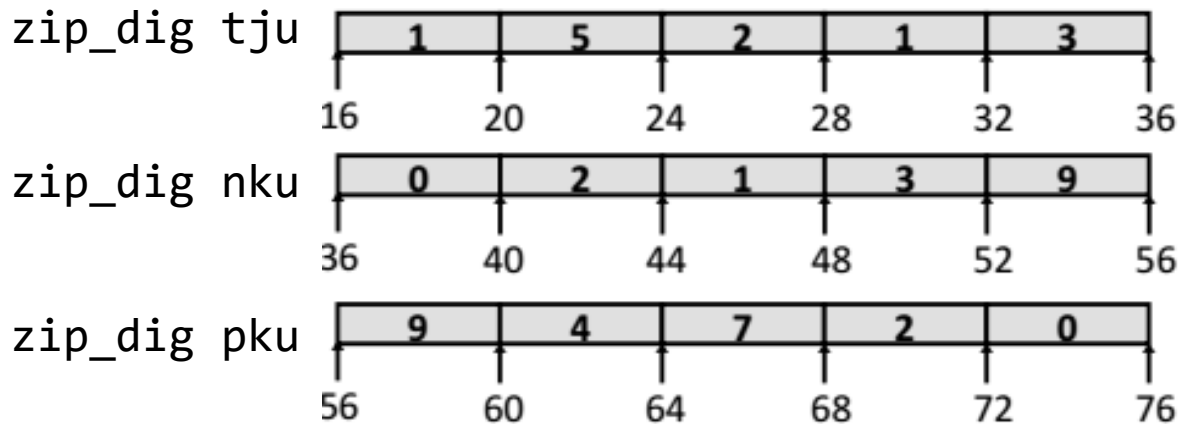
$T$   $A[L]$ ;

- Array of data type  $T$  and length  $L$
- Identifier  $A$  can be used as a pointer to array element 0: Type  $T *$

<code>int val[5];</code>		
	$x$	$x + 4$
	$x + 8$	$x + 12$
	$x + 16$	$x + 20$
<b>Reference</b>	<b>Type</b>	<b>Value</b>
<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>	$x+4i$

# Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];
zip_dig tju = { 1, 5, 2, 1, 3 };
zip_dig nku = { 0, 2, 1, 3, 9 };
zip_dig pku = { 9, 4, 7, 2, 0 };
```



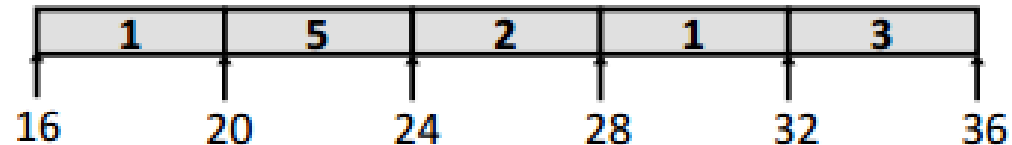
Declaration “zip\_dig tju” equivalent to “int tju[5]”

Example arrays were allocated in successive 20 byte blocks

- Not guaranteed to happen in general

# Array Accessing Example

zip\_dig tju



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

IA32

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4), %eax # z[dig]
```

Register %edx contains starting address of array

Register %eax contains array index

Desired digit at

$$4 * \%eax + \%edx$$

Use memory reference

(%edx, %eax, 4)

# Array Loop Example (IA32)

---

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

```
# %edx = z
movl $0, %eax          # %eax = i
.L4:                  # loop:
addl $1, (%edx,%eax,4) # z[i]++
addl $1, %eax          # i++
cmpl $5, %eax         # i:5
jl .L4                # if <=, goto loop
```



# Pointer Loop Example (IA32)

```
void zincr_p(zip_dig z) {  
    int *zend = z+ZLEN;  
    do {  
        (*z)++;  
        z++;  
    } while (z != zend);  
}
```



```
void zincr_v(zip_dig z) {  
    void *vz = z;  
    int i = 0;  
    do {  
        (*((int *) (vz+i)))++;  
        i += ISIZE;  
    } while (i != ISIZE*ZLEN);  
}
```

```
    movl $0, %eax  
.L8:  
    addl $1, (%edx,%eax)  
    addl $4, %eax  
    cmpl $20, %eax  
    jne .L8
```

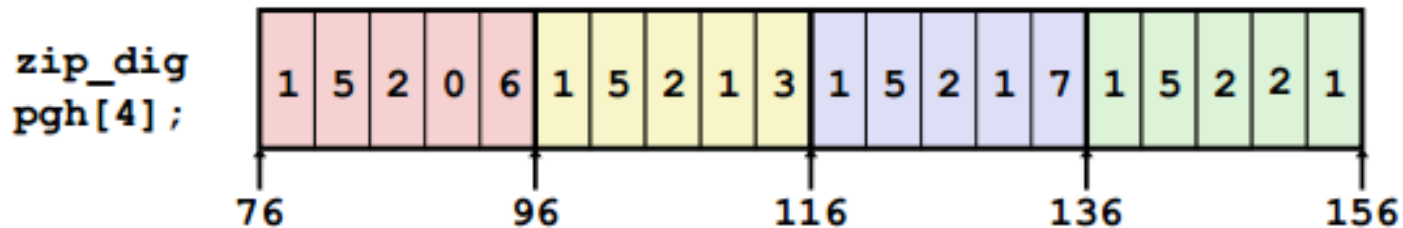
```
# edx = z = vz  
# i = 0  
# loop:  
# Increment vz+i  
# i += 4  
# Compare i:20  
# if !=, goto loop
```

# Arrays : Multi-dimensional

---

# 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]” 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 guaranteed

# Multidimensional (Nested) Arrays

## Declaration

```
T A[R][C];
```

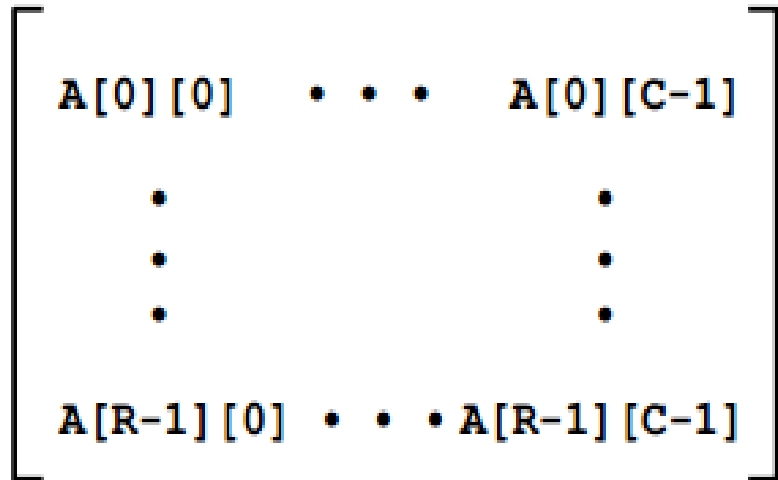
- 2D array of data type T
- R rows C columns
- Type T element requires K bytes

## Array size

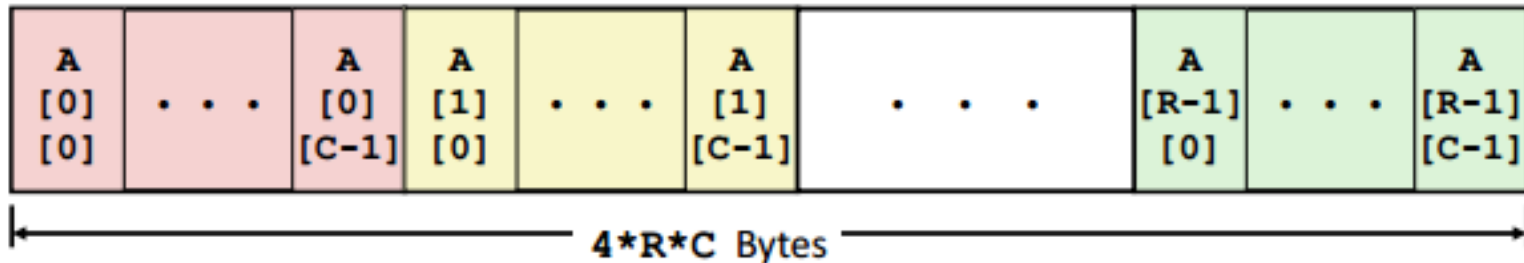
- R \* C \* K bytes

## Arrangement

- Row-Major Ordering



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

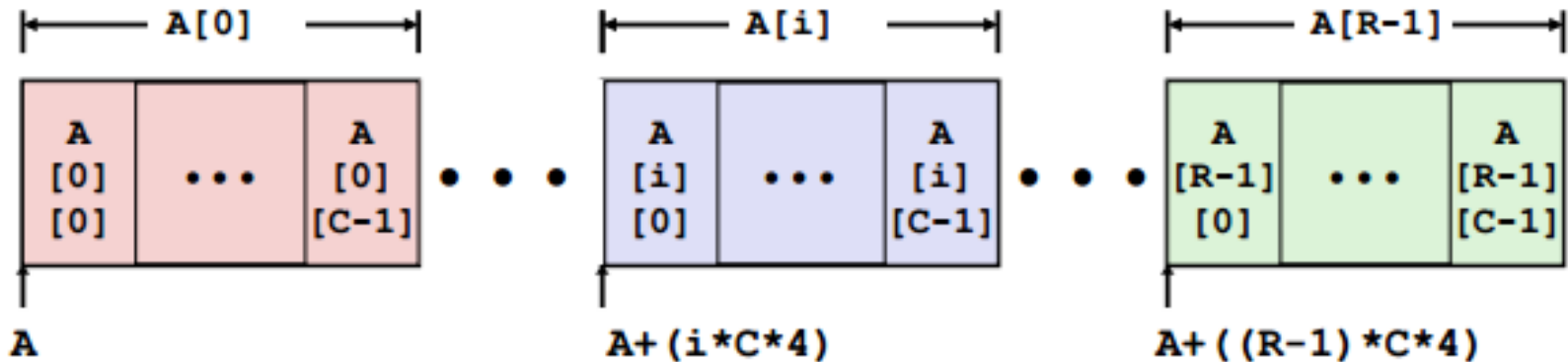


# Nested Array Row Access

## 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

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

```
#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 }};
```

```
# %eax = index
leal (%eax,%eax,4),%eax      # 5 * index
leal pgh(,%eax,4), %eax     # pgh + (20 * index)
```

## Row Vector

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

## IA32 Code

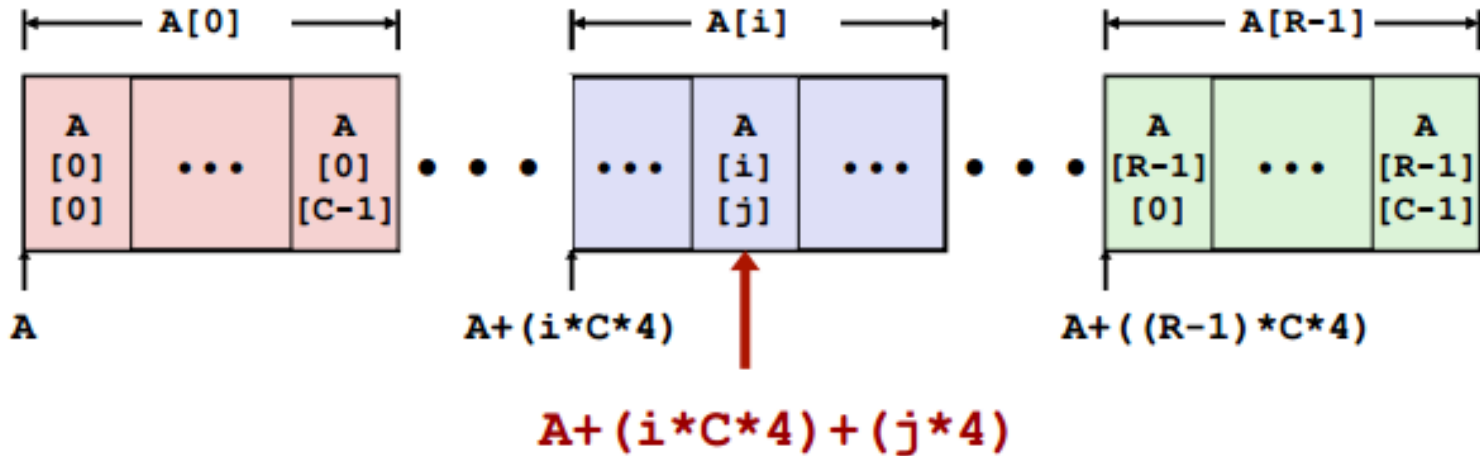
- Computes and returns address
- Compute as `pgh + 4 * (index + 4 * index)`

# Nested Array Row 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

---

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

```
movl 8(%ebp), %eax      # index
leal (%eax,%eax,4), %eax # 5*index
addl 12(%ebp), %eax     # 5*index+dig
movl pgh(,%eax,4), %eax # offset 4*(5*index+dig)
```

## Array Elements

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

## IA32 Code

- Computes address:  $pgh + 4 * ((index + 4 * index) + dig)$



# Arrays : Multi-Level

---

# Multi-Level Array Example

```
zip_dig tju = { 1, 5, 2, 1, 3 };  
zip_dig nku = { 0, 2, 1, 3, 9 };  
zip_dig pku = { 9, 4, 7, 2, 0 };
```

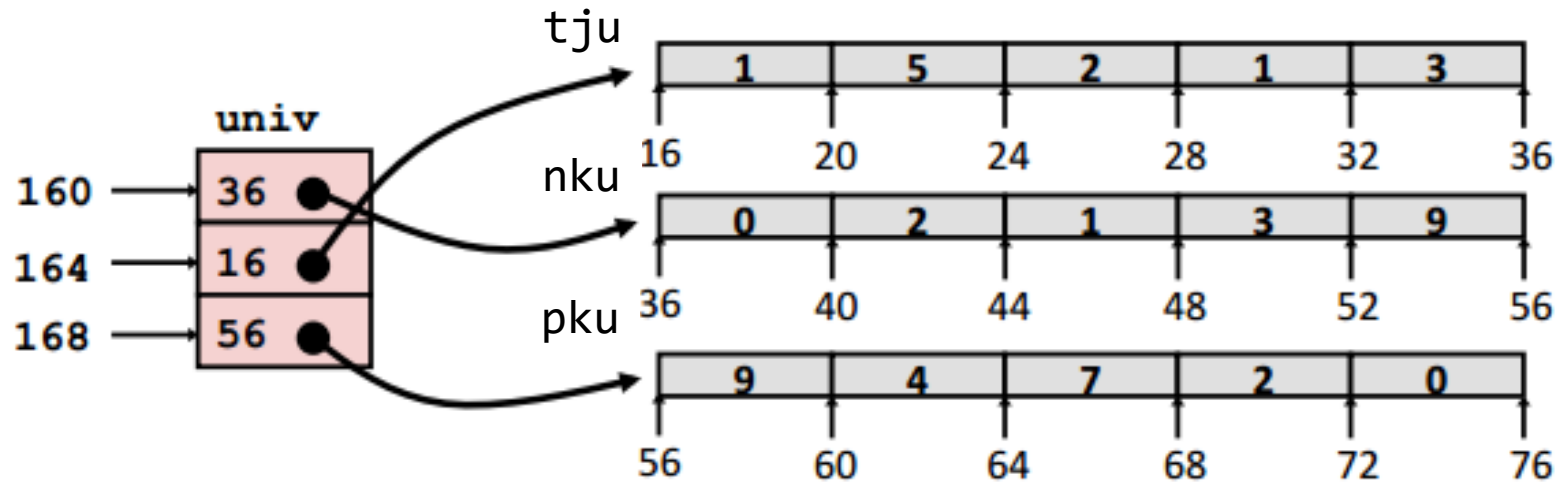
```
#define UCOUNT 3  
int *univ[UCOUNT] = {nku, tju, pku};
```

Variable univ denotes array of 3 elements

Each element is a pointer

- 4 bytes

Each pointer points to array of int's



# Element Access in Multi-Level Array

---

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

```
movl 8(%ebp), %eax      # index
movl univ(,%eax,4), %edx # p = univ[index]
movl 12(%ebp), %eax     # dig
movl (%edx,%eax,4), %eax # p[dig]
```

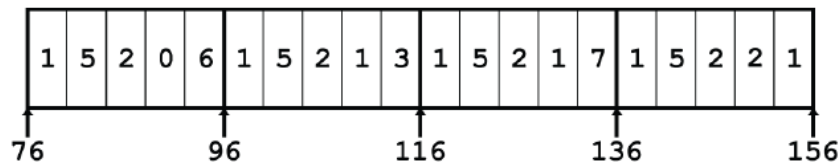
## Computation (IA32)

- Element access  $\text{Mem}[\text{Mem}[\text{univ} + 4 \cdot \text{index}] + 4 \cdot \text{dig}]$
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

# Array Element Accesses

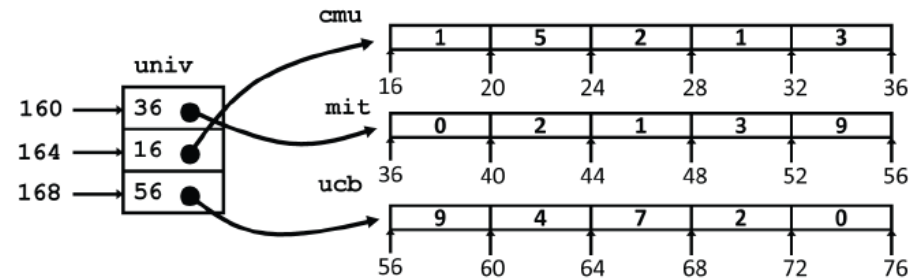
## Nested array

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



## Multi-level array

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



Accesses looks similar in C but addresses very different:

$\text{Mem}[\text{pgh} + 20 * \text{index} + 4 * \text{dig}]$

$\text{Mem}[\text{Mem}[\text{univ} + 4 * \text{index}] + 4 * \text{dig}]$

# n x n Matrix Code

---

## Fixed dimensions

- Know value of N at compile time

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

## Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

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

## Variable dimensions implicit indexing

- Now supported by gcc

```
int var_ele
(int n, int a[n][n], int i, int 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, int i, int j) {  
    return a[i][j];  
}
```

```
movl 12(%ebp), %edx      # i  
sall $6, %edx           # i*64  
movl 16(%ebp), %eax     # j  
sall $2, %eax           # j*4  
addl 8(%ebp), %eax      # a + j*4  
movl (%eax,%edx), %eax  # *(a + j*4 + i*64)
```

# n X n Matrix Access

---

## Array Elements

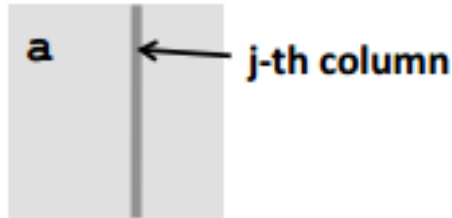
- Address  $A + i * (C*K) + j*K$
- $C = n, K = 4$

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

```
movl 8(%ebp), %eax    # n  
sall $2, %eax        # n*4  
movl %eax, %edx      # n*4  
imull 16(%ebp), %edx # i*n*4  
movl 20(%ebp), %eax  # j  
sall $2, %eax        # j*4  
addl 12(%ebp), %eax  # a + j*4  
movl (%eax,%edx), %eax # *(a + j*4 + i*n*4)
```

# Optimizing Fixed Array Access

---



## Computation

- Step through all elements in column j

## Optimization

- Retrieving successive elements from single column

```
#define N 16
typedef int fix_matrix[N][N]

/* Retrieve column j from array */
void fix_column
    (fix_matrix a, int j, int *dest)
{
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
```



# Optimizing Fixed Array Access

## Optimization

- Compute  $ajp = a[i][j]$
- Initially  $= a + 4*j$
- Increment by  $4*N$

Register	Value
%ecx	ajp
%ebx	dest
%edx	i

```
/* Retrieve column j from array */  
void fix_column  
    (fix_matrix a, int j, int *dest)  
{  
    int i;  
    for (i = 0; i < N; i++)  
        dest[i] = a[i][j];  
}
```

```
.L8:                # loop:  
    movl (%ecx), %eax    # Read *ajp  
    movl %eax, (%ebx,%edx,4) # Save in dest[i]  
    addl $1, %edx        # i++  
    addl $64, %ecx       # ajp += 4*N  
    cmpl $16, %edx      # i:N  
    jne .L8             # if !=, goto loop
```

# Optimizing Variable Array Access

Compute  $ajp = a[i][j]$

- Initially =  $a + 4*j$
- Increment by  $4*n$

Register	Value
%ecx	ajp
%edi	dest
%edx	i
%ebx	$4*n$
%esi	n

```
/* Retrieve column j from array */
void var_column
    (int n, int a[n][n],
     int j, int *dest)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = a[i][j];
}
```

```
.L18:                                # loop:
    movl (%ecx), %eax                # Read *ajp
    movl %eax, (%edi,%edx,4)         # Save in dest[i]
    addl $1, %edx                    # i++
    addl %ebx, %ecx                  # ajp += 4*n
    cmpl $edx, %esi                 # n:i
    jg .L18                          # if >, goto loop
```

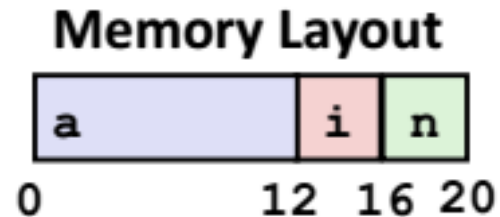
# Structures

---

# Structure Allocation

---

```
struct rec {  
    int a[3];  
    int i;  
    struct rec *n;  
};
```

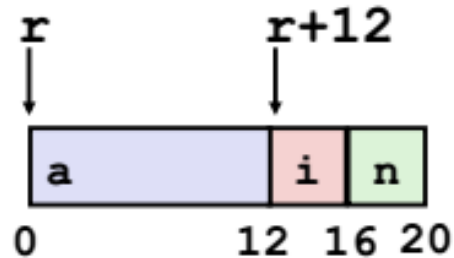


## Concept

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

# Structure Access

```
struct rec {  
    int a[3];  
    int i;  
    struct rec *n;  
};
```



## Accessing Structure Member

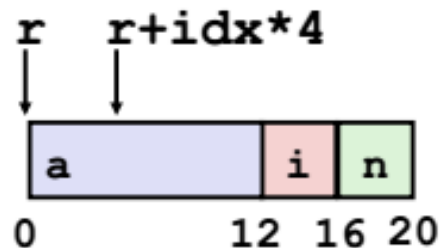
- Pointer indicates first byte of structure
- Access elements with offsets

```
void set_i(struct rec *r, int val)  
{  
    r->i = val;  
}
```

```
# IA32 Assembly  
# %edx = val  
# %eax = r  
movl %edx, 12(%eax)    # Mem[r+12] = val
```

# Generating Pointer to Structure Member

```
struct rec {  
    int a[3];  
    int i;  
    struct rec *n;  
};
```



## Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Arguments
  - Mem[%ebp+8]: r
  - Mem[%ebp+12]: idx

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

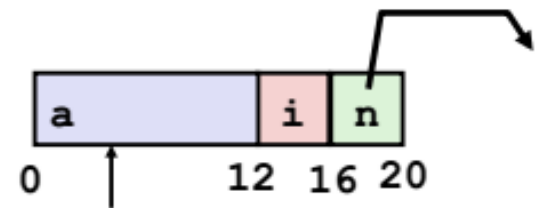
```
movl 12(%ebp), %eax # Get idx  
sall $2, %eax      # idx*4  
addl 8(%ebp), %eax # r+idx*4
```

# Following Linked List

C code

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

```
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```



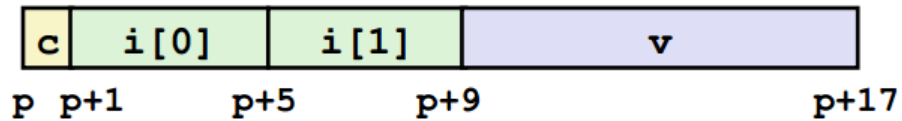
Element i

```
.L17:                # loop:
    movl 12(%edx), %eax    # r->i
    movl %ecx, (%edx,%eax,4) # r->a[i] = val
    movl 16(%edx), %edx   # r = r->n
    testl %edx, %edx     # Test r
    jne .L17             # If != 0 goto loop
```

Register	Value
%edx	r
%ecx	val

# Structures Alignment

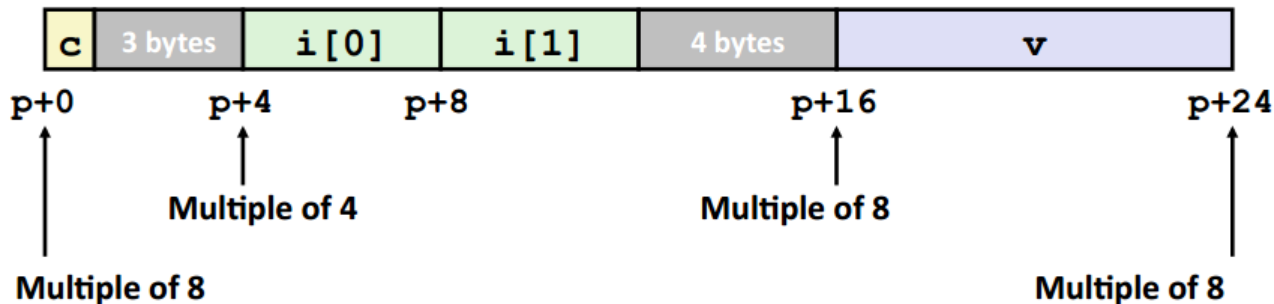
## Unaligned Data



```
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 IA32: **treated differently by IA32 Linux x86-64 Linux and Windows!**

## Motivation for Aligning Data

- Memory accessed by (aligned) chunks(数据块) of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory very tricky when datum spans 2 pages

## Compiler

- Inserts gaps in structure to ensure correct alignment of fields

# Specific Cases of Alignment (IA32)

---

1 byte: char, ...

- no restrictions on address

2 bytes: short, ...

- lowest 1 bit of address must be  $0_2$

4 bytes: int, float, char \*, ...

- lowest 2 bits of address must be  $00_2$

8 bytes: double, ...

- Windows (and most other OS's instruction sets):
  - lowest 3 bits of address must be  $000_2$
- Linux:
  - lowest 2 bits of address must be  $00_2$
  - i.e., treated the same as a 4-byte primitive data type

12 bytes: long double

- Windows, Linux:
  - lowest 2 bits of address must be  $00_2$
  - i.e., treated the same as a 4-byte primitive data type

# Specific Cases of Alignment (X86-64)

---

1 byte: char, ...

- no restrictions on address

2 bytes: short, ...

- lowest 1 bit of address must be  $0_2$

4 bytes: int, float, ...

- lowest 2 bits of address must be  $00_2$

8 bytes: double, char \*, ...

- Windows Linux
  - lowest 3 bits of address must be  $000_2$

16 bytes: long double

- Linux:
  - lowest 3 bits of address must be  $000_2$
  - i.e., treated the same as a 8-byte primitive data type

# Satisfying Alignment with Structures

Within structure:

- Must satisfy each element's alignment requirement

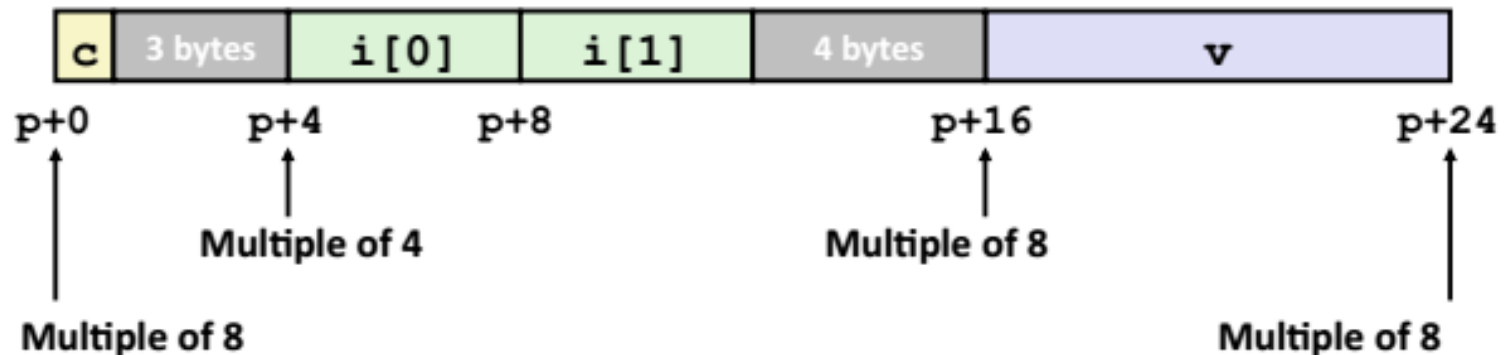
Overall structure placement

- Each structure has alignment requirement  $K$
- $K =$  Largest alignment of any element
- Initial address & structure length must be multiples of  $K$

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

Example (under Windows or x86-64):

- $K = 8$ , due to double element

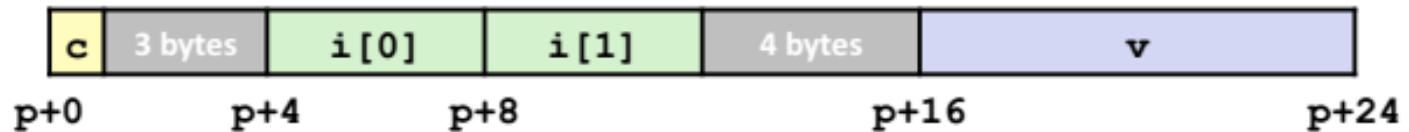


# Different Alignment Conventions

---

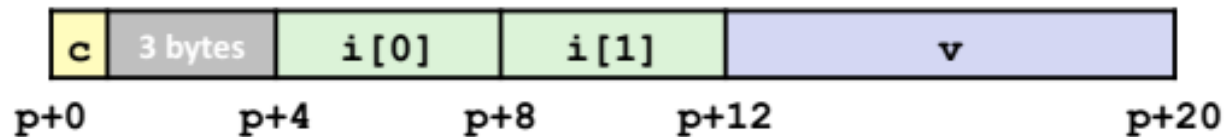
## X86-64 or IA32 Windows

- K=8, due to double element



## IA32 Linux

- K=4, double treated like a 4-byte data type



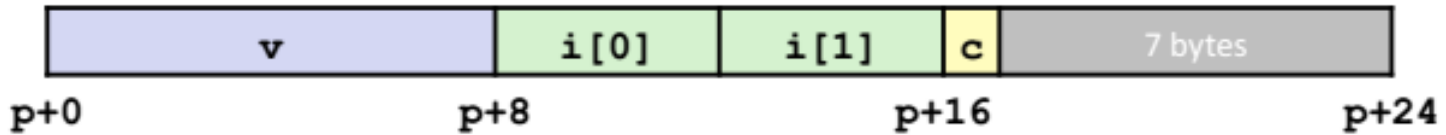
# Meeting Overall Alignment Requirement

---

For largest alignment requirement K

Overall structure must be multiple of K

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

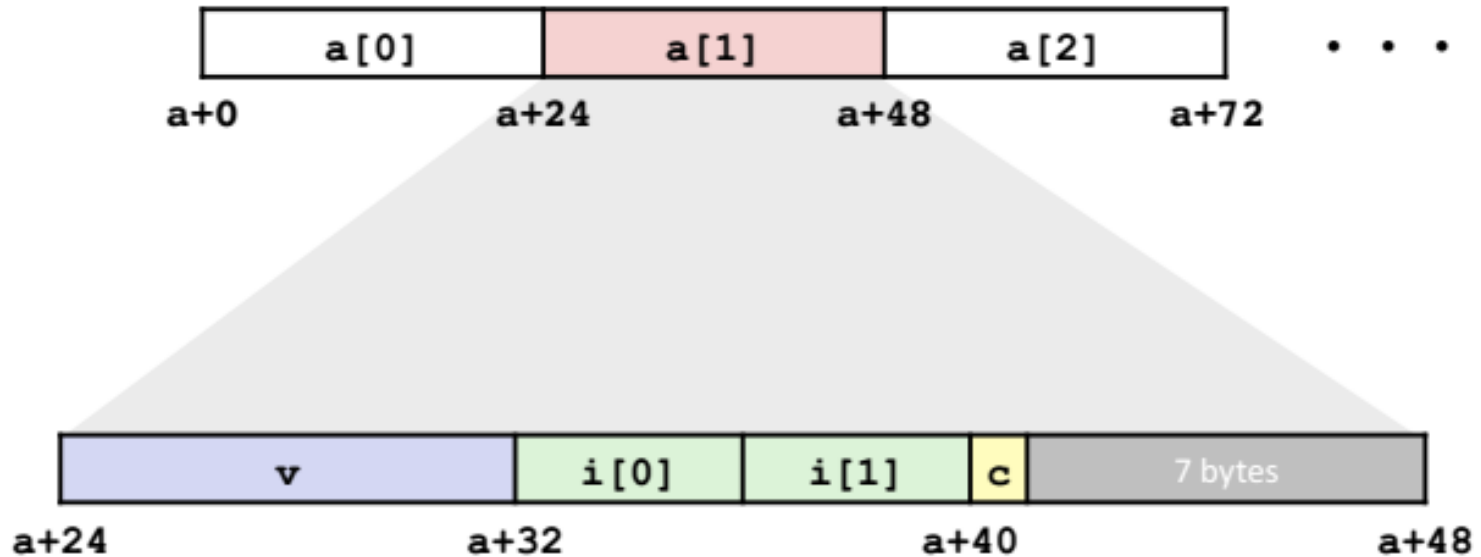


# Arrays of Structures

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  $12i$

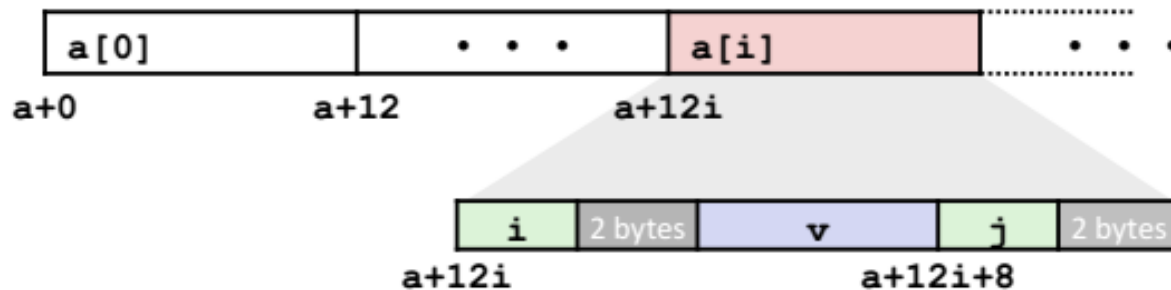
- `sizeof(S3)` including alignment spacers

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;  
}
```

```
# %eax = idx  
leal    (%eax,%eax,2),%eax    # 3*idx  
movswl  a+8(,%eax,4),%eax
```



# Saving Space

---

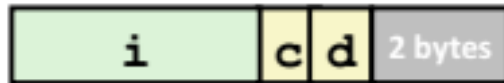
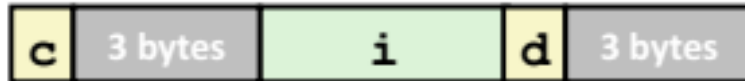
Put large data types first

```
struct S3 {  
    char c;  
    int i;  
    char d;  
} a[10];
```



```
struct S5 {  
    int i;  
    char c;  
    char d;  
} a[10];
```

Effect (K=4)



Union

---

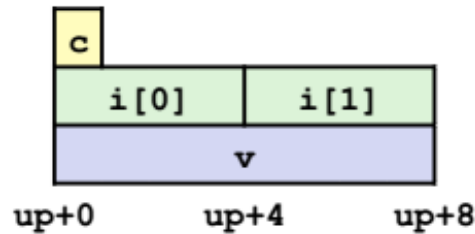
# Union Allocation

---

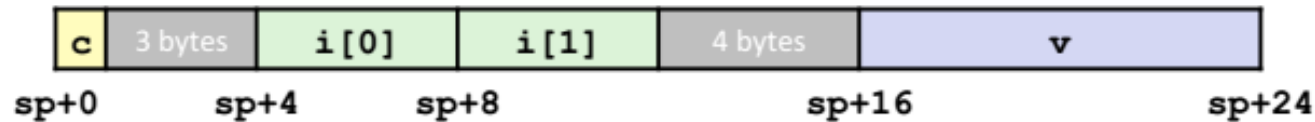
Allocate according to largest element

Can only use one field at a time

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

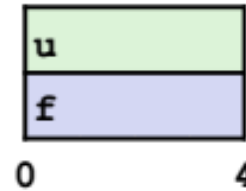


```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *sp;
```



# 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;  
}
```

Same as (float) u?

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

Same as (unsigned) f?

No !

# Byte Ordering Revisited

---

## Idea

- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which 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

## Bi Endian

- Can be configured either way
- ARM

# Byte Ordering Example

---

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

**32-bit**

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]							

**64-bit**

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]							

# 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]);

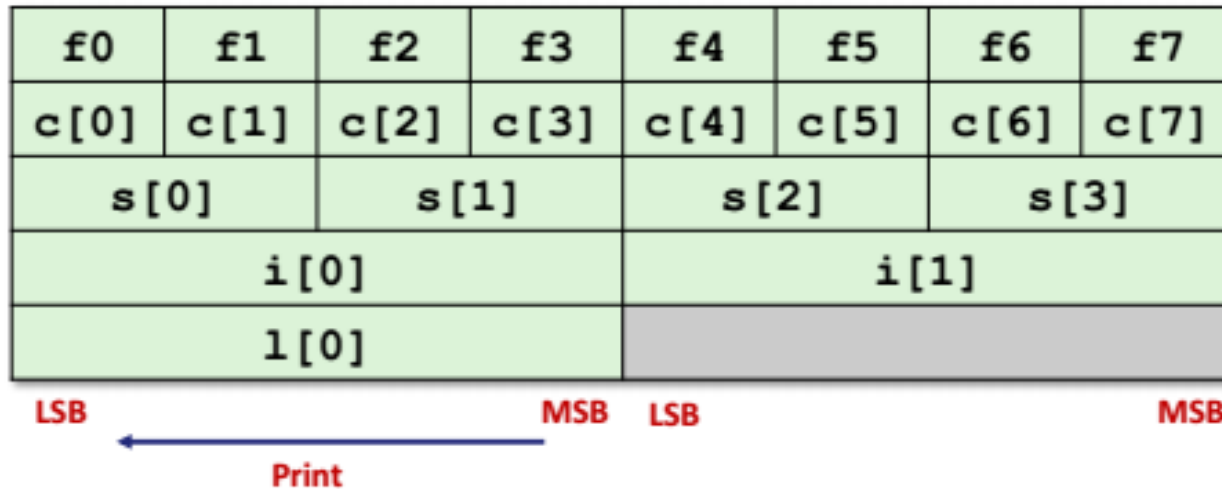
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

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

## Little Endian



## Output

Characters      0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]

Shorts          0-3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]

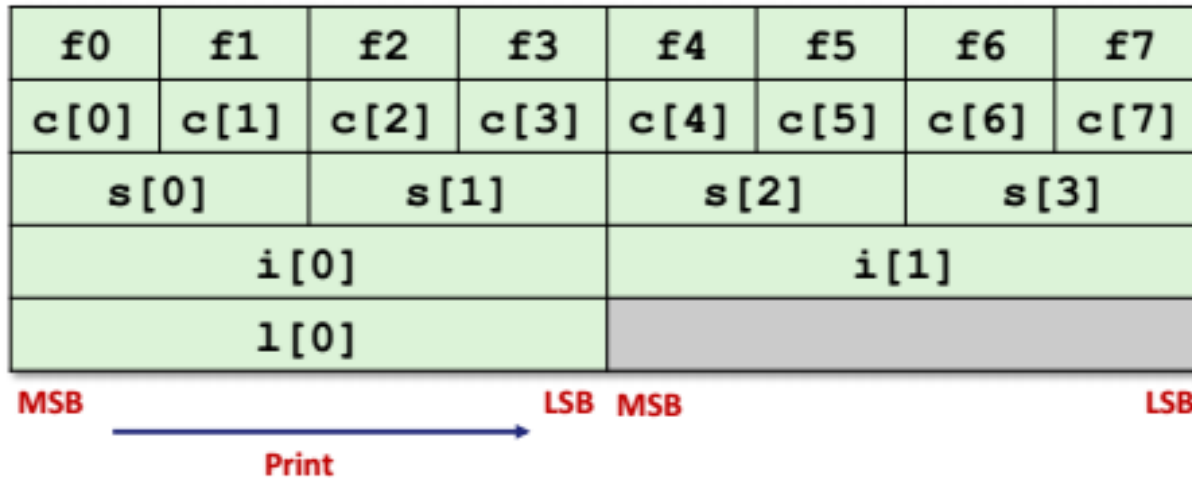
Ints            0-1 == [0xf3f2f1f0, 0xf7f6f5f4]

Long            0    == [0xf3f2f1f0]



# Byte Ordering on Sun

Big Endian



Output on Sun :

Characters      0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]

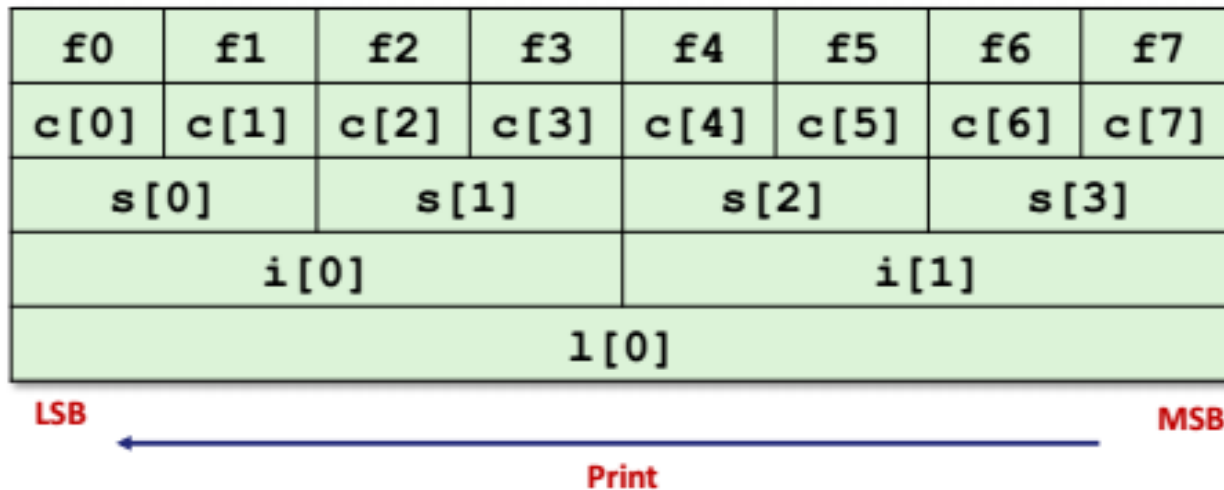
Shorts            0-3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]

Ints              0-1 == [0xf0f1f2f3, 0xf4f5f6f7]

Long              0    == [0xf0f1f2f3]

# Byte Ordering on x86-64

## Little Endian



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]

# Summary

---

## Arrays in C

- Contiguous allocation of memory
- Aligned to satisfy every element's alignment requirement
- Pointer to first element
- No bounds checking

## Structures

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

## Unions

- Overlay declarations
- Way to circumvent(绕过) type system