



第3章 程序的机器级表示

Machine-Level Programming IV: Data

100076202: 计算机系统导论
IV: 数据
IV: Data



任课教师:

计卫星 宿红毅 张艳

原作者:

Randal E. Bryant and David R. O'Hallaron

**Carnegie
Mellon
University**



议题

■ 数组 Arrays

- 一维 One-dimensional
- 多维（嵌套） Multi-dimensional (nested)
- 多级 Multi-level

■ 结构 Structures

- 分配 Allocation
- 访问 Access
- 对齐 Alignment

■ 浮点数 Floating Point

提醒：内存组织



Reminder: Memory Organization

■ 内存位置没有数据类型

Memory locations do not have data types

- 类型隐含在机器指令如何使用内存
Types are implicit in how machine instructions *use* memory

■ 地址指定字节位置

Addresses specify byte locations

- 大型数据地址是其第一字节的地址
Address of a larger datum is the address of its first byte
- 后继项目的地址根据项目大小而不同
Addresses of successive

Address	chars	ints	longs
4000			
4001		Addr =	
4002		4000	
4003			Addr =
4004			4000
4005		Addr =	
4006		4004	
4007			
4008			
4009		Addr =	
400A		4008	
400B			Addr =
400C			4008
400D		Addr =	
400E		400C	
400F			

数组分配 Array Allocation

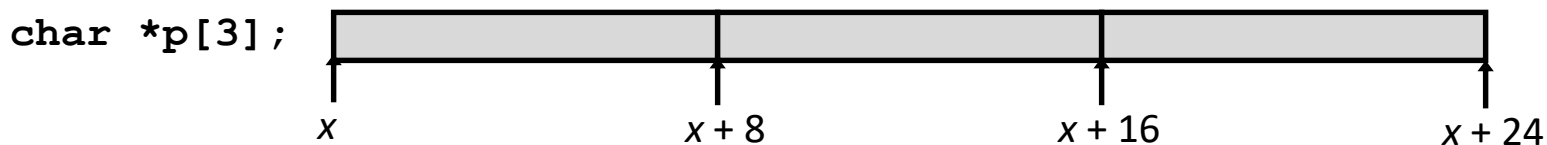
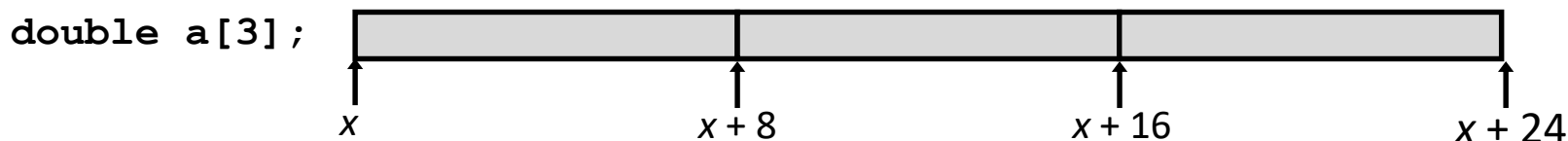
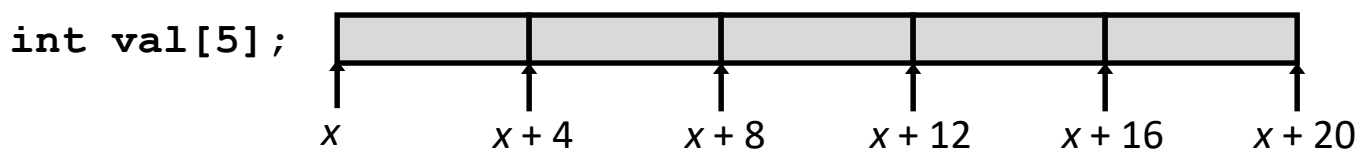
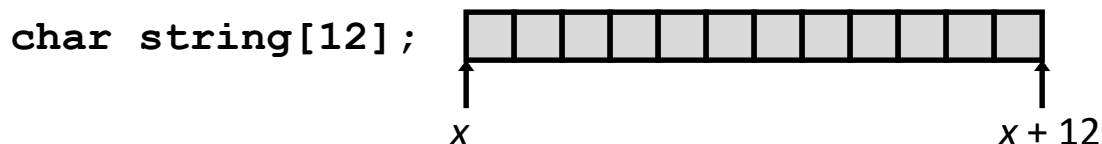


■ 基本原则 Basic Principle

$T \ A[L];$

■ 数组的数据类型为T且长度为L Array of data type T and length L

■ 在内存中分配 $L * \text{sizeof}(T)$ 字节的连续区域 Contiguously allocated region of $L * \text{sizeof}(T)$ bytes in memory



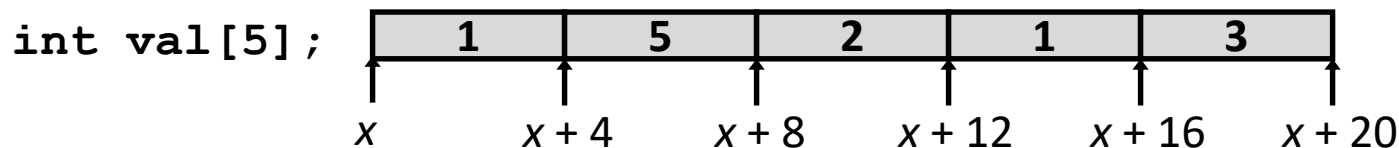
数组访问 Array Access



■ 基本原则 Basic Principle

T **A**[L];

- 数组的数据类型为 T 且长度为 L Array of data type T and length L
- 标识符 A 可以用作指向数组第0个元素的指针 Identifier **A** can be used as a pointer to array element 0: Type T^*



■ 引用Reference 类型Type 值Value

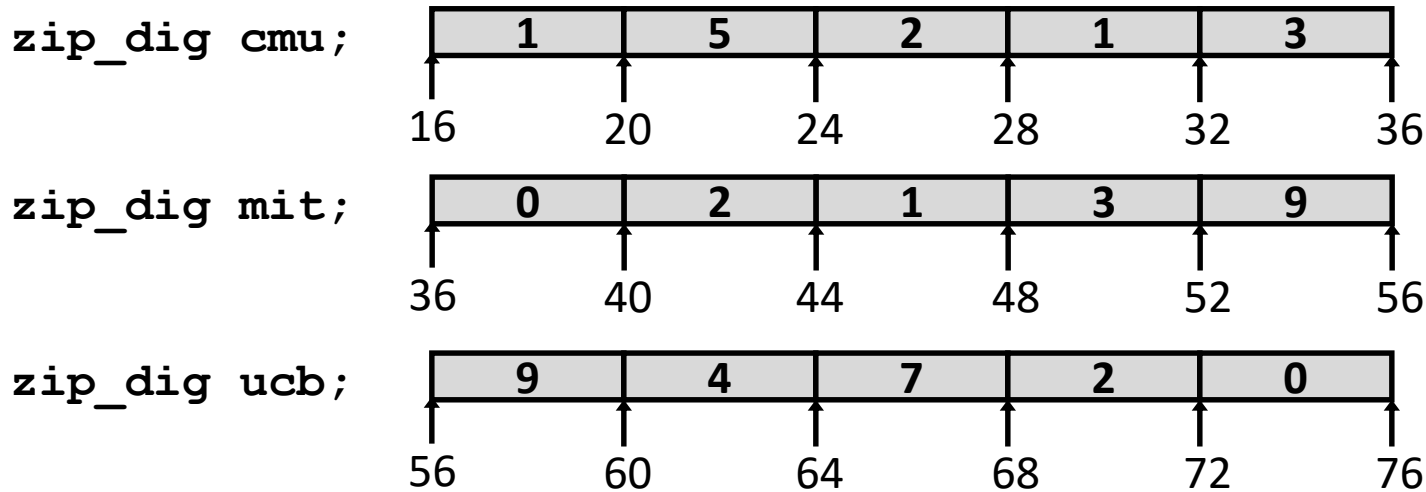
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	x
<code>val+1</code>	<code>int *</code>	$x + 4$
<code>&val[2]</code>	<code>int *</code>	$x + 8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	$x + 4i$



数组示例 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 };
```

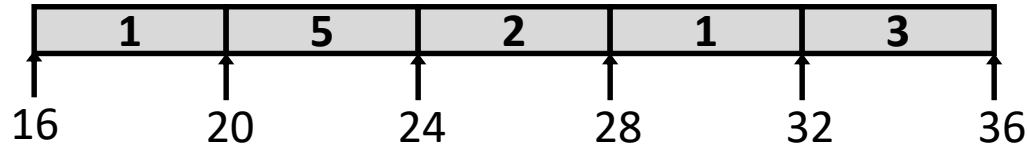


- 声明zip_dig cmu等价于int cmu[5] Declaration “zip_dig cmu” equivalent to “int cmu[5]”
- 示例数组以连续的20字节块进行分配 Example arrays were allocated in successive 20 byte blocks
 - 一般并不确保这样 Not guaranteed to happen in general

数组访问示例 Array Accessing Example



zip_dig cmu;



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

x86-64

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

- 寄存器%rdi包含数组起始地址 Register %rdi contains starting address of array
- 寄存器%rsi包含数组索引 Register %rsi contains array index
- 期望的数值在 Desired digit at $\%rdi + 4 * \%rsi$
- 使用内存引用 Use memory reference $(\%rdi, \%rsi, 4)$



数组循环示例 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
```


理解指针和数组 #1

Understanding Pointers & Arrays #1



Decl	<i>An</i>			<i>*An</i>		
	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>						
<code>int *A2</code>						

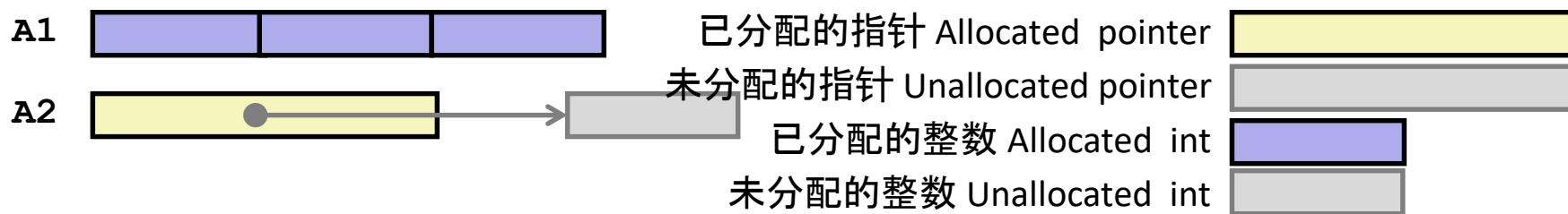
- Cmp: Compiles (Y/N) 编译时
- Bad: Possible bad pointer reference (Y/N) 可能错误指针引用
- Size: Value returned by `sizeof` `sizeof`的返回值

理解指针和数组 #1

Understanding Pointers & Arrays #1



Decl	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]	Y	N	12	Y	N	4
int *A2	Y	N	8	Y	Y	4



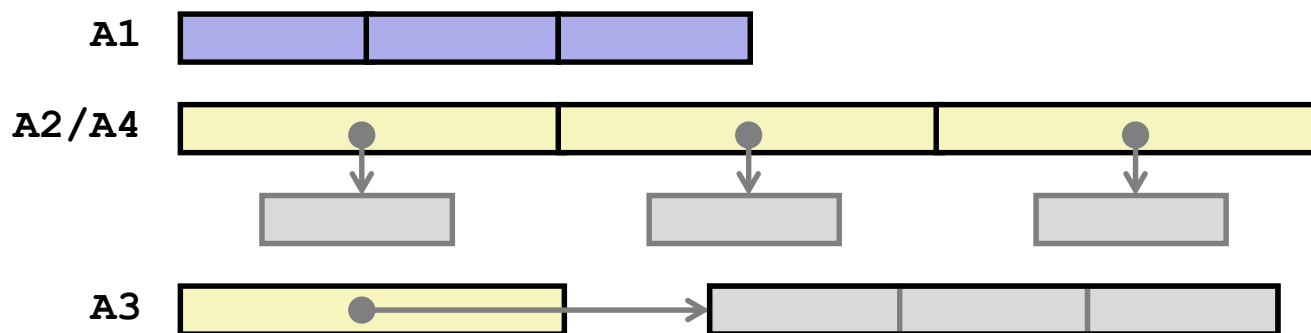
- **Cmp: Compiles (Y/N) 编译时**
- **Bad: Possible bad pointer reference (Y/N) 可能错误指针引用**
- **Size: Value returned by sizeof sizeof返回值**

理解指针和数组 #2



Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]	Y	N	12	Y	N	4	N	-	-
int *A2[3]	Y	N	24	Y	N	8	Y	Y	4
int (*A3)[3]	Y	N	8	Y	Y	12	Y	Y	4
int (*A4[3])	Y	N	24	Y	N	8	Y	Y	4

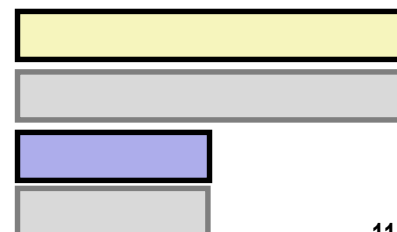


已分配的指针 Allocated pointer

未分配的指针 Unallocated pointer

已分配的整数 Allocated int

未分配的整数 Unallocated int



理解指针和数组 #3

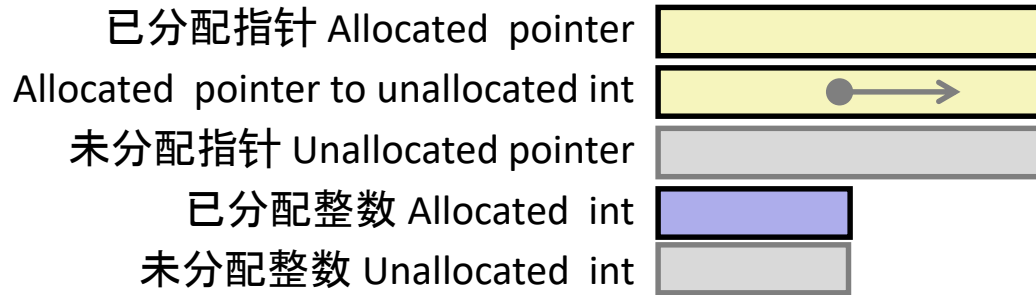


Understanding Pointers & Arrays #3

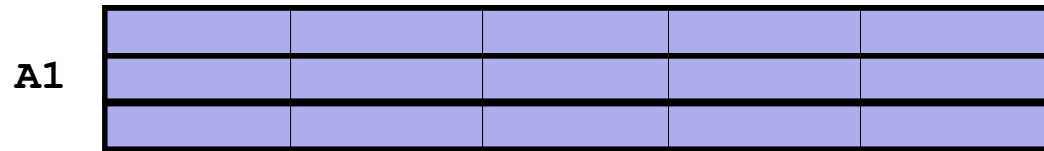
Decl	An			*An			**An		
	Cm p	Bad	Size	Cm p	Bad	Size	Cm p	Bad	Size
<code>int A1[3][5]</code>									
<code>int *A2[3][5]</code>									
<code>int (*A3)[3][5]</code>									
<code>int *(A4[3][5])</code>									
<code>int (*A5[3])[5]</code>									

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

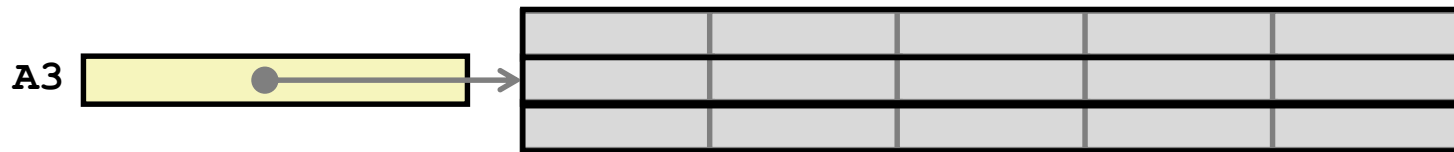
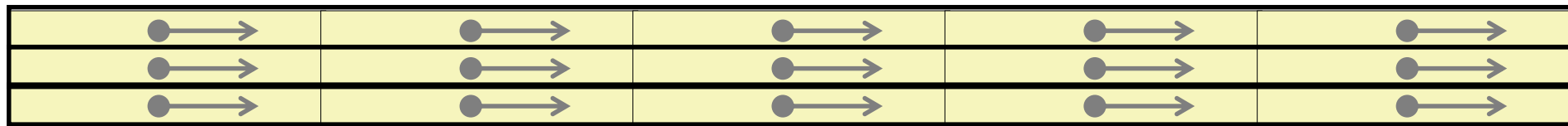
Decl	***An		
	Cm p	Bad	Size
<code>int A1[3][5]</code>			
<code>int *A2[3][5]</code>			
<code>int (*A3)[3][5]</code>			
<code>int *(A4[3][5])</code>			
<code>int (*A5[3])[5]</code>			



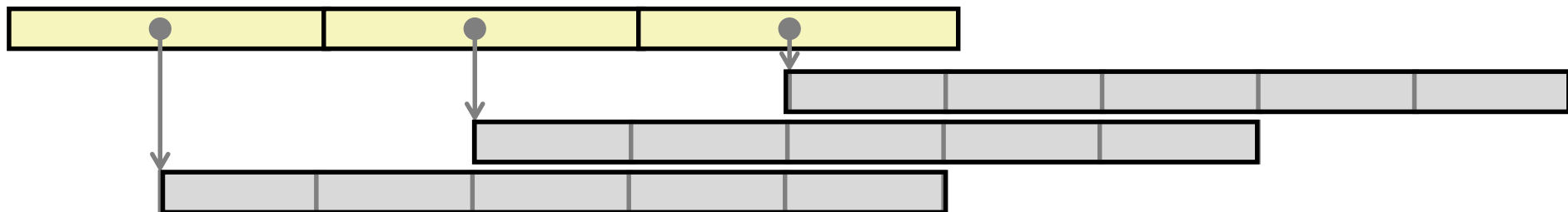
Declaration
<code>int A1[3][5]</code>
<code>int *A2[3][5]</code>
<code>int (*A3)[3][5]</code>
<code>int *(A4[3][5])</code>
<code>int (*A5[3])[5]</code>



A2/A4



A5



理解指针和数组 #3



Understanding Pointers & Arrays #3

Decl	An			*An			**An		
	Cm p	Bad	Size	Cm p	Bad	Size	Cm p	Bad	Size
<code>int A1[3][5]</code>	Y	N	60	Y	N	20	Y	N	4
<code>int *A2[3][5]</code>	Y	N	120	Y	N	40	Y	N	8
<code>int (*A3)[3][5]</code>	Y	N	8	Y	Y	60	Y	Y	20
<code>int *(A4[3][5])</code>	Y	N	120	Y	N	40	Y	N	8
<code>int (*A5[3])[5]</code>	Y	N	24	Y	N	8	Y	Y	20

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Decl	***An		
	Cm p	Bad	Size
<code>int A1[3][5]</code>	N	-	-
<code>int *A2[3][5]</code>	Y	Y	4
<code>int (*A3)[3][5]</code>	Y	Y	4
<code>int *(A4[3][5])</code>	Y	Y	4
<code>int (*A5[3])[5]</code>	Y	Y	4

多维（嵌套）数组

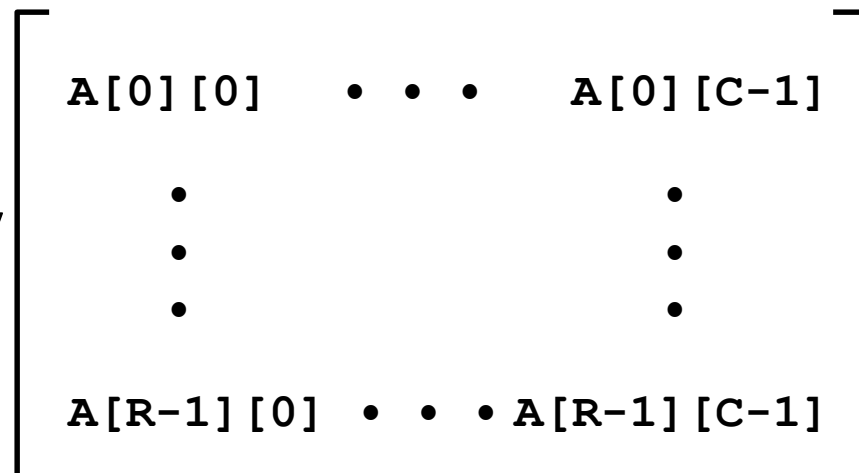


Multidimensional (Nested) Arrays

■ 声明 Declaration

$T \ A[R][C];$

- 数据类型为T的二维数组 2D array of data type T
- R行C列 R rows, C columns
- 类型T元素需要K个字节 Type T element requires K bytes

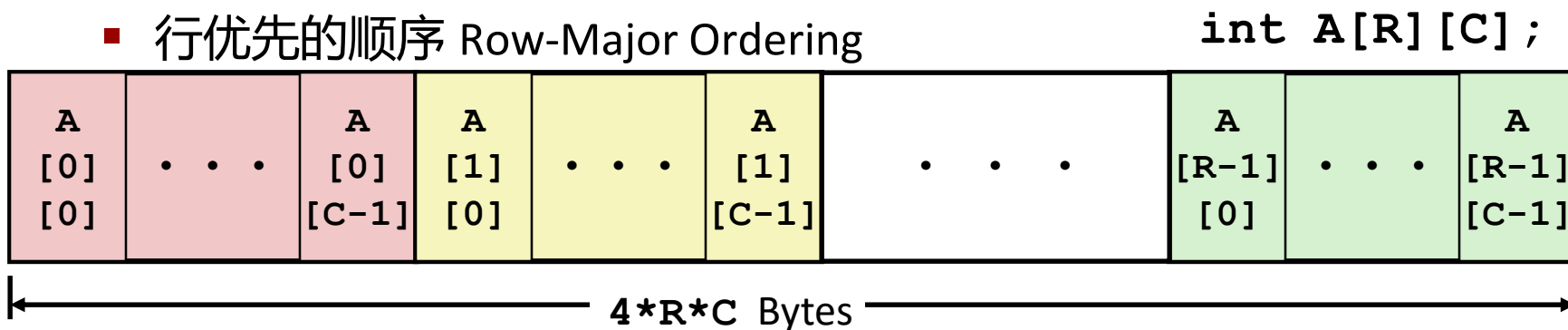


■ 数组大小 Array Size

- $R * C * K$ bytes字节

■ 排列 Arrangement

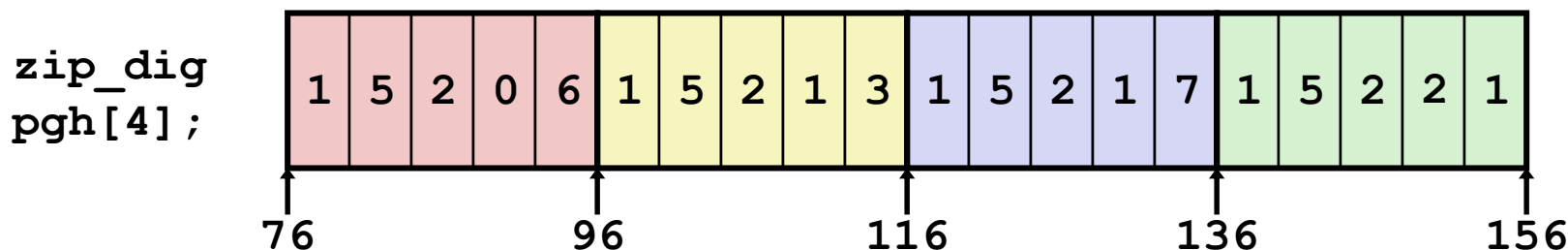
- 行优先的顺序 Row-Major Ordering





嵌套数组示例 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]” 等价于equivalent to “int pgh[4][5]”
 - 变量pgh: 四个元素的数组, 连续分配 Variable **pgh**: array of 4 elements, allocated contiguously
 - 每个元素是五个整数的数组, 连续分配 Each element is an array of 5 **int**'s, allocated contiguously
- 在内存中所有元素都采用 “行优先” 顺序排列 “Row-Major” ordering of all elements in memory

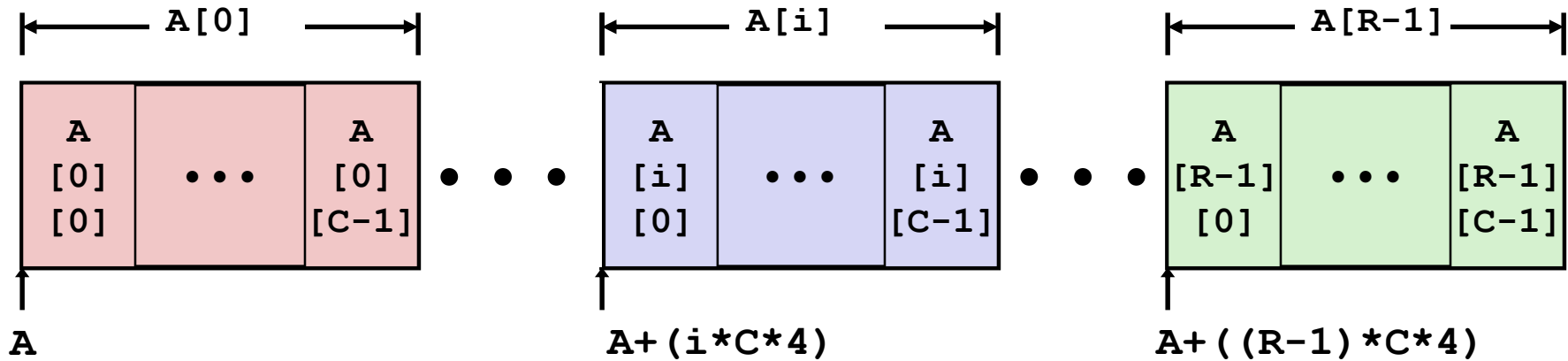
嵌套数组行访问 Nested Array Row Access



■ 行向量 Row Vectors

- $A[i]$ 是C个元素的数组 $A[i]$ is array of C elements
- 每个类型为T的元素需要k字节 Each element of type T requires K bytes
- 起始地址为 Starting address $A + i * (C * K)$

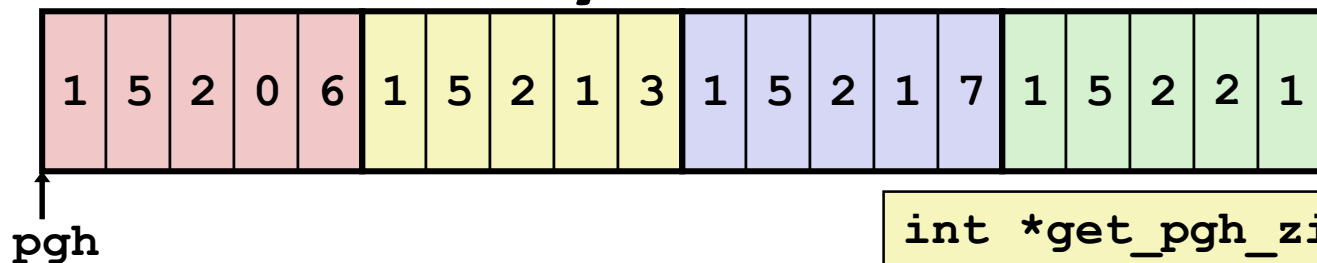
```
int A[R][C];
```



嵌套数组行访问代码



Nested Array Row Access Code



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

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

■ 行向量 Row Vector

- `pgh[index]` 是五个整数的数组 `pgh[index]` is array of 5 `int`'s
- 起始地址为 Starting address `pgh+20*index`

■ 机器代码 Machine Code

- 计算和返回地址 Computes and returns address
- 计算方法 Compute as `pgh + 4*(index+4*index)`

嵌套数组元素访问

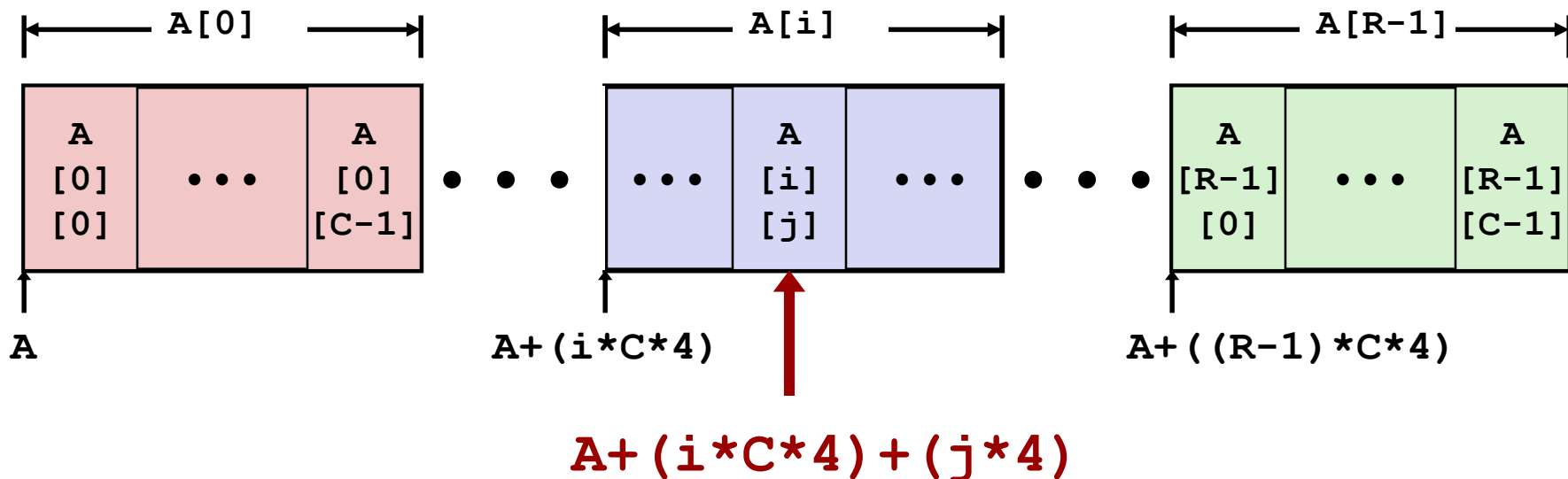


Nested Array Element Access

■ 数组元素 Array Elements

- $A[i][j]$ 是类型为 T 的元素, 需要 K 字节 $A[i][j]$ is element of type T , which requires K bytes
- 地址为 Address $A + i * (C * K) + j * K = A + (i * C + j) * K$

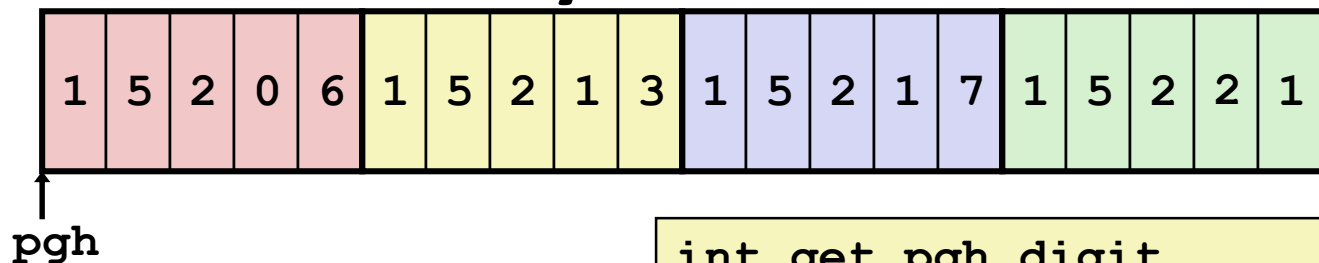
```
int A[R][C];
```



嵌套数组元素访问代码



Nested Array Element Access Code



```
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)]
```

■ 数组元素 Array Elements

- 数组元素是整数 $\text{pgh}[\text{index}][\text{dig}]$ is int
- 地址: Address: $\text{pgh} + 20 * \text{index} + 4 * \text{dig}$
 - $= \text{pgh} + 4 * (5 * \text{index} + \text{dig})$

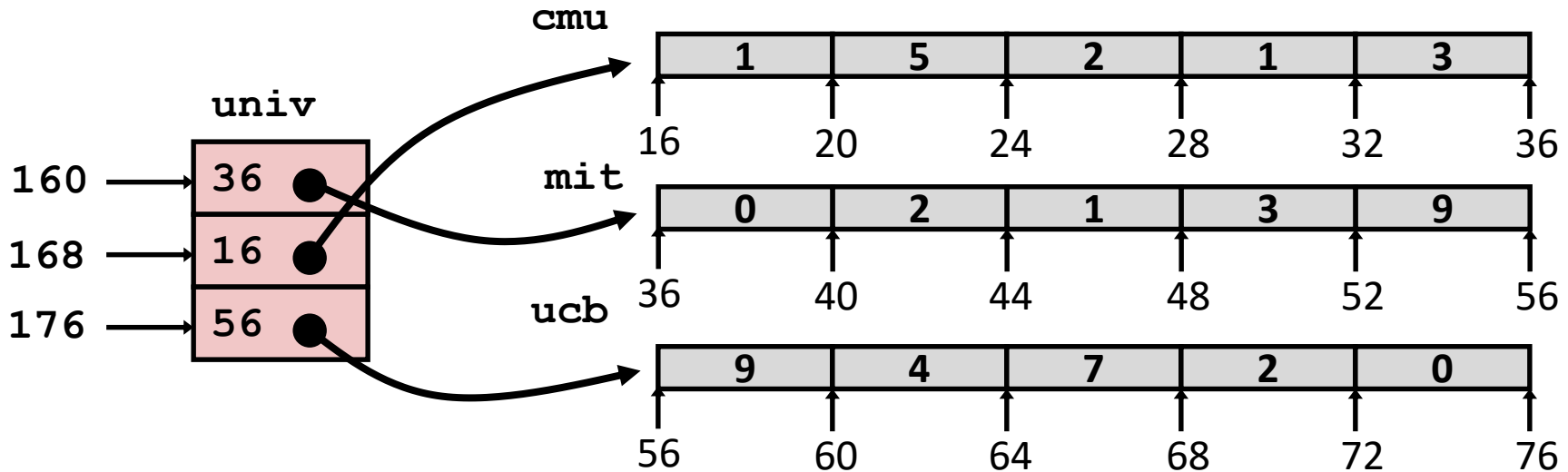
多级数组示例 Multi-Level Array Example



```
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};
```

- 变量univ指向有3个元素的数组 Variable univ denotes array of 3 elements
- 每个元素是一个指针 Each element is a pointer
 - 8 bytes字节
- 每个指针指向整数数组 Each pointer points to array of int's

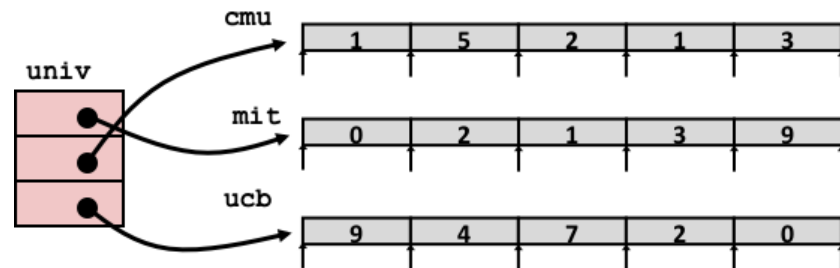


在多级数组中访问元素

Element Access in Multi-Level 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
```

■ 计算 Computation

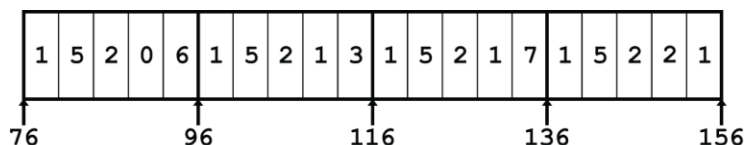
- 元素访问 Element access $\text{Mem}[\text{Mem}[\text{univ} + 8 * \text{index}] + 4 * \text{digit}]$
- 必须进行两次内存读操作 Must do two memory reads
 - 第一次得到行数组的指针 First get pointer to row array
 - 然后访问数组内元素 Then access element within array

数组元素访问 Array Element Accesses



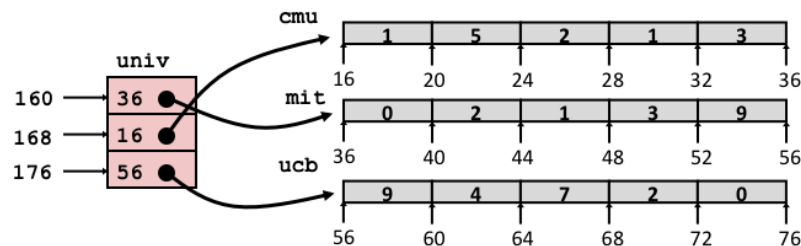
嵌套数组 Nested array

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



多级数组 Multi-level array

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



在C语言中访问看起来是类似的，但是地址计算方法非常不同

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

$\text{Mem}[\text{pgh} + 20 * \text{index} + 4 * \text{digit}]$ $\text{Mem}[\text{Mem}[\text{univ} + 8 * \text{index}] + 4 * \text{digit}]$



二维矩阵代码

N X N Matrix Code

■ 固定维数 Fixed dimensions

- 编译时知道N的值 Know value of N at compile time

```
#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];
}
```

■ 变化维数，显式索引 Variable dimensions, explicit indexing

- 传统方法实现动态数组 Traditional way to implement dynamic arrays

```
#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)];
}
```

■ 变化维数，隐式索引 Variable dimensions, implicit indexing

- 目前gcc支持 Now supported by gcc

```
/* 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];
}
```


16X16矩阵访问 16 X 16 Matrix Access



■ 数组元素 Array Elements

- 地址为 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 矩阵访问 n X n Matrix Access

■ 数组元素 Array Elements

- 地址为 $\text{Address } A + i * (C * K) + j * K$
- $C = n, K = 4$
- 必须执行整数乘法 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
```

示例：数组访问 Example: Array Access



```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    zip_dig pgh[PCOUNT] =
        {{1, 5, 2, 0, 6},
         {1, 5, 2, 1, 3 },
         {1, 5, 2, 1, 7 },
         {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
```

示例：数组访问 Example: Array Access



```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    zip_dig pgh[PCOUNT] =
        {{1, 5, 2, 0, 6},
         {1, 5, 2, 1, 3 },
         {1, 5, 2, 1, 7 },
         {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
result: 9
```



议题

■ 数组 Arrays

- 一维 One-dimensional
- 多维(嵌套) Multi-dimensional (nested)
- 多级 Multi-level

■ 结构 Structures

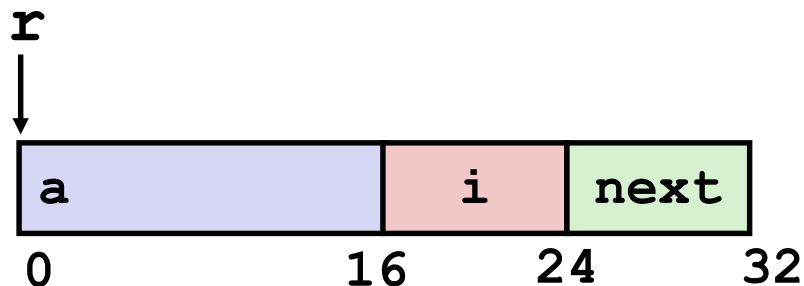
- 分配 Allocation
- 访问 Access
- 对齐 Alignment

■ 浮点数 Floating Point

结构表示 Structure Representation



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



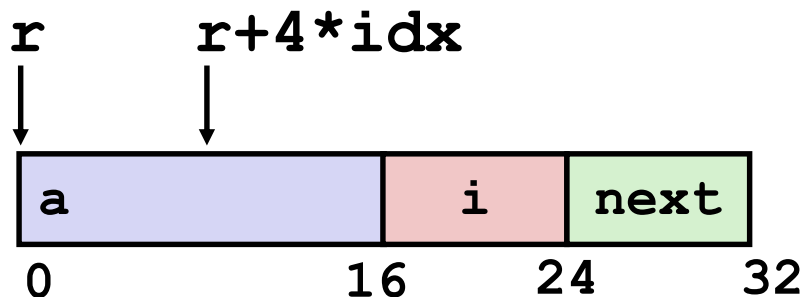
- **结构表示为内存块** Structure represented as block of memory
 - 大到足够装下所有字段 Big enough to hold all of the fields
- **字段顺序按照声明顺序** Fields ordered according to declaration
 - 即使另一种顺序可能还能够更加紧凑进行表示 Even if another ordering could yield a more compact representation
- **编译器决定总体大小+字段位置** Compiler determines overall size + positions of fields
 - 机器级程序并不理解源代码中的结构 Machine-level program has no

生成结构成员的指针



Generating Pointer to Structure Member

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



■ 生成数组元素的指针 Generating Pointer to Array Element

- 每个结构成员的偏移在编译时确定 Offset of each structure member determined at compile time
- 计算方法 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
```

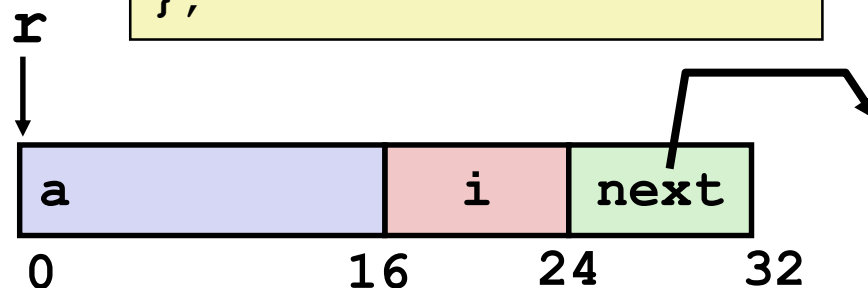
遍历链表#1

Following Linked List #1

■ C代码 C Code

```
long length(struct rec *r) {  
    long len = 0L;  
    while (r) {  
        len ++;  
        r = r->next;  
    }  
    return len;  
}
```

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



寄存器 Register	值 Value
%rdi	r
%rax	len

■ 循环汇编代码 Loop assembly code

```
.L11:                                # loop:  
    addq    $1, %rax                # len ++  
    movq    24(%rdi), %rdi          # r = Mem[r+24]  
    testq   %rdi, %rdi              # Test r  
    jne     .L11                    # If != 0, goto loop
```

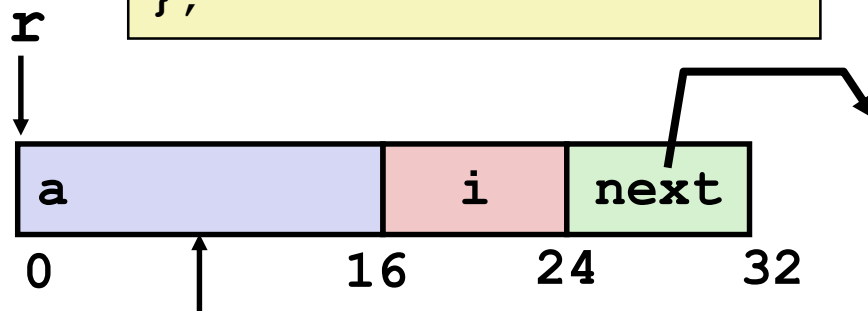
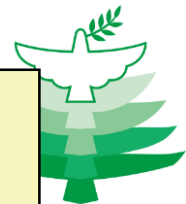

遍历链表#2

Following Linked List #2

■ C代码 C Code

```
void set_val
(struct rec *r, int val)
{
    while (r) {
        size_t i = r->i;
        // No bounds check
        r->a[i] = val;
        r = r->next;
    }
}
```

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



元素 Element i

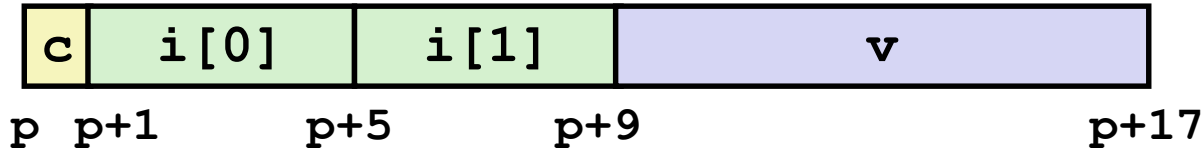
寄存器 Register	值 Value
%rdi	r
%rsi	val

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

结构和对齐 Structures & Alignment



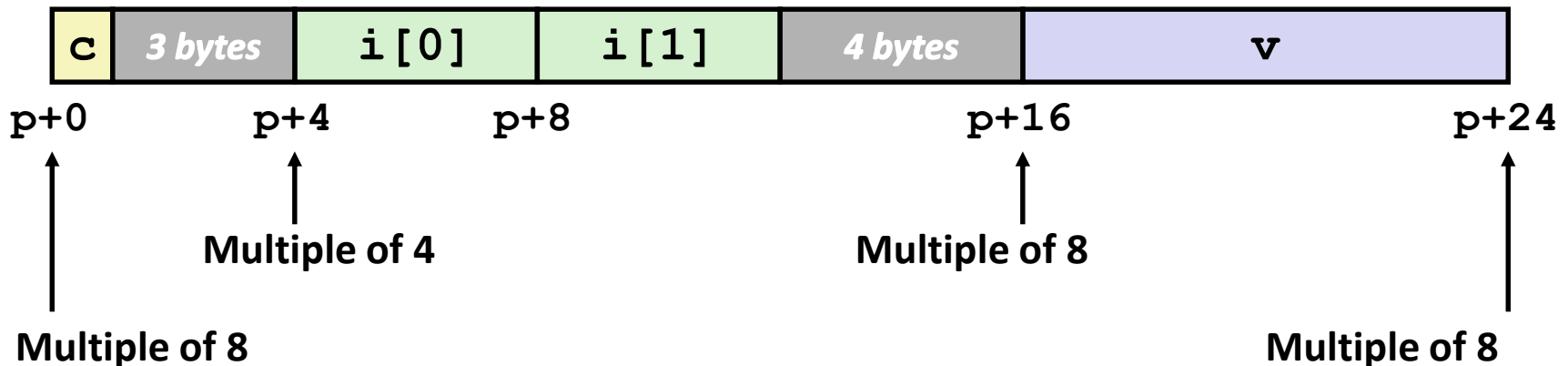
■ 没对齐的数据 Unaligned Data



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

■ 对齐的数据 Aligned Data

- 基本数据类型需要K字节 Primitive data type requires K bytes
- 地址必须是K的整数倍 Address must be multiple of K



对齐的原则 Alignment Principles



■ 对齐的数据 Aligned Data

- 基本数据类型需要K字节 Primitive data type requires *K* bytes
- 地址必须是K的整数倍 Address must be multiple of *K*
- 在某些机器上严格满足该要求；在x86-64上建议满足 Required on some machines; advised on x86-64

■ 对齐数据的动机 Motivation for Aligning Data

- 内存访问以（对齐的）4或8字节数据块（依赖于不同的系统）为单位 Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
 - 装载或存储数据比较低效，因为这样会跨越四字边界 Inefficient to load or store datum that spans quad word boundaries
 - 当数据跨越2页时虚拟存储器访问更棘手 Virtual memory trickier when datum spans 2 pages

■ 编译器 Compiler

- 在结构中插入间隔以确保字段的正确对齐 Inserts gaps in structure to ensure correct alignment of fields

对齐的特殊情况 (x86-64)

Specific Cases of Alignment (x86-64)



- **1 byte: char, ...**
 - 地址没有限制 no restrictions on address
- **2 bytes: short, ...**
 - 地址的最低一位必须为零 lowest 1 bit of address must be 0_2
- **4 bytes: int, float, ...**
 - 地址的最低二位必须为00 lowest 2 bits of address must be 00_2
- **8 bytes: double, long, char *, ...**
 - 地址的最低三位必须为000 lowest 3 bits of address must be 000_2
- **16 bytes: long double (GCC on Linux)**
 - 地址的最低四位必须为0000 lowest 4 bits of address must be 0000_2

结构体的对齐要求

Satisfying Alignment with Structures

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

■ 结构内部 Within structure:

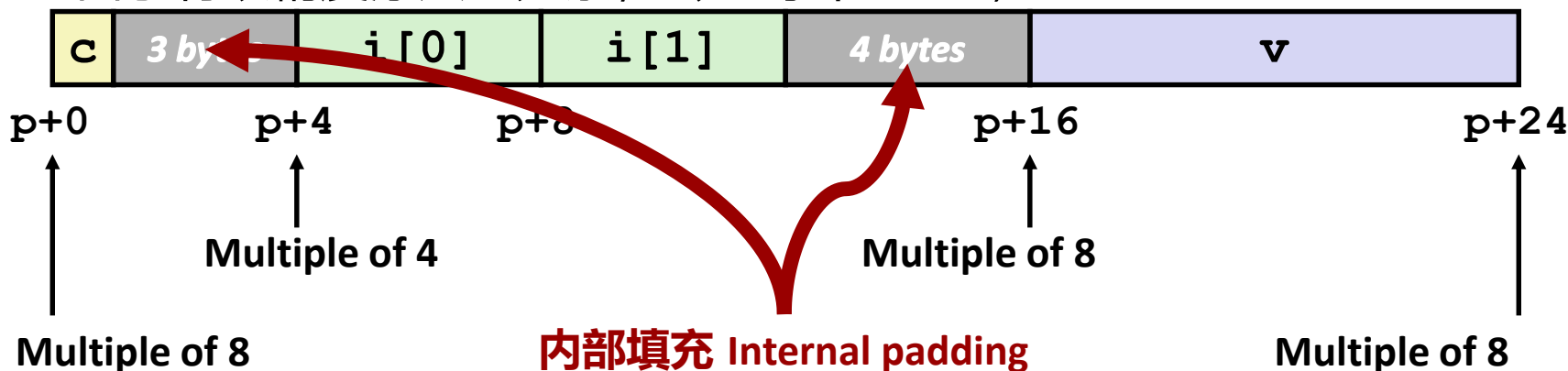
- 必须满足每个元素的对齐需求 Must satisfy each element's alignment requirement

■ 整个结构的排放 Overall structure placement

- 每个结构有对齐需求K Each structure has alignment requirement K
 - K为任何元素对齐要求的最大值 K = Largest alignment of any element
- 起始地址和结构长度必须为K的整数倍 Initial address & structure length must be multiples of K

■ 示例 Example:

- 由于有双精度浮点型元素, K为8字节 K = 8, due to **double** element



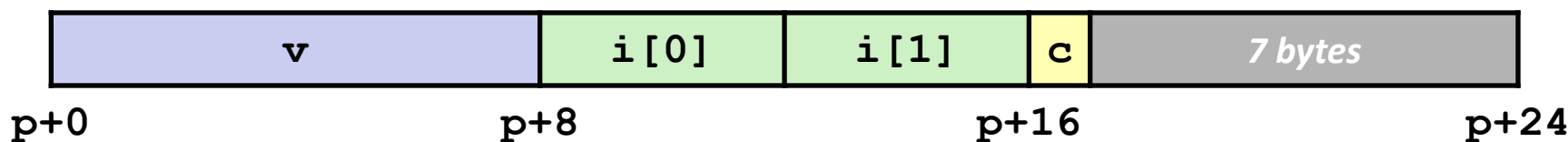
满足整体对齐需求

Meeting Overall Alignment Requirement



- 对于最大对齐需求K For largest alignment requirement K
- 整体结构必须是K的整数倍 Overall structure must be multiple of K

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



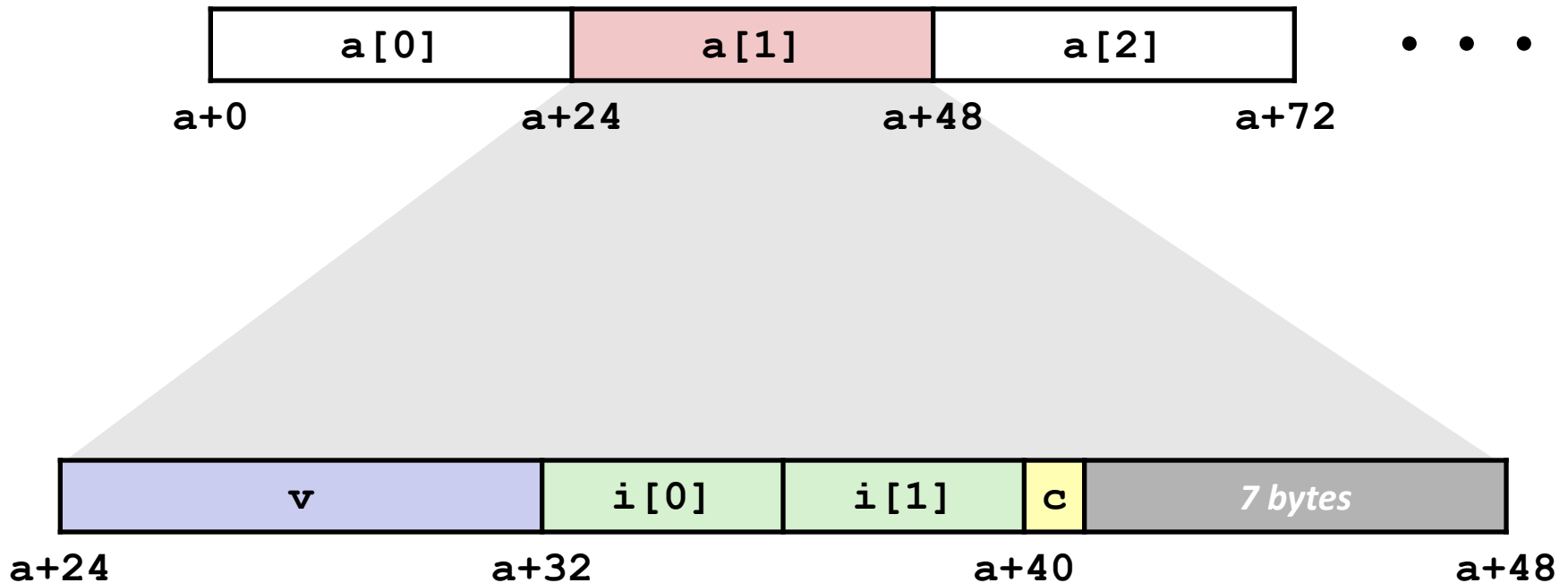
多个8字节 Multiple of K=8

结构数组 Arrays of Structures



- 整体结构长度是K的整数倍
Overall structure length multiple of K
- 满足每个元素的对齐需求 Satisfy alignment requirement for every element

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

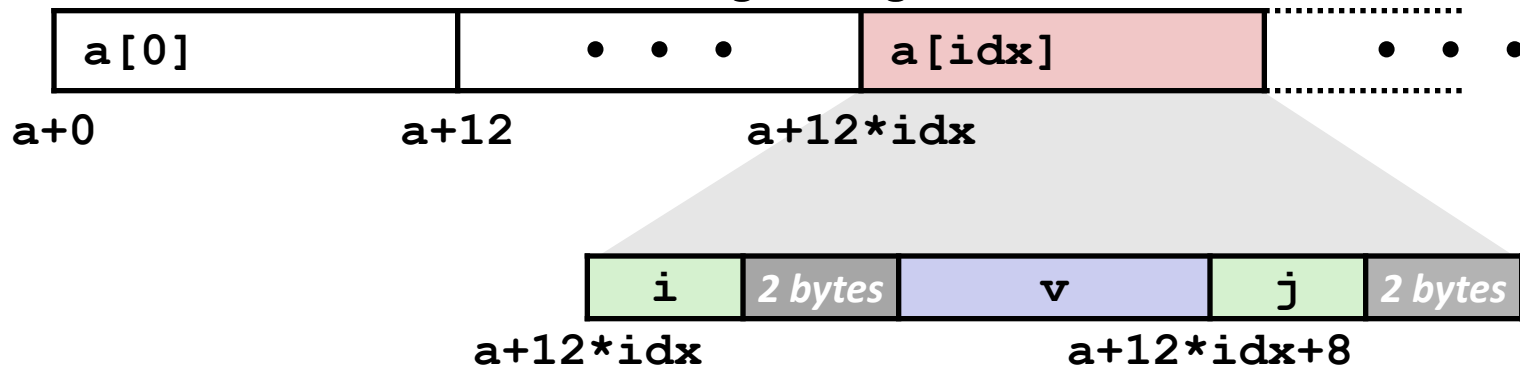


访问数组元素

Accessing Array Elements

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

- **计算数组的偏移** Compute array offset $12 * \text{idx}$
 - `sizeof(S3)`, 包括对齐所需填充空白 including alignment
- **元素j在结构内部的偏移是8** Element j is at offset 8 within structure
- **汇编器给出的偏移是a+8** Assembler gives offset a+8
 - 在链接时解析 Resolved during linking



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

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




节省空间 Saving Space

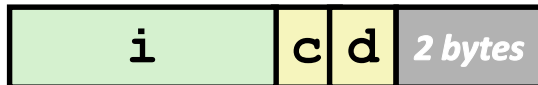
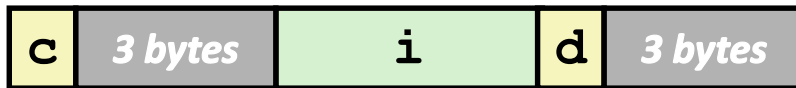
- 首先存放大的数据 Put large data types first

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



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

- 效果 Effect (K=4)





示例 结构 问题

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {  
    char a;  
    long b;  
    float c;  
    char d[3];  
    int *e;  
    short *f;  
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| a | X | X | X | X | X | X | X | b | b | b | b | b | b | b | b |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| c | c | c | c | d | d | d | X | e | e | e | e | e | e | e | e |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| f | f | f | f | f | f | f | f | | | | | | | | | | | |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| | | | | | | | | | | | | | | | | | | | | |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```



示例 结构 问题 (续)

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {  
    char a;  
    long b;  
    float c;  
    char d[3];  
    int *e;  
    short *f;  
} foo;
```

2. Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| a | d | d | d | c | c | c | c | b | b | b | b | b | b | b | b |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| e | e | e | e | e | e | e | e | f | f | f | f | f | f | f | f |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| | | | | | | | | | | | | | | | | |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| | | | | | | | | | | | | | | | | |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```



议题

■ 数组 Arrays

- 一维 One-dimensional
- 多维(嵌套) Multi-dimensional (nested)
- 多级 Multi-level

■ 结构 Structures

- 分配 Allocation
- 访问 Access
- 对齐 Alignment

■ 浮点数 Floating Point



背景 Background

■ 历史 History

- x87 FP
 - 遗留表示, 非常丑陋 Legacy, very ugly
- 流式SIMD扩展 SSE FP
 - Shark机器支持 Supported by Shark machines
 - 向量指令的特殊情况 Special case use of vector instructions
- 高级向量扩展 AVX FP
 - 最新的版本 Newest version
 - 类似于SSE Similar to SSE
 - 参考教材内容 Documented in book

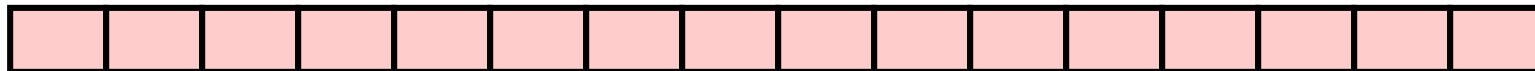


SSE4编程 Programming with SSE4

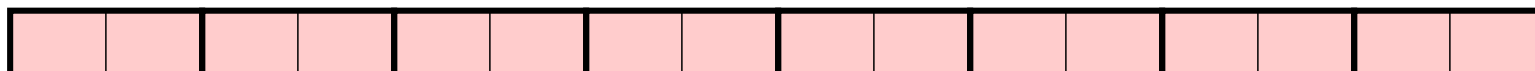
扩展多媒体寄存器 XMM Registers

■ 16个, 每个16字节 16 total, each 16 bytes

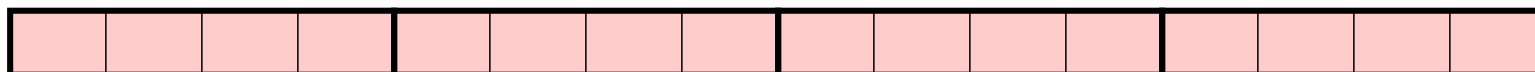
■ 16个单字节整数 16 single-byte integers



■ 8个16位整数 8 16-bit integers



■ 4个32位整数 4 32-bit integers



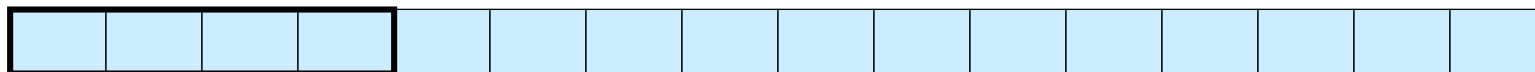
■ 4个单精度浮点数 4 single-precision floats



■ 2个双精度浮点数 2 double-precision floats



■ 1个单精度浮点数 1 single-precision float



■ 1个双精度浮点数 1 double-precision float

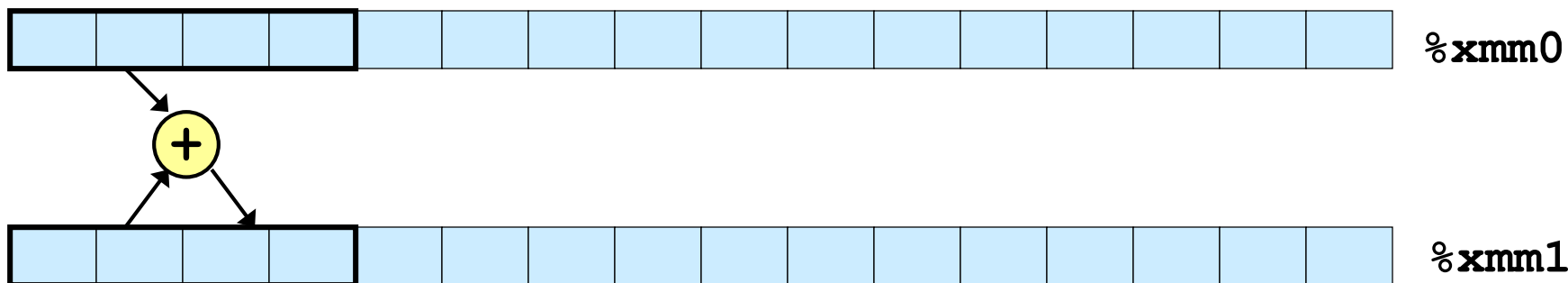


标量和SIMD操作 Scalar & SIMD Operations



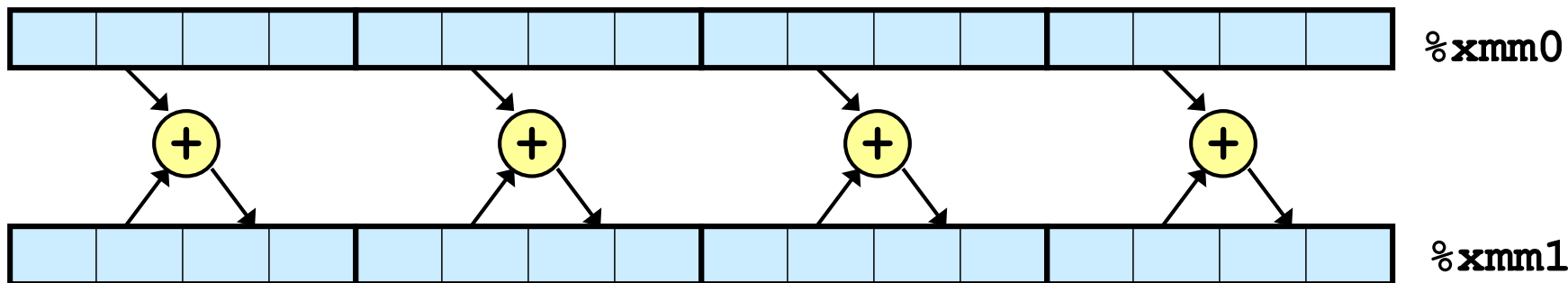
■ 标量操作：单精度 Scalar Operations: Single Precision

`addss %xmm0, %xmm1`



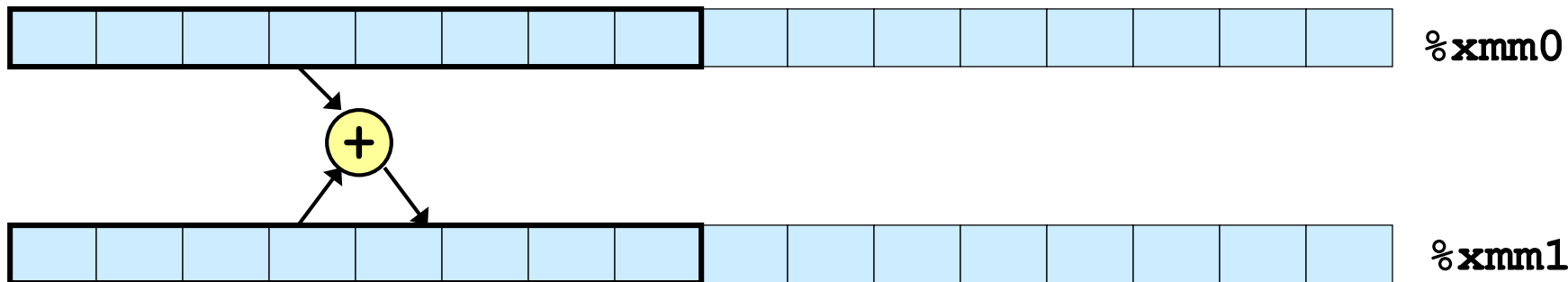
■ SIMD操作：单精度 SIMD Operations: Single Precision

`addps %xmm0, %xmm1`



■ 标量操作：双精度 Scalar Operations: Double Precision

`addsd %xmm0, %xmm1`





浮点数基础 FP Basics

- **参数传递用** Arguments passed in `%xmm0`, `%xmm1`, ...
- **结果返回用** Result returned in `%xmm0`
- **所有XMM寄存器都由调用者保存** 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
```


浮点数传送操作 FP Movement Operations



- 在内存和寄存器之间，以及寄存器和寄存器之间传送值
Transfer values between memory and registers, as well as pairs of registers

指令 Instruction	源 Source	目的 Dest	描述 Description
vmovss	M_{32}	XMM	Move single precision单精度
vmovss	XMM	M_{32}	Move single precision单精度
vmovsd	M_{64}	XMM	Move double precision双精度
vmovsd	XMM	M_{64}	Move double precision双精度
vmovaps	XMM	XMM	Move aligned, packed single precision 对齐的单精度
vmovapd	XMM	XMM	Move aligned, packed double precision 对齐的双精度

浮点数算术运算 FP Arithmetic Operations



- 一个 (s_1) 或两个 (s_1, s_2) 源操作数和一个目的操作数 D One (S_1) or two (S_1, S_2) source operands and a destination operand D

- S_1 可以是 XMM 寄存器或内存单元 S_1 can be XMM register or a memory location
- S_2 和 D 必须是 XMM 寄存器 S_2 and D must be XMM registers

单精度 Single	双精度 Double	效果 Effect	描述 Description
vaddss	vaddsd	$D \leftarrow S_2 + S_1$	FP add 加
vsubss	vsubsd	$D \leftarrow S_2 - S_1$	FP subtract 减
vmulss	vmulsd	$D \leftarrow S_2 \times S_1$	FP multiply 乘
vdivss	vdivsd	$D \leftarrow S_2 / S_1$	FP divide 除
vmaxss	vmaxsd	$D \leftarrow \max(S_2, S_1)$	FP maximum 最大
vminss	vminsd	$D \leftarrow \min(S_2, S_1)$	FP minimum 最小
sqrtps	sqrtsd	$D \leftarrow S_1^{1/2}$	FP square root 开平方

浮点代码中使用比特级运算

Using Bitwise Operations in FP Code



单精度 Single	双精度 Double	效果 Effect	描述 Description
vxorps	xorpd	$D \leftarrow S_2 \wedge S_1$	Bitwise EXCLUSIVE-OR 比特位级异或
vandps	andpd	$D \leftarrow S_2 \& S_1$	Bitwise AND 比特位级与

```
double simple (...){  
    ...  
    return 0.0;  
}
```

```
simple:  
    ...  
    vxorpd %xmm0, %xmm0, %xmm0  
    ret
```

浮点数内存引用 FP Memory Referencing



- **整数（和指针）参数传递用常规的寄存器** Integer (and pointer) arguments passed in regular registers
- **浮点值传递用XMM寄存器** FP values passed in XMM registers
- **不同的mov指令在XMM寄存器之间和内存与XMM寄存器之间传送** 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
vmovapd    %xmm0, %xmm1    # Copy v
vmovsd     (%rdi), %xmm0    # x = *p
vaddsd     %xmm0, %xmm1    # t = x + v
vmovsd     %xmm1, (%rdi)    # *p = t
ret
```

浮点代码的其它方面

Other Aspects of FP Code



■ 很多指令 *Lots of instructions*

- 不同的操作, 不同的格式 Different operations, different formats, ...

■ 浮点数比较 Floating-point comparisons

- 指令 **ucomiss** 和 **ucomisd** Instructions **ucomiss** and **ucomisd**

- 设置条件码 Set condition codes ZF, **PF** and CF

- OF和SF置零 Zeros OF and SF

Parity Flag

UNORDERED: ZF,PF,CF←111

GREATER_THAN: ZF,PF,CF←000

LESS_THAN: ZF,PF,CF←001

EQUAL: ZF,PF,CF←100

■ 使用常量值 Using constant values

- 设置XMM0寄存器为0用指令 Set XMM0 register to 0 with instruction **xorpd %xmm0, %xmm0**
- 其它从内存装入 Others loaded from memory

小结 Summary



■ 数组 Arrays

- 元素包装进连续的内存区域 Elements packed into contiguous region of memory
- 使用索引计算来定位单独的元素 Use index arithmetic to locate individual elements

■ 结构 Structures

- 元素包装进单个内存区域 Elements packed into single region of memory
- 使用由编译器确定的偏移进行访问 Access using offsets determined by compiler
- 可能需要内部和外部填充确保对齐 Possible require internal and external padding to ensure alignment

■ 组合 Combinations

- 可以任意嵌套结构和数组代码 Can nest structure and array code arbitrarily

■ 浮点数 Floating Point

- 数据装入XMM寄存器进行操作 Data held and operated on in XMM registers