

程序的机器级表示IV: 数据

教师：吴锐

计算机科学与技术学院

哈尔滨工业大学

主要内容

■ 数组

- 一维
- 多维(嵌套)
- 多层次

■ 结构体

- 内存分配
- 访问
- 对齐

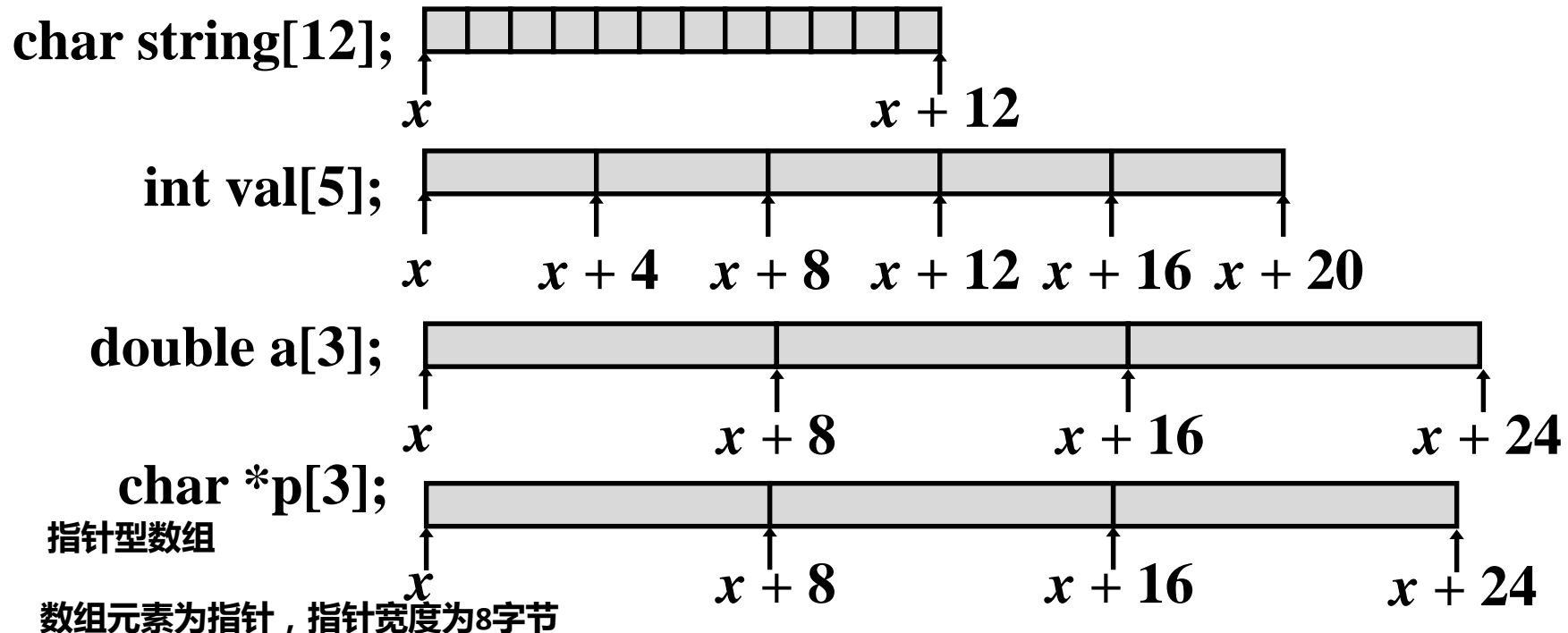
■ 联合

数组的内存分配

■ 基本准则

$T\ A[L];$

- 数据类型 T 、长度 L 的数组
- 在内存中连续分配的 $L * \text{sizeof}(T)$ 字节

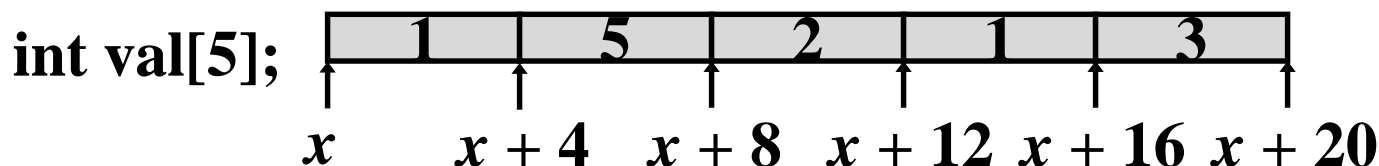


数组的访问

■ 基本准则

$T \ A[L];$

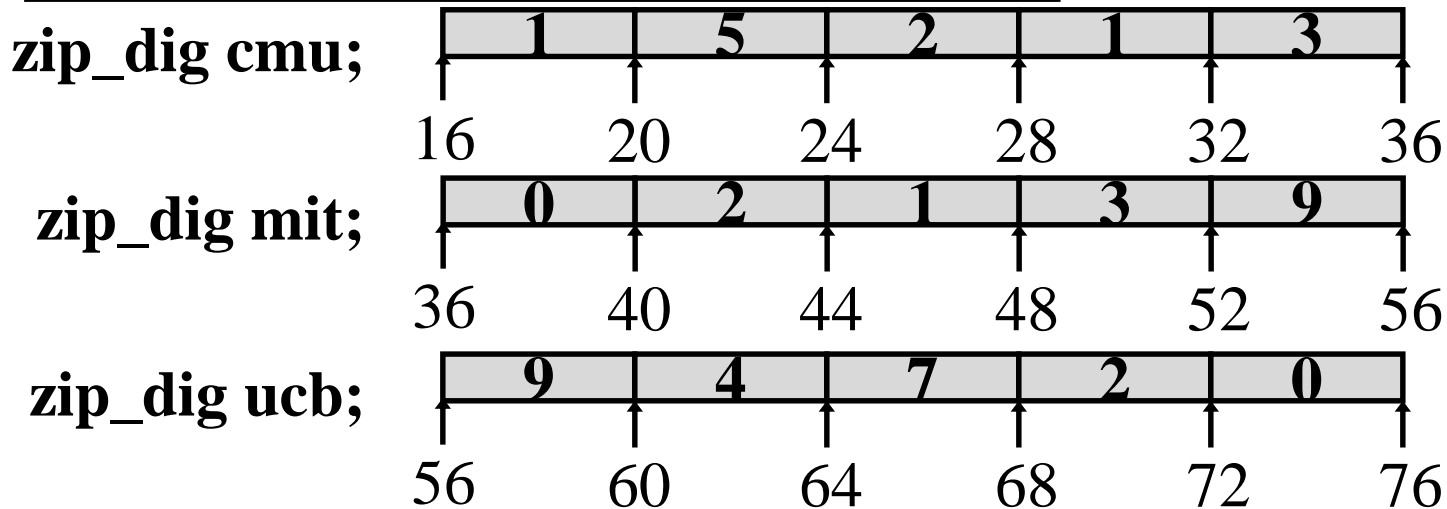
- 数据类型 T 、长度 L 的数组
- 标识符 A 可作为数组元素0的指针(常量): Type T^*



引用形式	类型	数值
<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>	$x + 4 i$

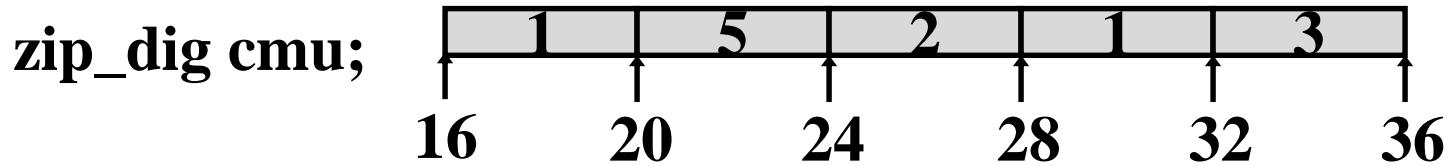
数组例子

```
#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]”
- 示例数组申请20个连续的内存字节(sizeof(cmu) or sizeof(zip_dig))

数组访问例子



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

IA32

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

- 寄存器 `%rdi` 保存数组的起始地址
- 寄存器 `%rsi` 保存数组元素的下标(索引)
- 期望的数据地址:
 $\text{\%rdi} + 4 * \text{\%rsi}$
- 内存寻址形式
 $(\text{\%rdi}, \text{\%rsi}, 4)$

数组和循环的例子

```
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
```

多维(嵌套) 数组

■ 声明

T $A[R][C];$

- 数据类型 T 的两维数组
- R 行, C 列
- 元素类型 T , K 字节

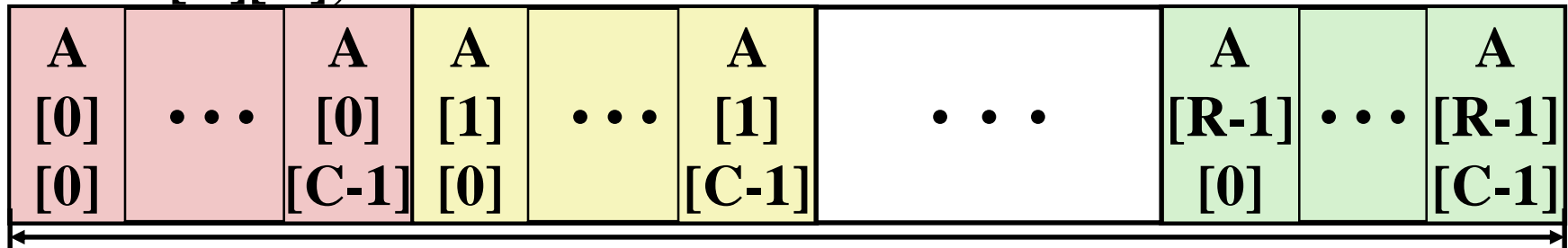
■ 数组尺寸、sizeof(A)

- $R * C * K$ 字节

■ 存储：行优先排列

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

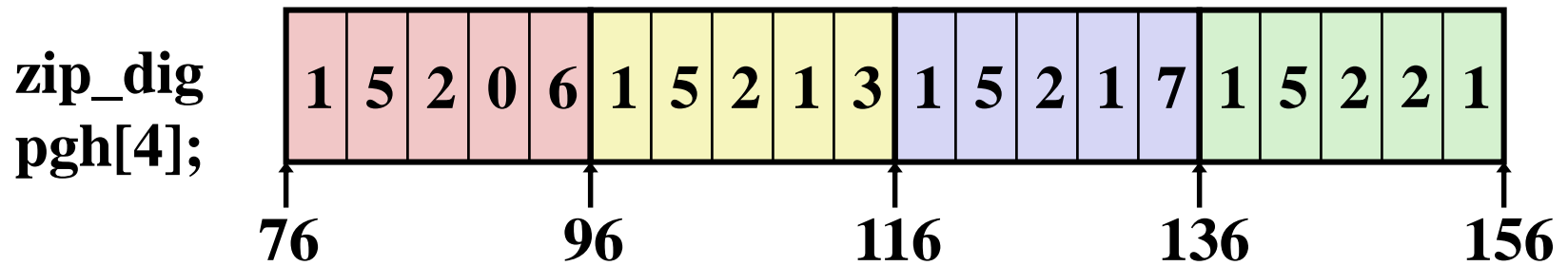
int $A[R][C];$



$4 * R * C$ Bytes

嵌套数组例子

```
#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 }};
```

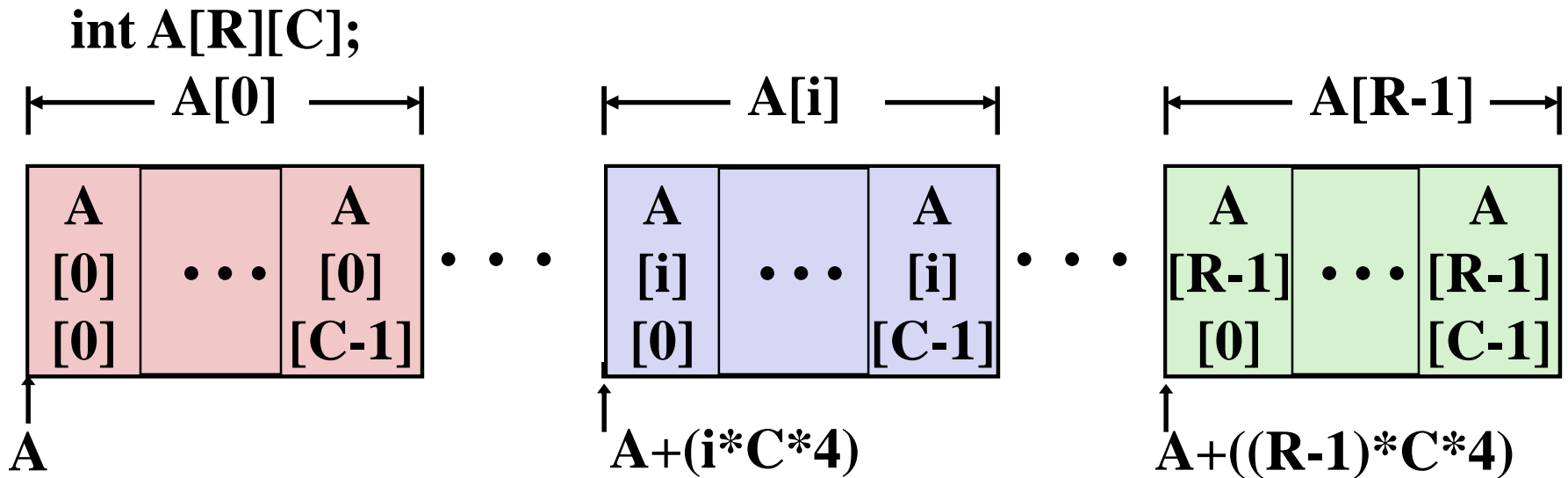


- “`zip_dig pgh[4]`” 等价于 “`int pgh[4][5]`”
 - 变量 `pgh`: 有4元素的数组, 占用连续内存
 - 每个元素是一个有5个整数的数组, 占用连续内存
- 内存排列: 行优先

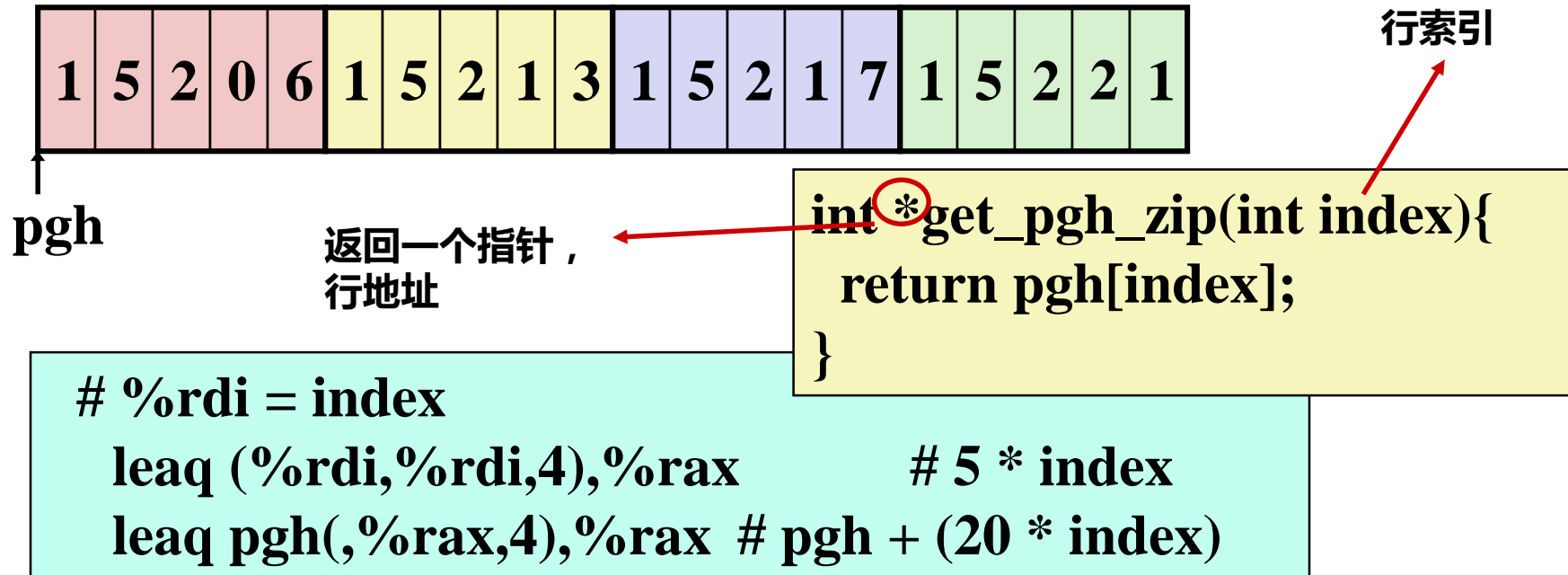
嵌套数组行访问

■ 行向量

- $A[i]$ 是 C 个元素的数组
- 类型 T 的每个元素需要 K 个字节
- 起始地址 $A + i * (C * K)$



嵌套数组行访问代码



■ 行向量

- `pgh[index]` : 有5个整数的数组
- 起始地址 `pgh+20*index`

■ 机器代码

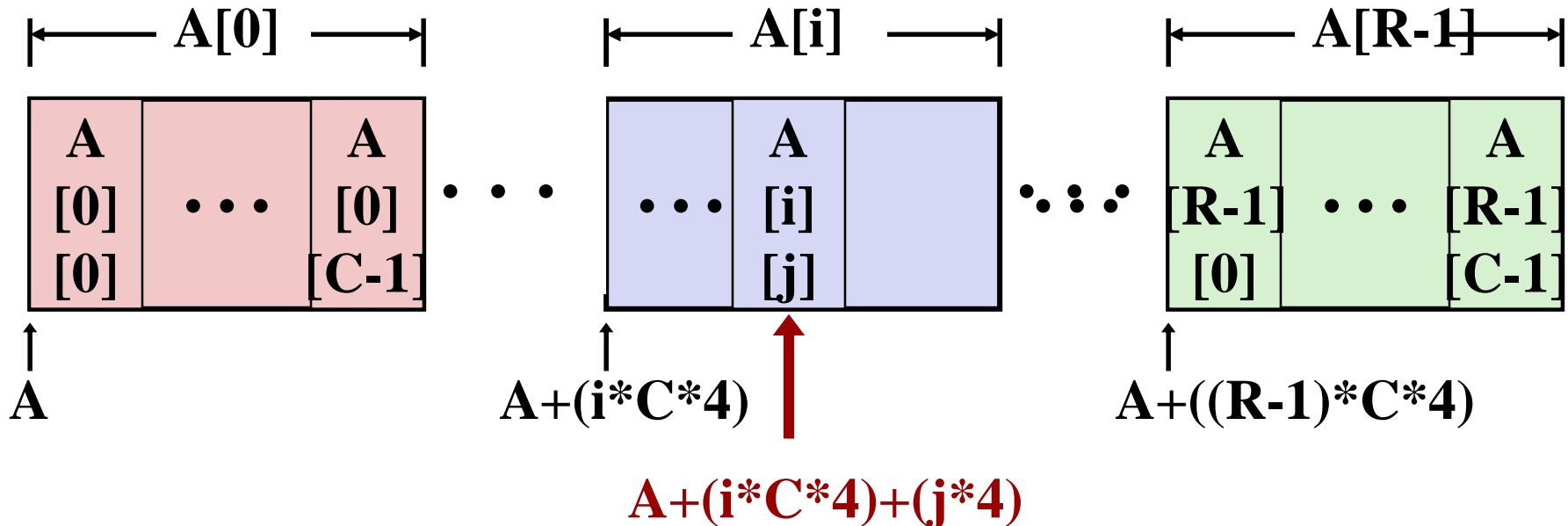
- 计算和返回地址: `pgh + 4 * (index+4*index)`

嵌套数组元素访问

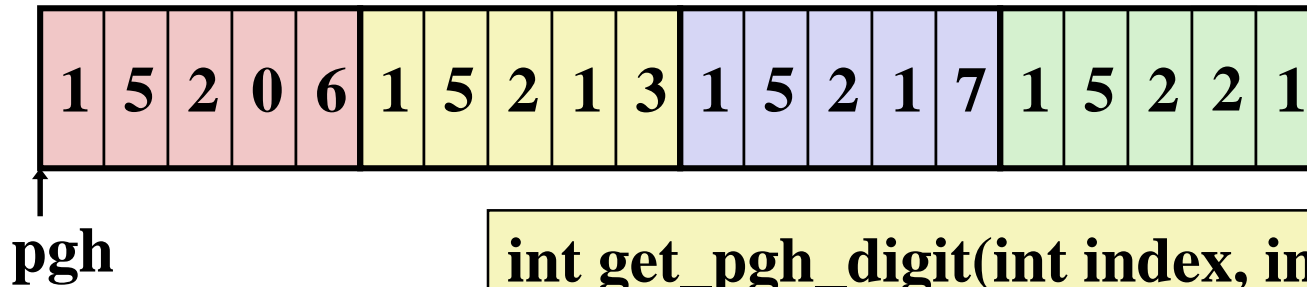
■ 数组元素

- $A[i][j]$ 类型为 T 的元素, 需要 K 个字节
- 地址: $A + i * (C * K) + j * K = A + (i * C + j) * K$

`int A[R][C];`



嵌套数组元素访问代码



```
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

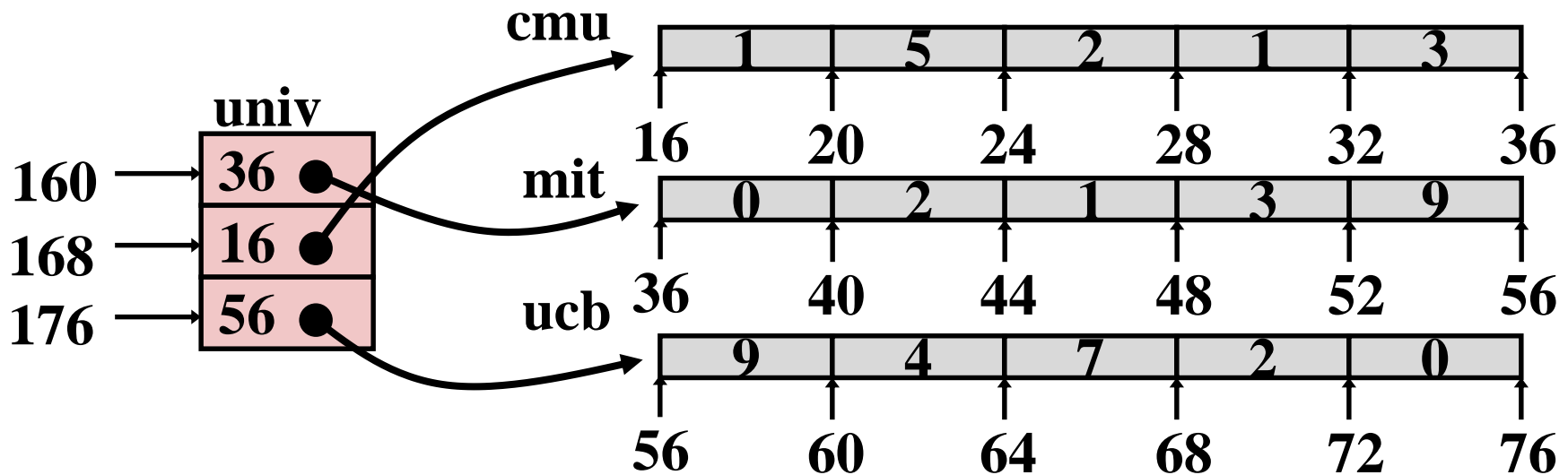
- `pgh[index][dig]` 是 `int` 型
- 地址: `pgh + 20*index + 4*dig`
 - `= pgh + 4*(5*index + dig)`

多层次数组例子

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

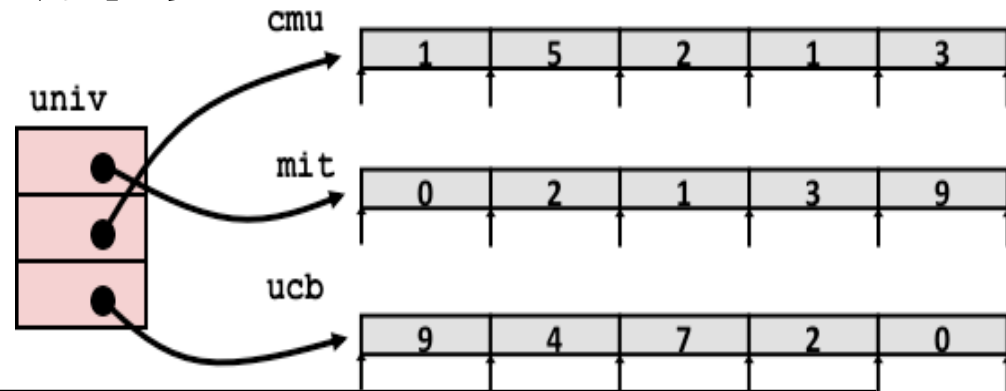
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- 变量 `univ` 是有3个元素的数组
- 每个元素是指针类型
 - 8 bytes
- 每个指针指向一个整数数组



多层次数组元素的访问

```
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

- 元素访问 $\text{Mem}[\text{Mem}[\text{univ} + 8 * \text{index}] + 4 * \text{digit}]$
- 需要两次内存读
 - 首先，获取行数组的地址
 - 然后，访问数组内的元素

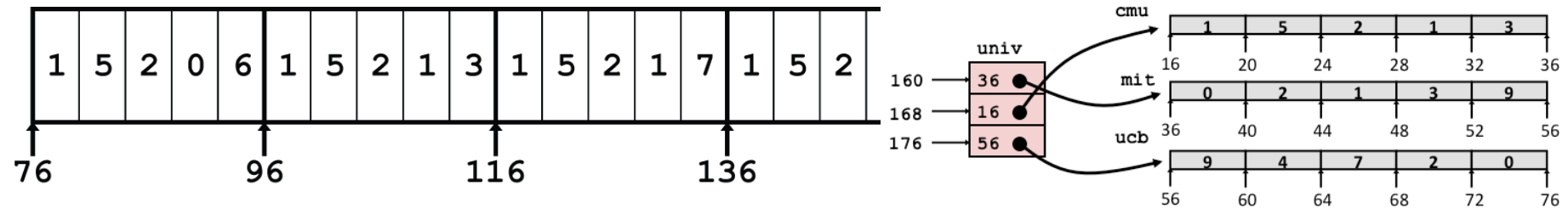
数组元素访问

嵌套数组

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

多层次数组

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



与C相似，但地址的计算方式完全不同：

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

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

N × N 矩阵

■ 固定维数

- 编译的时候有确定的N值

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

■ 可变维数：显示索引

- 动态数组的传统实现方法

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

■ 可变维数：隐含索引

- gcc支持

```
/* 获得元素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 矩阵的访问

■ 数组元素 $A[i][j]$

- 地址: $A + i * (C * K) + j * K$
- $C = 16, K = 4$

/* 获得元素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 矩阵的访问

■ 数组元素A[i][j]

- 地址: $A + i * (C * K) + j * K$
- $C = n, K = 4$
- 必须实现整数乘积

```
/* 获得元素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
```

主要内容

■ 数组

- 一维
- 多维(嵌套)
- 多层次

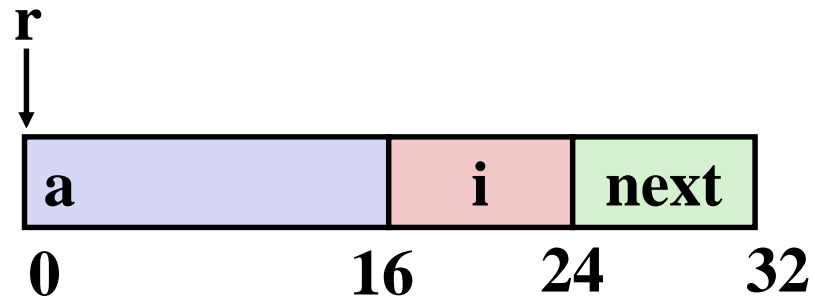
■ 结构体

- 内存分配
- 访问
- 对齐

■ 联合

结构体表示

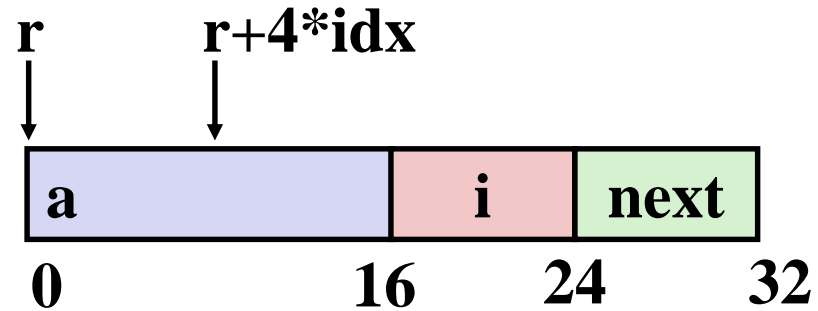
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- 结构体用内存块来表示
 - 足够大，可容纳所有字段
- 字段顺序**必须**与声明一致
 - 即便其他顺序能使得内存更紧凑——**也不行！**
- 编译器决定总的尺寸和各字段位置
 - 机器级程序不解读(理解)源代码中的结构体

结构体成员地址的生成

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



■ 数组元素的地址

- 每个结构体成员的偏移量(Offset)是在编译阶段确定的
- 地址计算形式:
 $r + 4 * idx$

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

→ 取a[idx]的地址

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

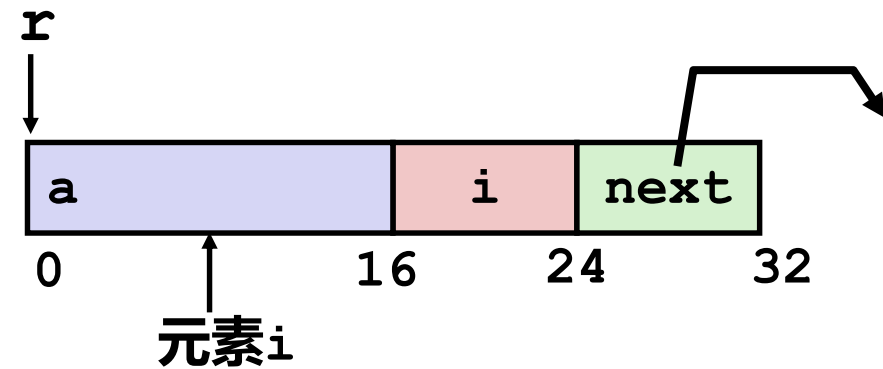
链表

■ C代码

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

寄存器	数值
%rdi	r
%rsi	val

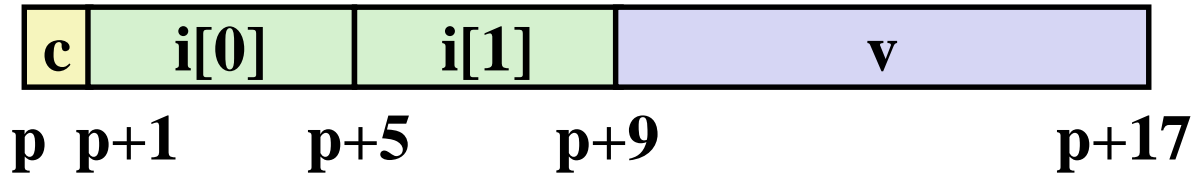
```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```



```
.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
```

结构体与对齐

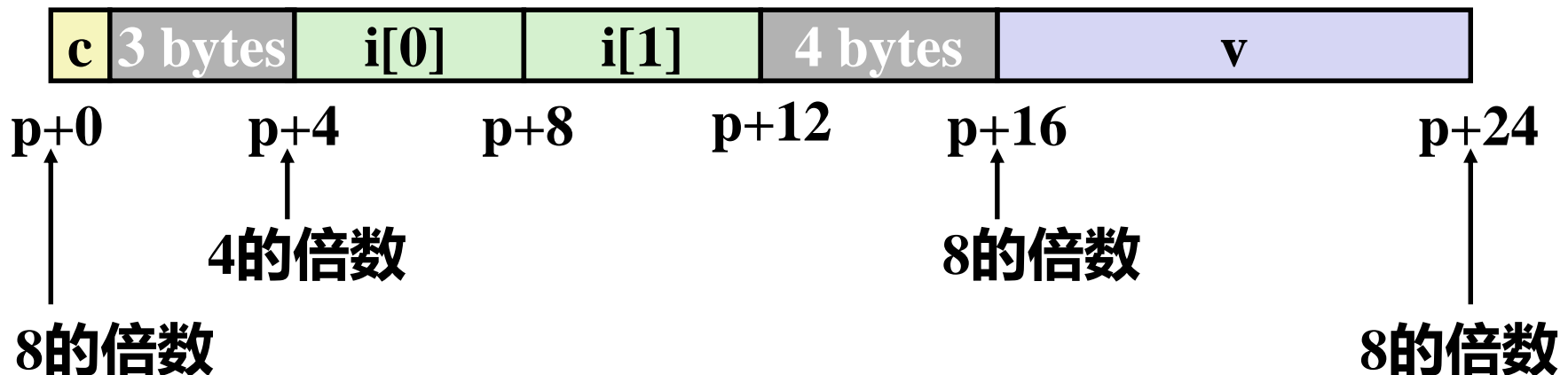
■ 未对齐的数据



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

■ 对齐后的数据

- 基本数据类型需要 K 字节
- 地址必须是 K 的倍数



对齐的准则

■ 对齐后的数据

- 基本数据类型需要 k 字节
- 地址必须是 k 的倍数
- 一些机器要求、x86-64机器推荐

■ 对齐数据的动机

- 内存按4字节或8字节(对齐的)块来访问（4/8依赖于系统）
 - 不能高效地装载或存储跨越四字边界的数据
 - 当一个数据跨越2个页面时，虚拟内存比较棘手

■ 编译器

- 在结构体中插入空白，以确保字段的正确对齐

x86-64对齐

- **1字节: `char`, ...**
 - 对地址无要求
- **2字节: `short`, ...**
 - 低字节地址必须偶数: $*****0_2$
- **4字节: `int`, `float`, ...**
 - 低字节地址必须是4的倍数: $*****00_2$
- **8字节: `double`, `long`, `char *`, ...**
 - 低字节地址必须是8的倍数: $*****000_2$
- **16字节: `long double` (GCC on Linux)**
 - 低字节地址必须是16的倍数: $*****0000_2$

结构体的对齐

■ 结构体内部

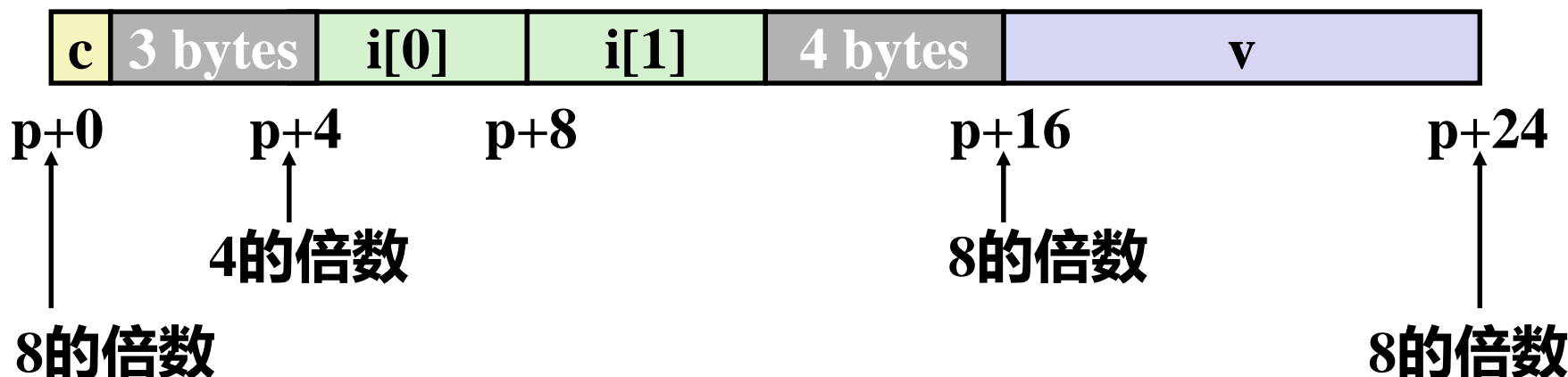
- 满足每个元素的对齐要求

■ 结构体的整体对齐存放

- 结构体的整体对齐要求值K
 - K = 所有元素的最大对齐要求值
- 起始地址& 结构体长度必须是 K的倍数

■ Example: K = 8, 有double型元素

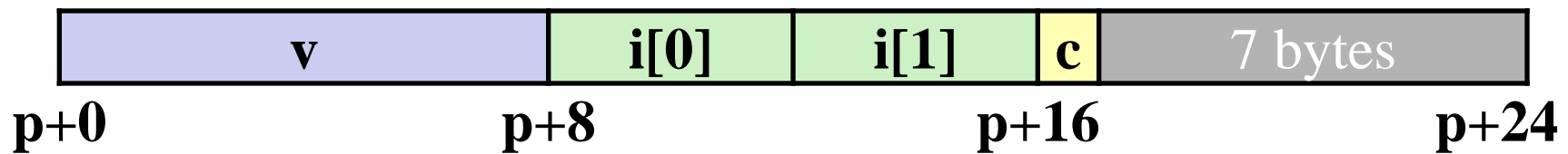
```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```



满足整体对齐要求

- 最大对齐要求: K
- 结构体整体大小必须是 K 的倍数

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

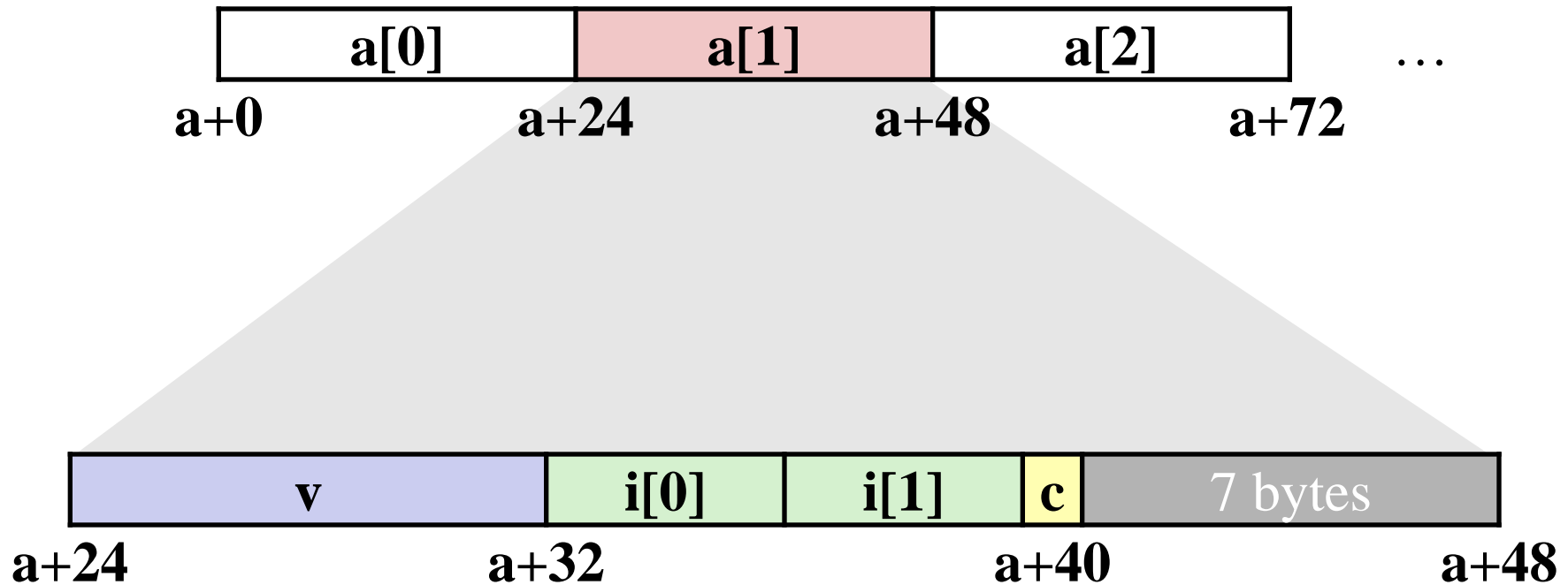


K 的倍数($K=8$)

结构体数组

- 结构体整体大小：K的倍数
- 每个元素都满足对齐要求

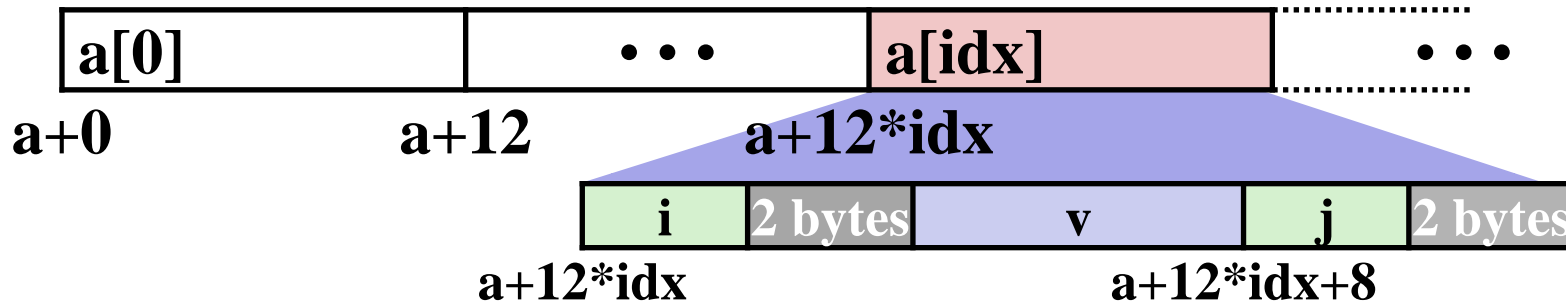
```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



访问数组元素

- 计算数组元素的offset: $12 * \text{idx}$
 - `sizeof(S3)`, 包括对齐引入的空白
- 字段 `j` 在结构体内的offset: 8
- 汇编器给出的offset: `a+8`
 - `a`: 链接时确定

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



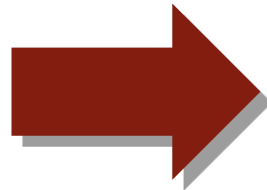
```
short get_j(int idx)
{
    return a[idx].j;
}
```

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

空间的节省

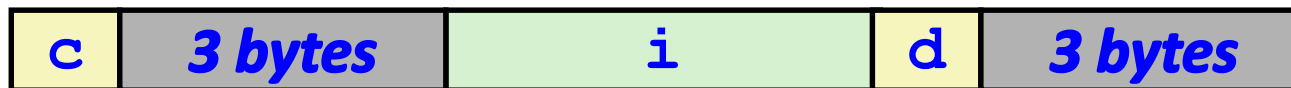
■ 大尺寸数据类型在前

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



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

■ 节省效果 (K=4)



主要内容

■ 数组

- 一维
- 多维(嵌套)
- 多层次

■ 结构体

- 内存分配
- 访问
- 对齐

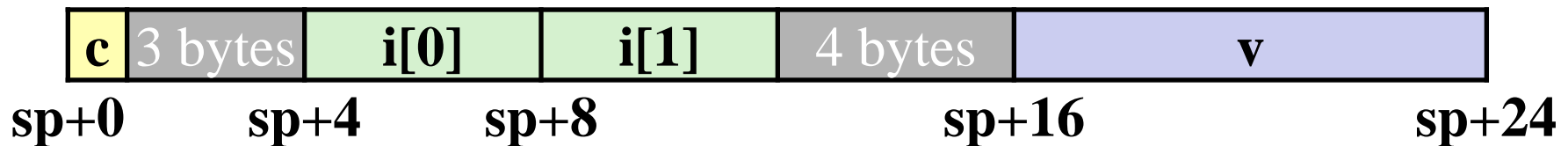
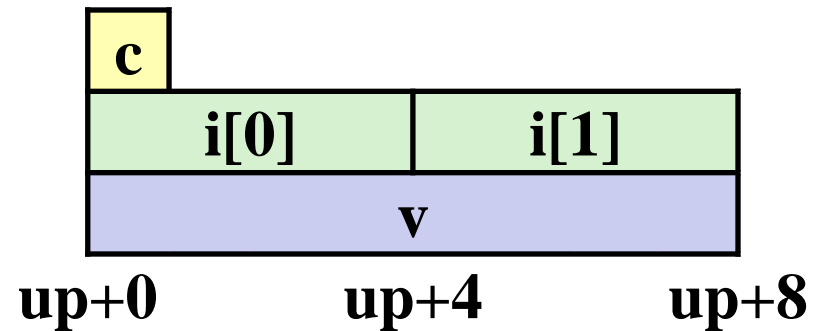
■ 联合

联合的内存分配

- 依据最大成员申请内存
- 同时只能使用一个成员

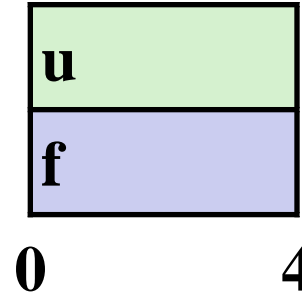
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

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



使用联合获取位模式

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



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

是否和(float)u 相同？

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

是否和(unsigned)f 相同？

字节序

■ 想法

- short/long/quad words 在内存中用连续的2/4/8 字节存储
- 哪个字节是最高/低位？
- 在不同机器之间交换顺序，会有问题。

■ 大端序(Big Endian)

- 最高有效位在低地址，如sun 的工作站Sparc

■ 小端序(Little Endian)

- 最低有效位在低地址，如Intel x86, ARM Android、IOS

■ 双端序(Bi Endian)

- 可配置成大/小端序，如ARM

字节序的例子

```
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]							

```
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]);
```

IA32的字节序

小端序

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]							

LSB ← **MSB** **LSB** **MSB**
Print

输出:

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]

Sun的字节序

大端序

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 $\xrightarrow{\text{Print}}$ LSB MSB LSB

Sun机器的输出:

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]

x86-64的字节序

小端

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]							

LSB ←—————→ MSB
Print

x86-64机器的输出

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]

C语言复合类型总结

■ 数组

- 连续分配内存
- 对齐：满足每个元素对齐要求
- 数组名是首个元素的指针常量
- 没有越界检查！

■ 结构体

- 各成员按结构体定义中的顺序分配内容
- 在中间、末尾填充字节，以满足对齐要求

■ 联合

- 覆盖的声明
- 规避类型系统对编程束缚的方法

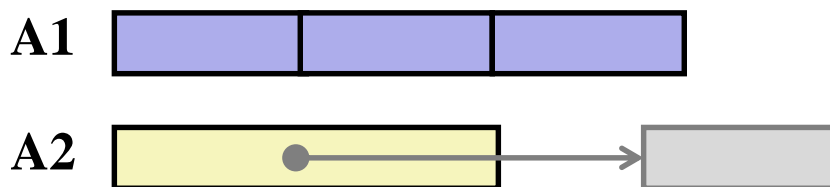
理解指针和数组#1

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

- Cmp: 能通过编译 (Y/N)
- Bad: 可能有错误指针引用(Y/N)
- Size: sizeof()的返回值

理解指针和数组#1

声明	A_n			$*A_n$		
	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>	Y	N	12	Y	N	4
<code>int *A2</code>	Y	N	8	Y	Y	4



- Cmp: 能通过编译 (Y/N)
- Bad: 可能有错误指针引用(Y/N)
- Size: sizeof()的返回值

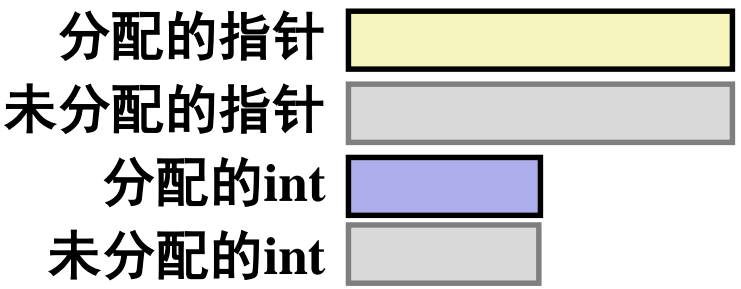
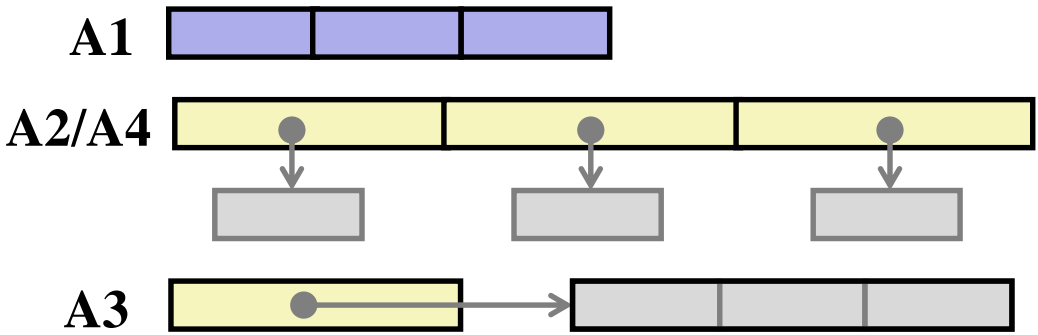
理解指针和数组#2

声明	An			$*An$			$**An$		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]									
int *A2[3]									
int (*A3)[3]									
int (*A4[3])									

- Cmp: 能通过编译 (Y/N)
- Bad: 可能有错误指针引用(Y/N)
- Size: sizeof()的返回值

理解指针和数组#2

声明	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



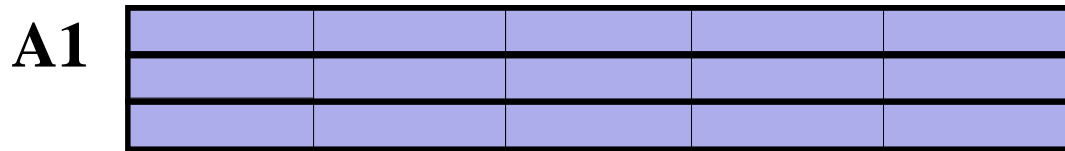
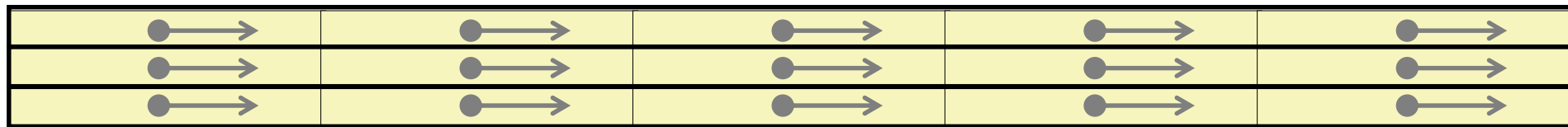
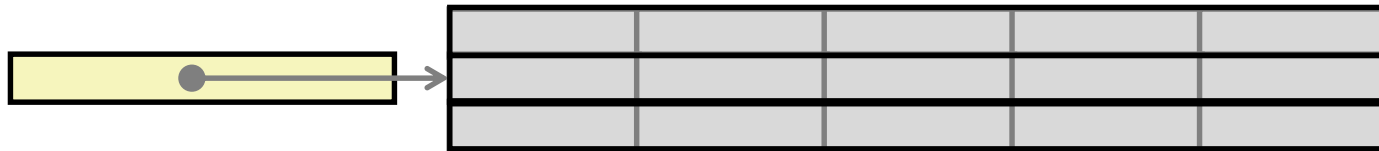
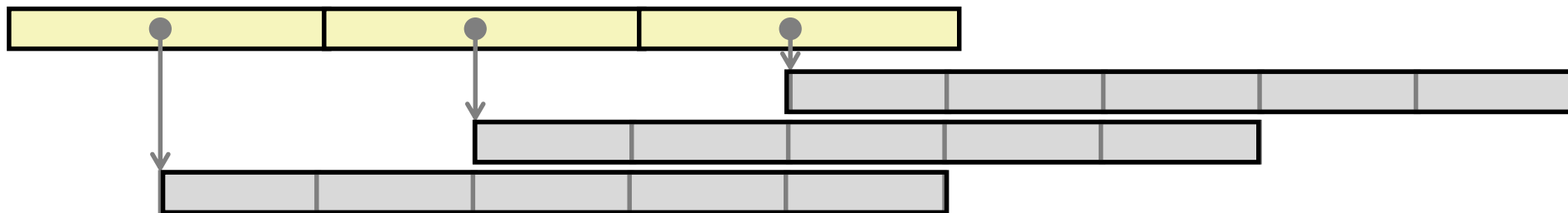
理解指针和数组#3

声明	<i>An</i>			<i>*An</i>			<i>**An</i>		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3][5]									
int *A2[3][5]				声明			<i>***An</i>		
int (*A3)[3][5]							Cmp	Bad	Size
int *(A4[3][5])				int A1[3][5]					
int (*A5[3])[5]				int *A2[3][5]					
				int (*A3)[3][5]					
				int *(A4[3][5])					
				int (*A5[3])[5]					

- Cmp: 能通过编译 (Y/N)
- Bad: 可能有错误指针引用(Y/N)
- Size: sizeof()的返回值



声明

`int A1[3][5]``int *A2[3][5]``int (*A3)[3][5]``int *(A4[3][5])``int (*A5[3])[5]`**A2/A4****A3****A5**

理解指针和数组#3

声明	A_n			$*A_n$			$**A_n$		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	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: 能通过编译 (Y/N)
- Bad: 可能有错误指针引用(Y/N)
- Size: sizeof()的返回值

声明	$***A_n$		
	Cmp	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