机器级编程 3
Machine-Level Programming III

课 程 名 ：计算机系统

第 5 讲 （2025 年 4 月 30 日）

主 讲 人 ：杜海文

本节内容

- **递归过程**
- 数组Arrays
  - 一维数组One-dimensional
  - 多维数组（嵌套）Multi-dimensional (nested)
  - 多级数组（指针数组）Multi-level (array of pointers)

递归函数

```
/* Recursive popcount */
long pcount_r(unsigned long x)
{
    if (x == 0)
        return 0;
    else
        return (x & 1) +
               pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归终点

```
/* Recursive popcount */
long pcount_r(unsigned long x)
{
    if (x == 0)
        return 0;
    else
        return (x & 1) +
               pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

同 ret
见 3.6.4 旁注

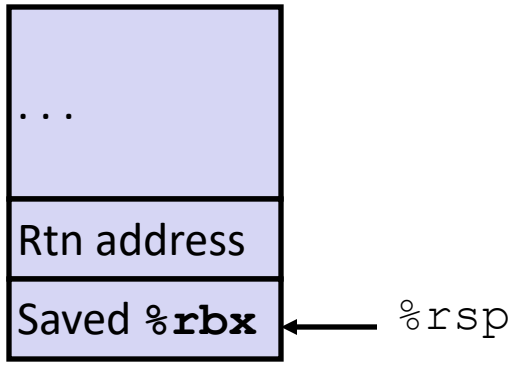
寄存器	取值	种类
%rdi	x	Argument
%rax	Return value	Return value

递归过程对寄存器的保护

```
/* Recursive popcount */
long pcount_r(unsigned long x)
{
    if (x == 0)
        return 0;
    else
        return (x & 1) +
               pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

寄存器	取值	种类
%rdi	x	Argument



递归过程调用准备

```
/* Recursive popcount */
long pcount_r(unsigned long x)
{
    if (x == 0)
        return 0;
    else
        return (x & 1) +
               pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

寄存器	取值	种类
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

递归过程调用

```
/* Recursive popcount */
long pcount_r(unsigned long x)
{
    if (x == 0)
        return 0;
    else
        return (x & 1) +
               pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

寄存器	取值	种类
%rdi	x >> 1	Callee-saved
%rax	Recursive call return value	

递归过程的结果

```
/* Recursive popcount */
long pcount_r(unsigned long x)
{
    if (x == 0)
        return 0;
    else
        return (x & 1) +
               pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

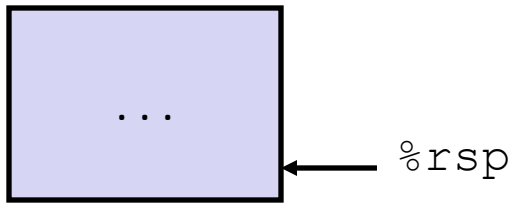
寄存器	取值	种类
%rbx	x & 1	Callee-saved
%rax	Return value	

递归过程的完成

```
/* Recursive popcount */
long pcount_r(unsigned long x)
{
    if (x == 0)
        return 0;
    else
        return (x & 1) +
               pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

寄存器	取值	种类
%rax	Return value	Return value

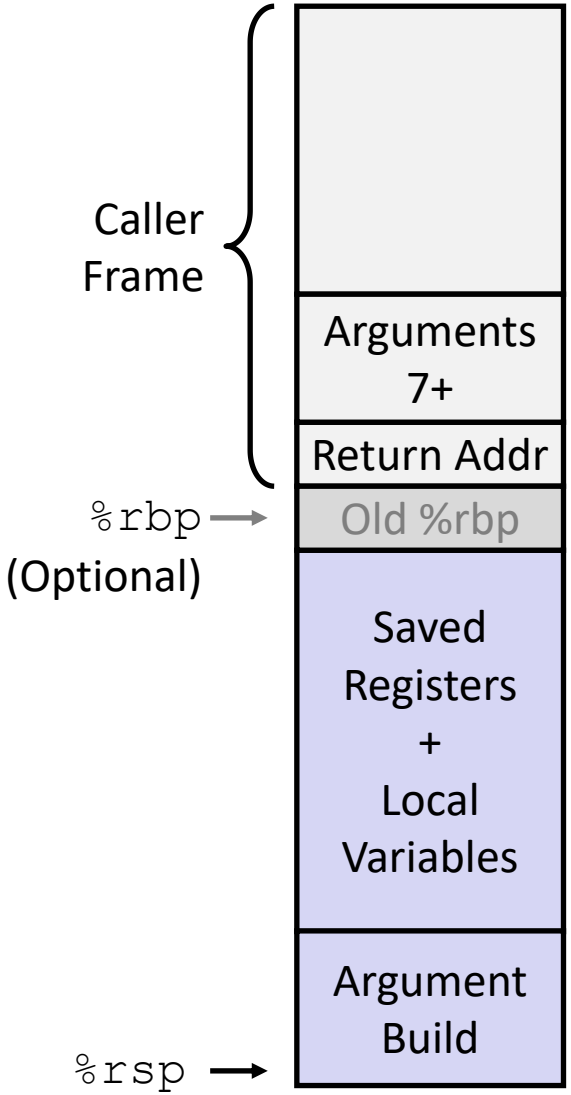


对递归的进一步观察

- 处理方式与普通过程并无不同
  - 栈帧的采用，意味着系统会为每一次过程调用分配其私有的存储空间
    - Saved registers & local variables
    - Saved return pointer
    - 即使变量同名，亦为不同变量
  - 寄存器保存约定，可确保本过程所需的数据不被其它过程破坏
    - Unless the C code explicitly does so (e.g., buffer overflow)
  - 栈的运行规则与 call / return 的方式相符合
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- 对于相互（间接）递归同样适用
  - P calls Q; Q calls P

x86-64 过程小结

- 重点
  - 栈是处理过程调用与返回的天选数据结构
    - If P calls Q, then Q returns before P
- 递归（含相互递归）采用的调用规则与普通过程完全一样
  - 过程所需的数值被妥善保存在其专属栈帧当中，或者也可放心地存于 callee-saved 寄存器中
  - 超量的函数参数置于靠近栈顶处
  - 结果由 `%rax` 传回
- 指针的实质是数值的地址
  - 位于栈区或全局数据区（第 7 章）



本节内容

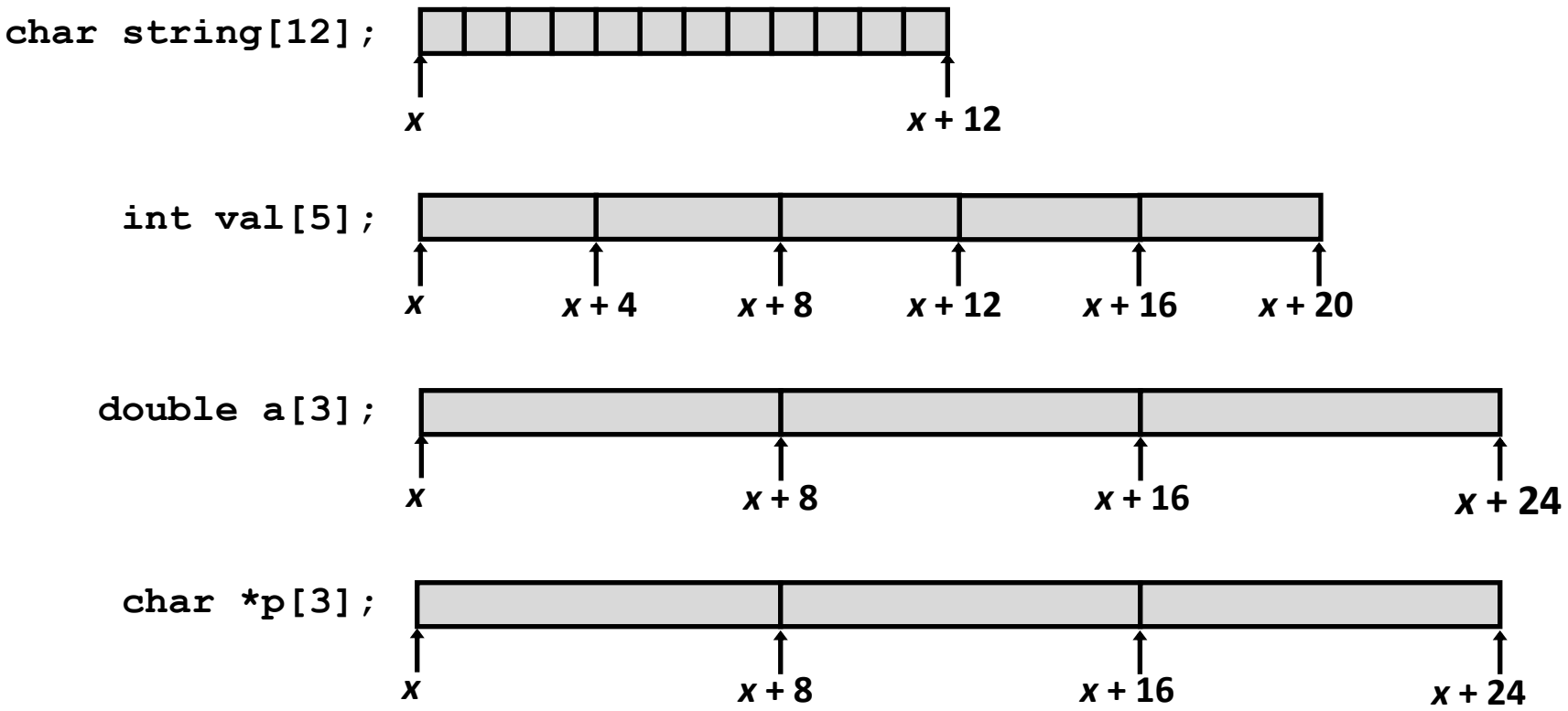
- 递归过程
- **数组Arrays**
  - **一维数组**
  - **多维数组（嵌套）Multi-dimensional (nested)**
  - **多级数组（指针数组）Multi-level (array of pointers)**

数组的内存分配

- **基本原则**

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

  - 类型为 *T*，长度为 *L* 的数组
  - 在内存中为其分配一段大小为 *L* * sizeof(*T*) 字节的连续区域

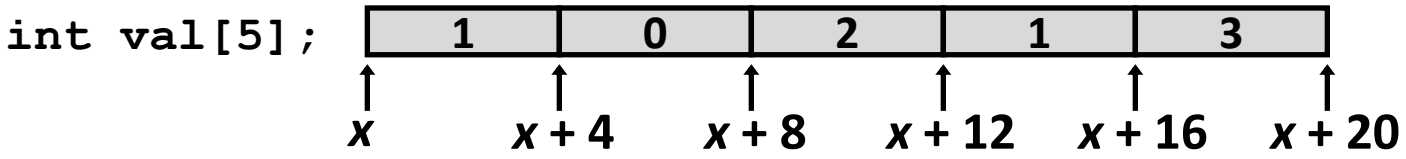


数组的访问

- 基本原则

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

  - 类型为 *T*，长度为 *L* 的数组
  - 标识符 A 可作为指向数组第 0 号元素的指针，其类型为 *T**



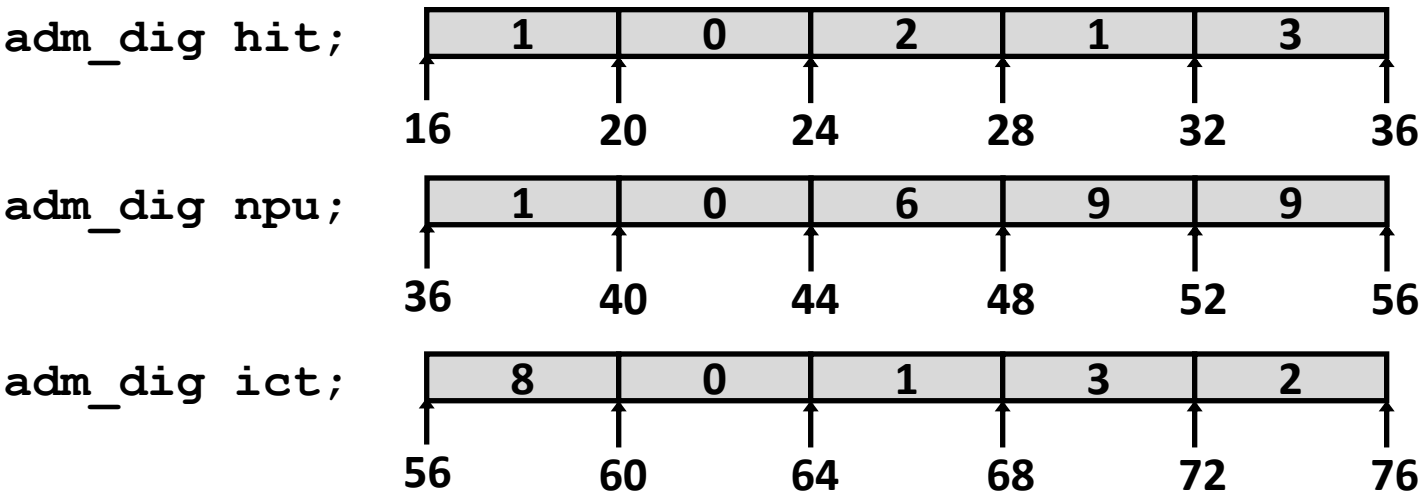
- 引用 类型 取值

val[4]	int	3
val	int *	*x*
val+1	int *	*x* + 4
&val[2]	int *	*x* + 8
val[5]	int	??
*(val+1)	int	0
val + *i*	int *	*x* + 4*i*

数组举例

```
#define ALEN 5
typedef int adm_dig[ALEN];

adm_dig hit = { 1, 0, 2, 1, 3 };
adm_dig npu = { 1, 0, 6, 9, 9 };
adm_dig ict = { 8, 0, 1, 3, 2 };
```



- 声明“adm_dig hit”等同于“int hit[5]”
- 本例中，会为每一个数组分配连续 20 字节的内存区域
  - 但在其它系统中不保证一定如此

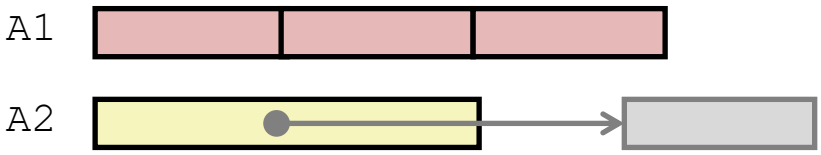
理解指针和数组 #1

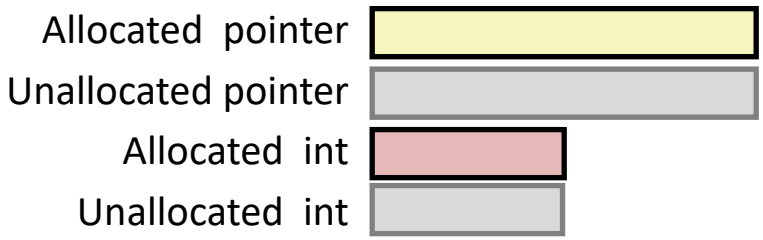
声明	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`						
`int *A2`						

- Cmp：能否编译通过（Y/N）
- Bad：引用可能失败（Y/N）
- Size：`sizeof` 的结果

理解指针和数组 #1

声明	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：能否编译通过（Y/N）
- Bad：引用可能失败（Y/N）
- Size：sizeof 的结果

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])`									

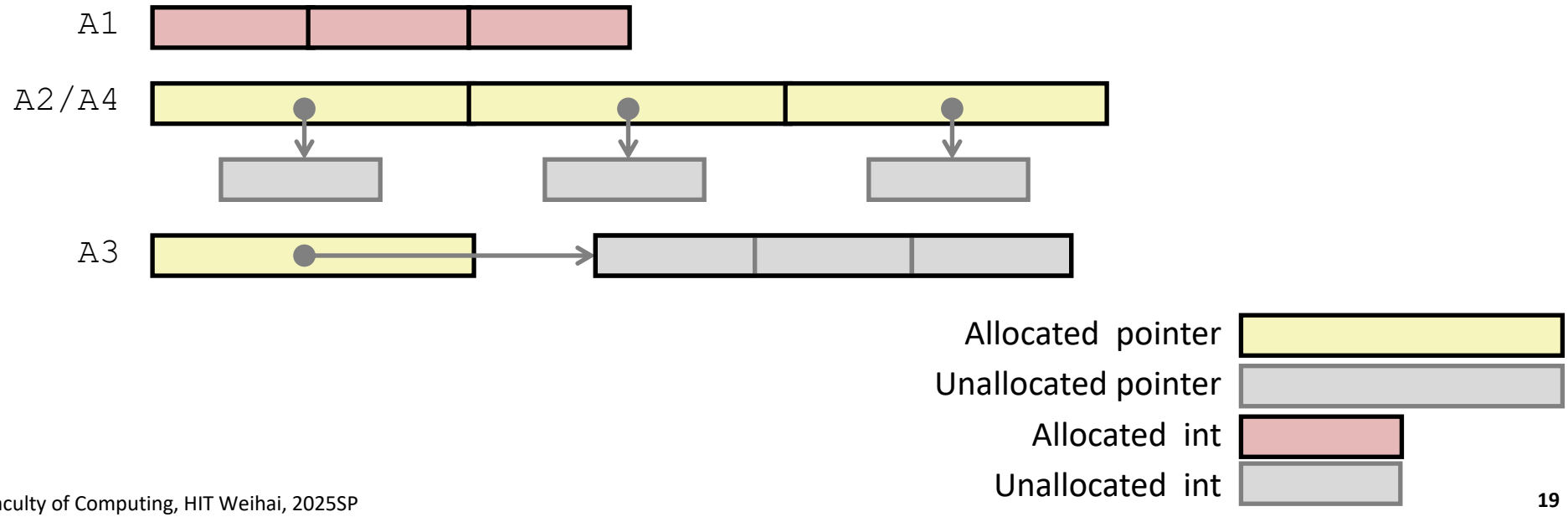
- Cmp：能否编译通过（Y/N）
- Bad：引用可能失败（Y/N）
- Size：`sizeof` 的结果

TABLE 2-1. PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

OPERATORS	ASSOCIATIVITY
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right

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

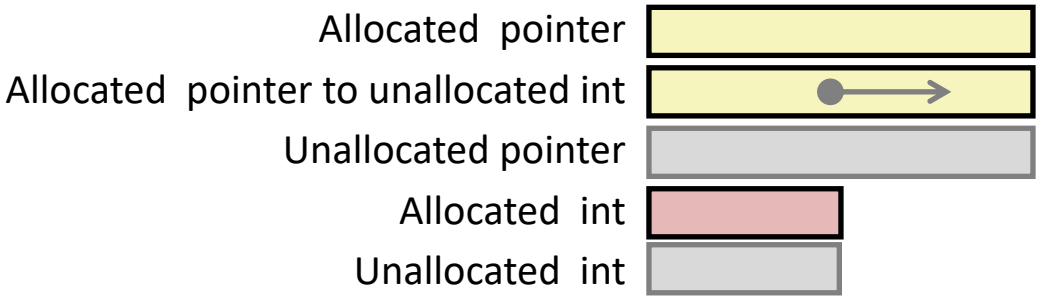


理解指针和数组 #3

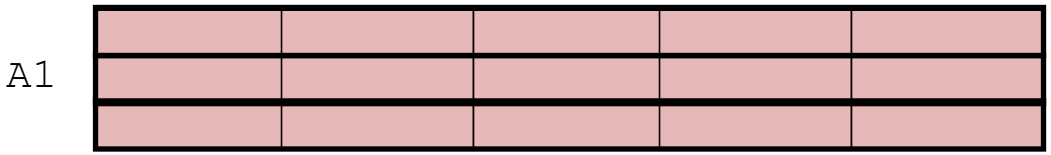
声明	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`									
`int *A2[3][5]`									
`int (*A3)[3][5]`									
`int *(A4[3][5])`									
`int (*A5[3])[5]`									

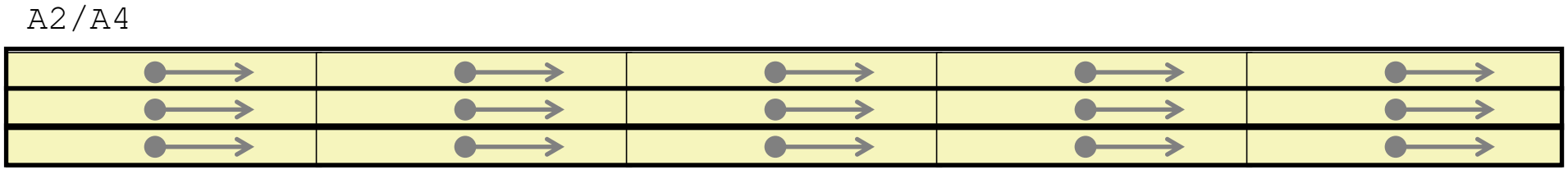
- Cmp：能否编译通过（Y/N）
- Bad：引用可能失败（Y/N）
- Size：`sizeof` 的结果

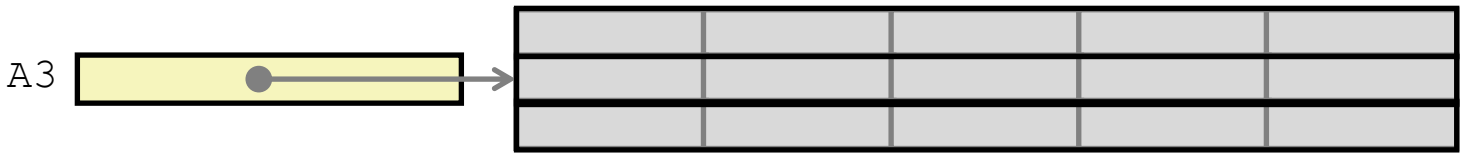
声明	***An		
	Cmp	Bad	Size
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

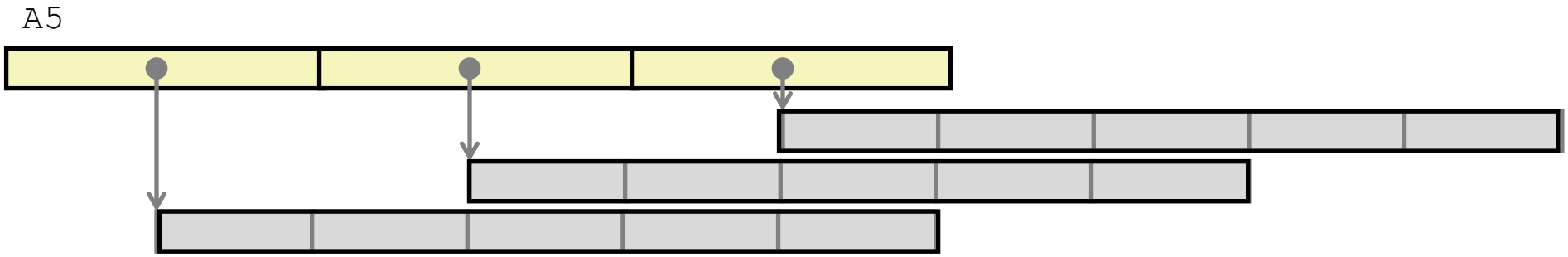


声明
`int A1[3][5]`
`int *A2[3][5]`
`int (*A3)[3][5]`
`int *(A4[3][5])`
`int (*A5[3])[5]`









理解指针和数组 #3

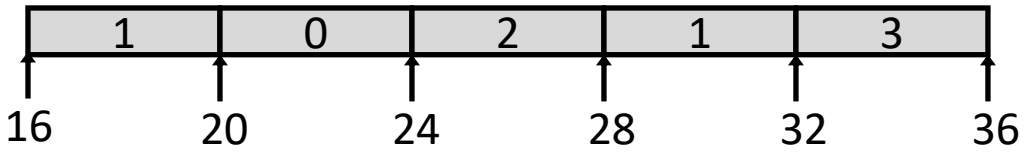
声明	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`	Y	N	60	Y	N	20	Y	N	4
`int *A2[3][5]`	Y	N	120	Y	N	40	Y	N	8
`int (*A3)[3][5]`	Y	N	8	Y	Y	60	Y	Y	20
`int *(A4[3][5])`	Y	N	120	Y	N	40	Y	N	8
`int (*A5[3])[5]`	Y	N	24	Y	N	8	Y	Y	20

- Cmp：能否编译通过（Y/N）
- Bad：引用可能失败（Y/N）
- Size：`sizeof` 的结果

声明	***An		
	Cmp	Bad	Size
`int A1[3][5]`	N	-	-
`int *A2[3][5]`	Y	Y	4
`int (*A3)[3][5]`	Y	Y	4
`int *(A4[3][5])`	Y	Y	4
`int (*A5[3])[5]`	Y	Y	4

数组访问举例

```
adm_dig hit;
```




```
int get_digit(adm_dig a, int digit)
{
    return a[digit];
}
```

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

- %rdi 寄存器存放数组的起始地址
- %rsi 存放数组元素的下标
- 欲访问的数字位于地址：%rdi + 4*%rsi
- 内存引用：(%rdi,%rsi,4)

数组循环访问举例

```
void aincr(adm_dig a)
{
    size_t i;

    for (i = 0; i < ALEN; i++)
        a[i]++;
}
```

```
# %rdi = a
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   jump to 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                    # rep相当于空操作，见3.6.4旁注
```

多维（嵌套型）数组

- 声明

  T A[*R*][*C*];

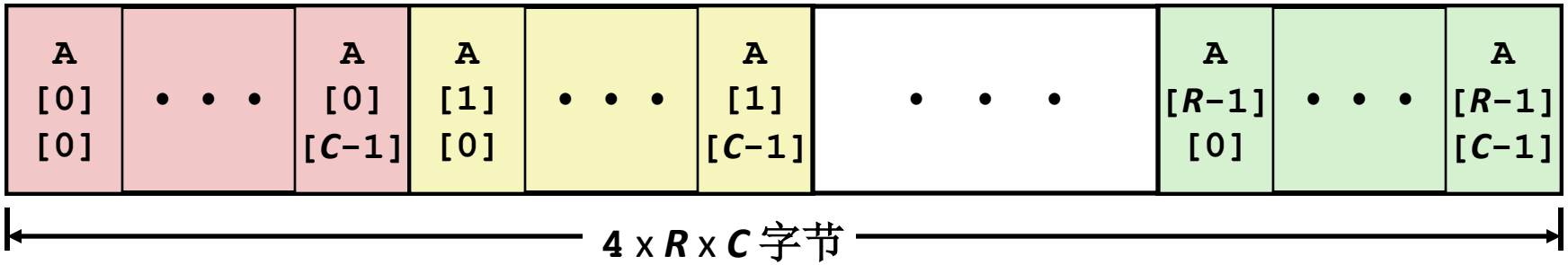
  - 类型为 *T* 的二维数组
  - *R* 行，*C* 列
  - 每个 *T* 型元素占 *K* 字节

- 数组大小
  - *R* x *C* x *K* 字节

- 内存分配
  - 按行存储

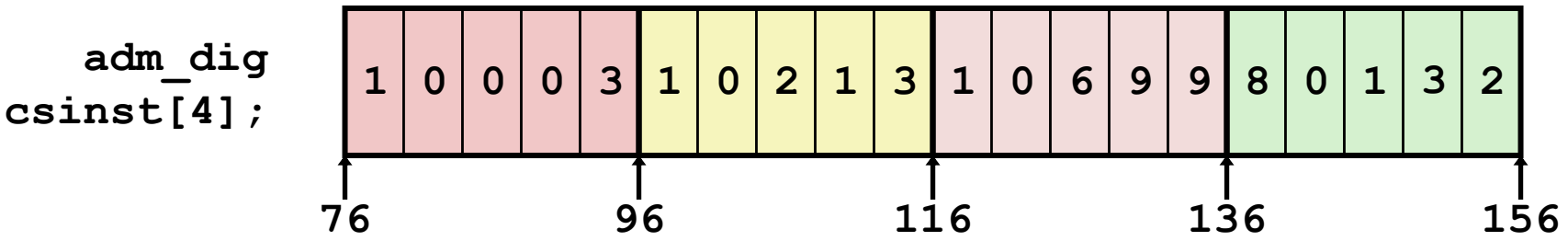
$$\begin{bmatrix} A[0][0] & \cdots & A[0][C-1] \\ \vdots & & \vdots \\ A[R-1][0] & \cdots & A[R-1][C-1] \end{bmatrix}$$

int A[*R*][*C*];



嵌套型数组举例

```
#define ICOUNT 4
adm_dig csinst[ICOUNT] =
  {{ 1, 0, 0, 0, 3 },
   { 1, 0, 2, 1, 3 },
   { 1, 0, 6, 9, 9 },
   { 8, 0, 1, 3, 2 }};
```

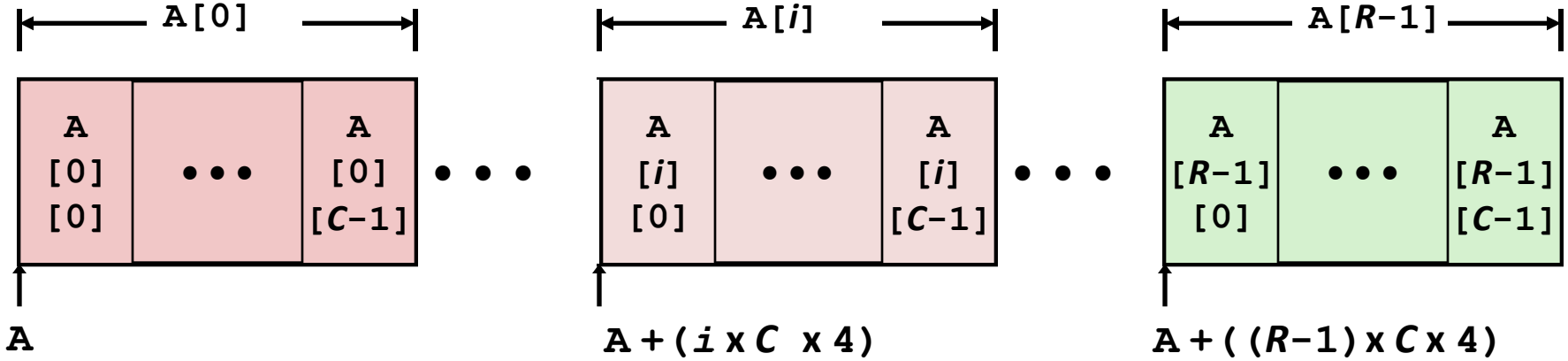


- “adm_dig csinst[4]”等同于“int csinst[4][5]”
  - csinst：含 4 个元素的数组（在内存中相互邻接）
  - 其中每个元素为含有 5 个 int 的数组（在内存中相互邻接）
- 所有元素在内存中均采用行优先的方式（按行存储）

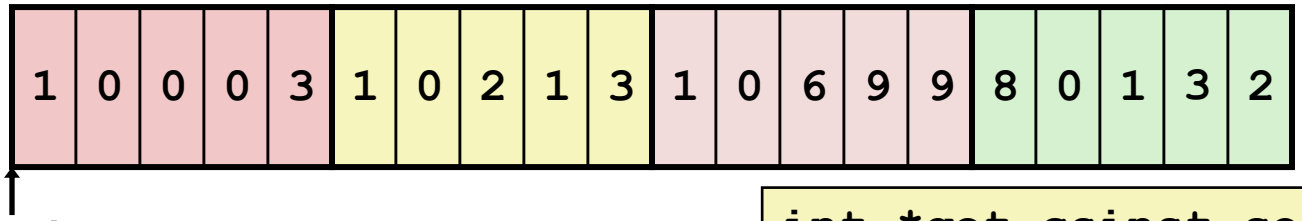
访问嵌套型数组的行向量

- 行向量
  - 设 A[*i*] 为一含有 *C* 个元素的数组
  - 每个类型为 *T* 的元素占用 *K* 个字节
  - 起始地址 A + *i* x (*C* x *K*)

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



访问嵌套型数组的行向量——举例




```
int *get_csinst_code(int index)
{
    return csinst[index];
}
```

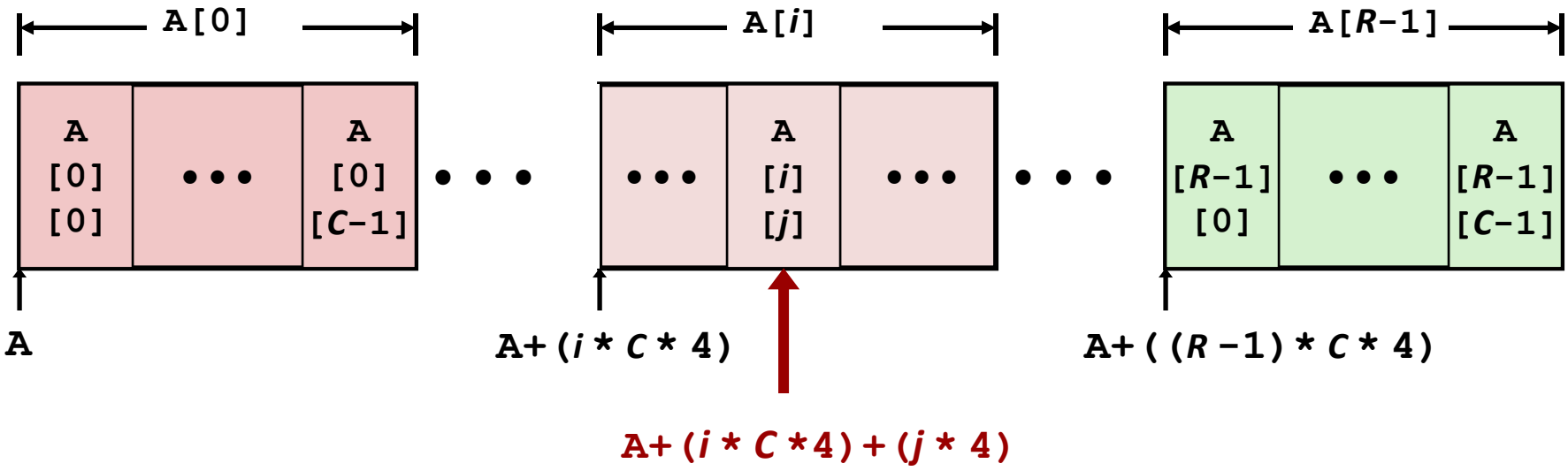
```
# %rdi = index
  leaq (%rdi,%rdi,4),%rax          # 5 x index
  leaq csinst(,%rax,4),%rax        # csinst + (20 x index)
```

- 行向量
  - `csinst[index]` 为含有 5 个 `int` 的数组
  - 起始地址为 `csinst + 20 x index`
- 机器码
  - 计算并返回地址
  - 结果为：`csinst + 4 * (index+4*index)`

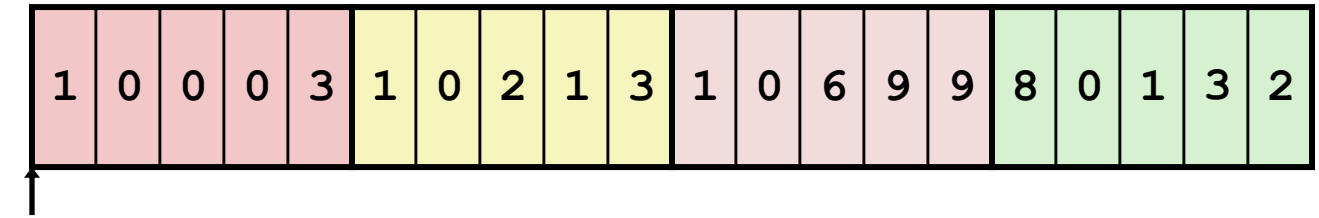
访问嵌套型数组的元素

- 数组元素
  - A[*i*] [*j*] 为类型为 *T* 的元素，占用 *K* 个字节
  - 地址为：A + *i* x (*C* x *K*) + *j* x *K* = A + (*i* x *C* + *j*) x *K*

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



访问嵌套型数组的元素——举例




```
int get_csinst_digit(int index, int dig)
{
    return csinst[index][dig];
}
```

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

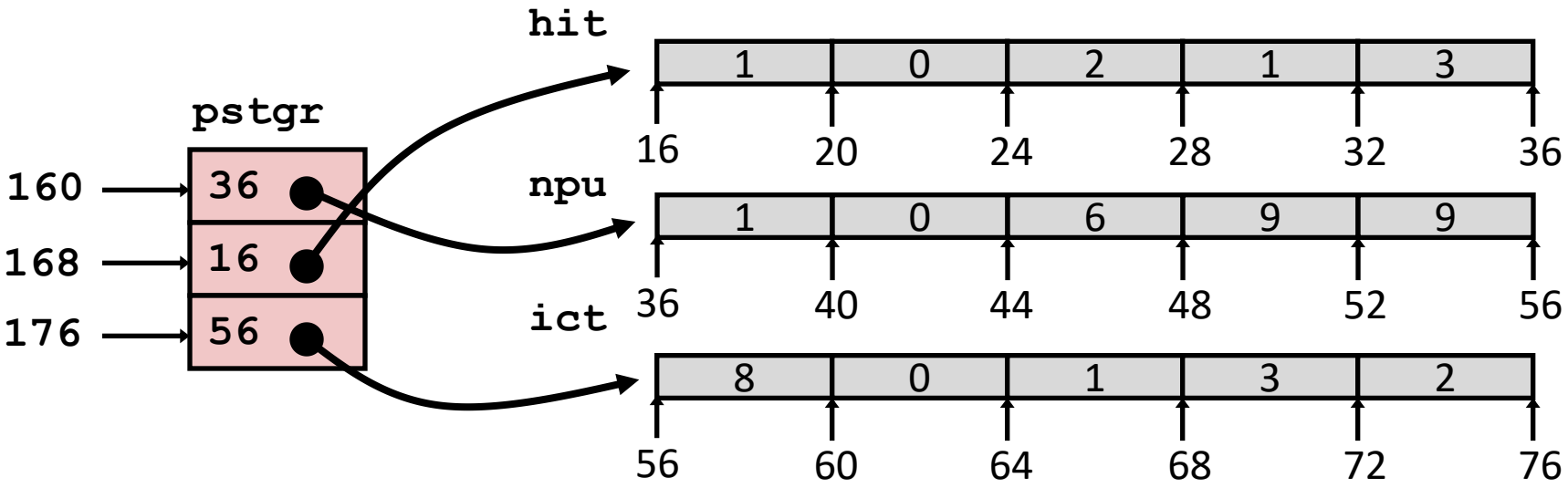
- 数组元素
  - csinst[index][dig] 为一 int 型变量
  - 地址：csinst + 20 x index + 4 x dig
    = csinst + 4 x (5 x index + dig)

指针（多级型）数组举例

```
adm_dig hit = { 1, 0, 2, 1, 3 };
adm_dig npu = { 1, 0, 6, 9, 9 };
adm_dig ict = { 8, 0, 1, 3, 2 };
```

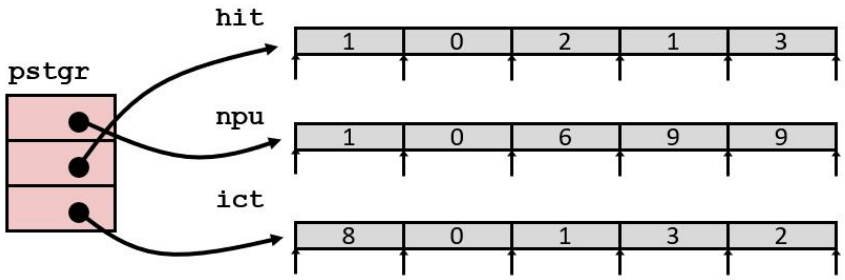
```
#define PCOUNT 3
int *pstgr[PCOUNT] = { hit, npu, ict };
```

- pstgr 为含有 3 个元素的数组
- 其元素为 int 型的指针
  - 每个占 8 字节
- 每个指针指向一个 int 型数组



访问多级型数组的元素

```
int get_pstgr_digit
    (size_t index, size_t dig)
{
    return pstgr[index][dig];
}
```




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

- 计算
  - 元素访问 M[M[pstgr + 8 x index] + 4 x dig]
  - 须进行 2 次内存读操作
    - 第一次：获取指向行数组的指针
    - 第二次：访问该数组中的元素

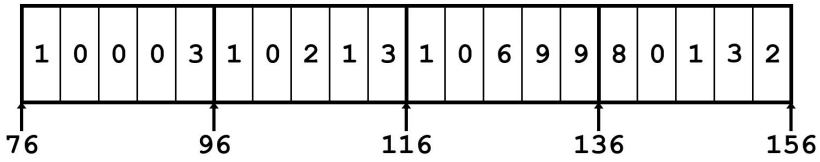
数组元素访问

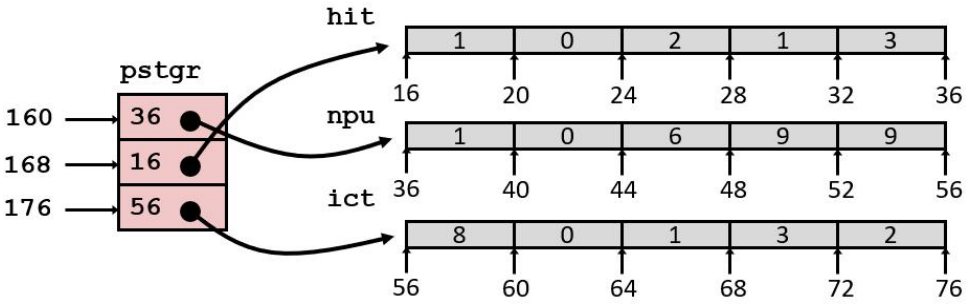
嵌套型数组

```
int get_csinst_digit
    (size_t index, size_t dig)
{
    return csinst[index][dig];
}
```

多级型数组

```
int get_pstgr_digit
    (size_t index, size_t dig)
{
    return pstgr[index][dig];
}
```





C 语句完全相同，但地址计算方法迥异：

```
M[csinst+20*index+4*dig]
```

VS

```
M[M[pstgr+8*index]+4*dig]
```

N x *N* 矩阵代码

- 固定维数
  - *N* 的值在编译时已经确定
- 可变维数
  - 实现动态数组的传统方法
  - 由主调函数固定数组的维数
- 可变维数
  - ISO C99 引入
  - 已获 GCC 支持

```
#define N 16
typedef int fix_matrix[N][N];
/* 获取元素 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))
/* 获取元素 a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j)
{
    return a[IDX(n, i, j)];
}
```

```
/* 获取元素 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 矩阵访问（固定维）

- 数组元素
  - 地址：A + *i* x (*C* x *K*) + *j* x *K*
  - *C* = 16, *K* = 4

```
/* 获取元素 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 x i
  addq  %rsi, %rdi              # a + 64 x i
  movl  (%rdi,%rdx,4), %eax     # M[a + 64 x i + 4 x j]
  ret
```

n x *n* 矩阵访问（可变维）

- 数组元素
  - 地址： A + *i* x (*C* x *K*) + *j* x *K*
  - *C* = *n*, *K* = 4
  - 须执行整数乘法

```
/* 获取元素 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 x i
  leaq    (%rsi,%rdi,4), %rax     # a + 4 x n x i
  movl    (%rax,%rcx,4), %eax     # a + 4 x n x i + 4 x j
  ret
```