# 数据

## 计算机系统基础

任课教师：
龚奕利
yiligong@whu.edu.cn

# 主要内容

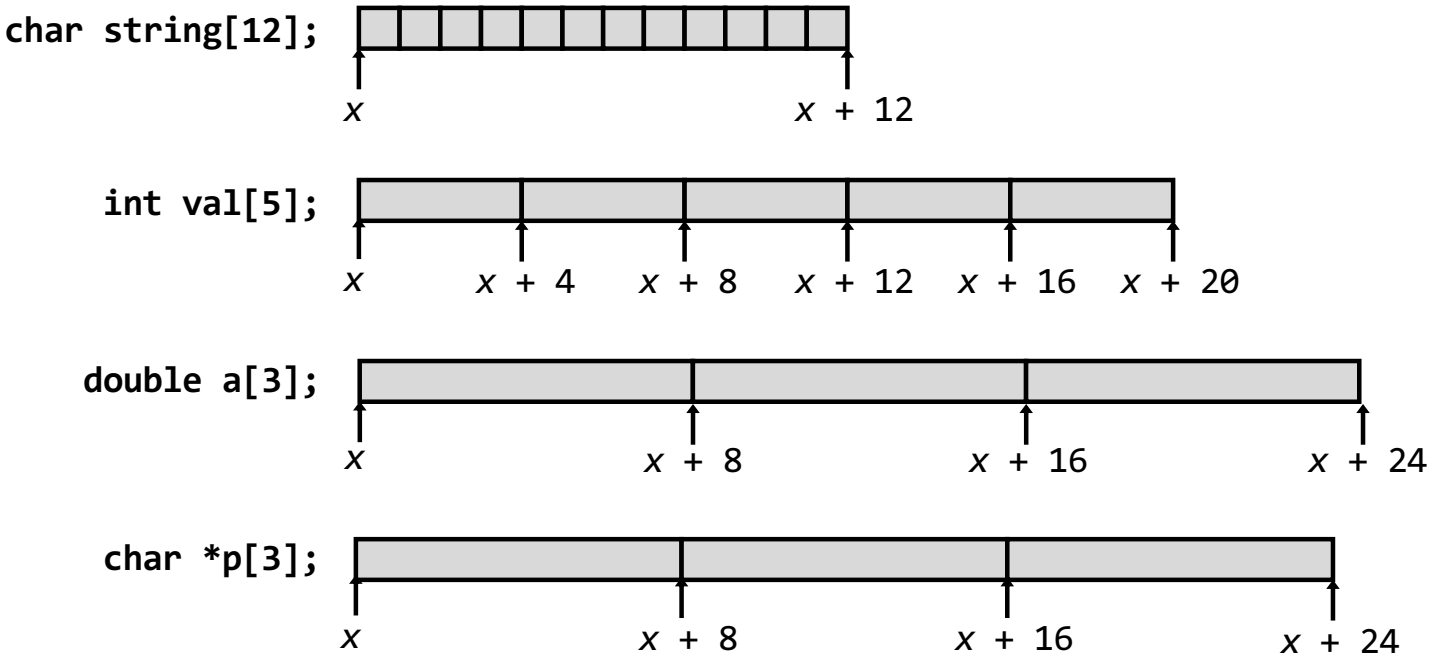
- 数组
  - 一维数组
  - 多维数组（嵌套）
  - 变长数组
- 结构体
  - 分配
  - 存取
  - 数据对齐
- 联合体

# 数组空间分配

- 基本原理

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

  - 数据类型为 *T* 长度为 *L*
  - 在内存中连续地分配 *L* * **sizeof(***T***)** 字节大小的空间

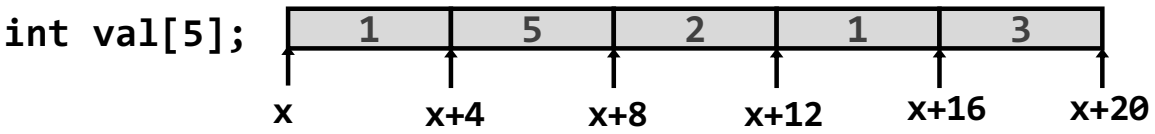


# 数组的访问

- 基本原则

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

  - 数据类型为 *T* 长度为 *L*
  - 标识符 **A** 可以被用作指向数组第**0**个元素的指针：类型 *T**

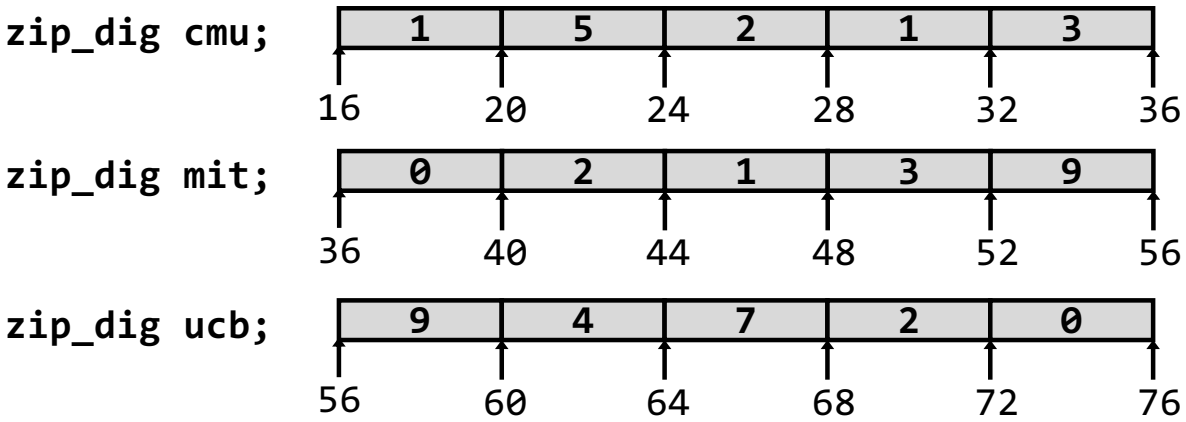



- 示例

| 示例 | 类型 | 值 |
|---|---|---|
| val[4] | int | 3 |
| val | int * | x |
| val+1 | int * | x + 4 |
| &val[2] | int * | x + 8 |
| val[5] | int | ?? |
| *(val+1) | int | 5 //val[1] |
| val + *i* | int * | x + 4 * i //&val[i] |

# 数组示例

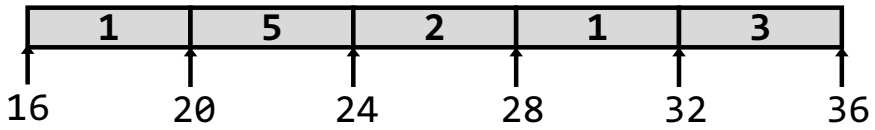
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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]**”
- 三个示例数组被分配到三个连续的**20**字节内存块中
  - 一般来说，不保证会这样分配

# 数组的访问示例

```
zip_dig cmu;
```




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

- 寄存器 **%rdi** 包含数组的起始地址
- 寄存器 **%rsi** 包含数组索引
- 期望的数据位于 **%rdi + 4*%rsi**
- 使用内存引用 **(%rdi,%rsi,4)**

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

# 数组循环示例

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

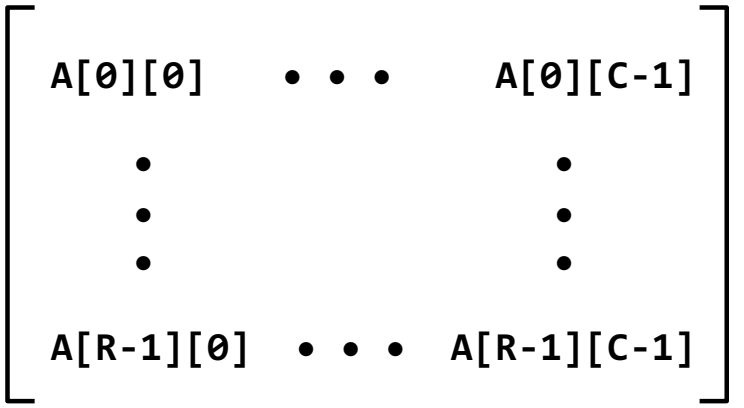
```
  # %rdi = z
  movl    $0, %eax            #  i = 0
  jmp     .L3                 #  goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $4, %rax            #   i:4
  jbe     .L4                 #   if <=, goto loop
  rep; ret
```

# 多维（嵌套）数组

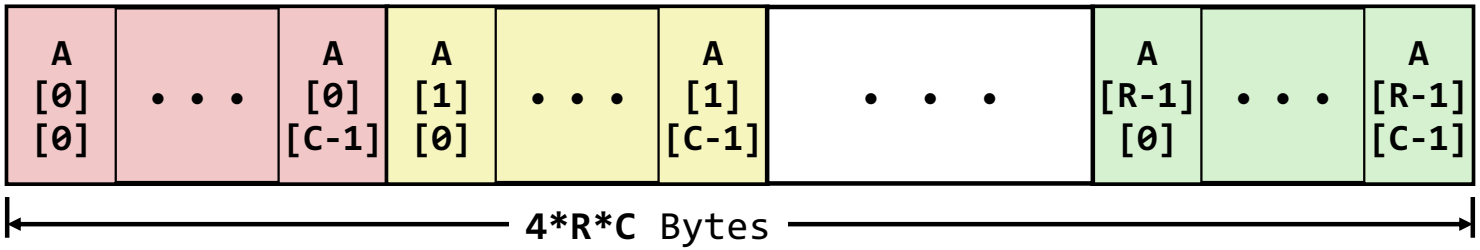
- 声明

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

  - 数据类型为 *T* 的二维数组
  - *R* 行，*C* 列
- 数组大小
  - *R* * *C* * **sizeof**(*T*) bytes
- 排列
  - 按行优先排序

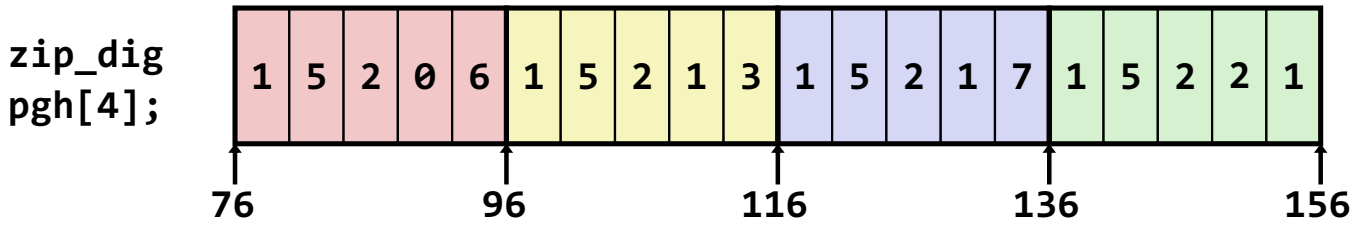



**int A[R][C];**



# 数组嵌套示例

```
#define PCOUNT 4
typedef int zip_dig[5];

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

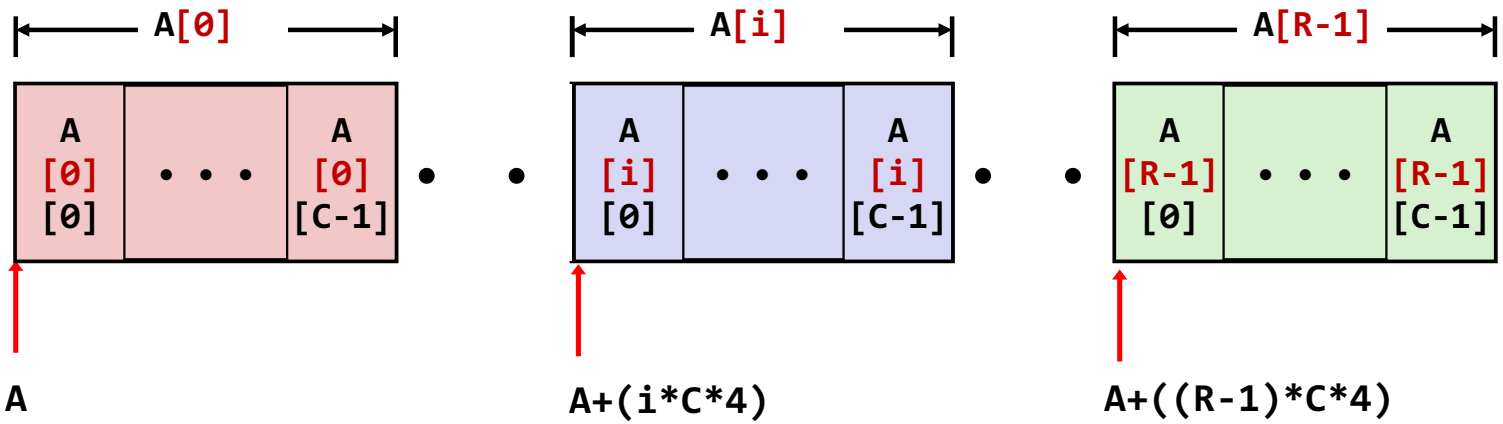



- “zip_dig pgh[4]” 等价于 “int pgh[4][5]”
  - 变量 pgh: 4个元素（20字节/元素）组成的数组，连续分配
  - 每个元素都是一个5个整型（4个字节/元素）的数组，连续分配
- 内存中所有的元素按“行优先”排序

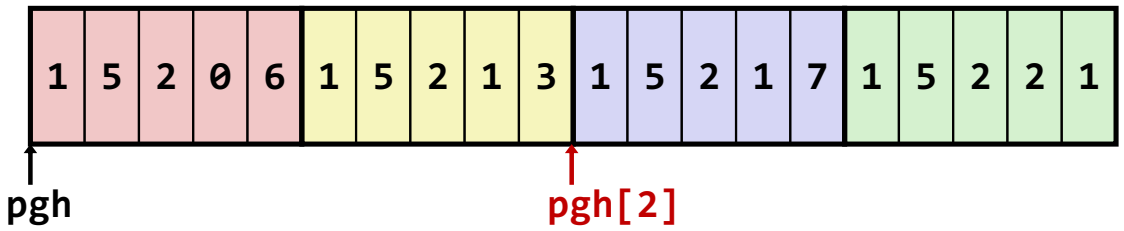
# 嵌套数组的行访问

- 多维数组中的行向量
  - **A[i]** 是类型为 *T* 的 *C* 个元素的数组
  - 起始地址为 **A + i * (C * sizeof(*T*))**

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



# 嵌套数组行访问代码




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

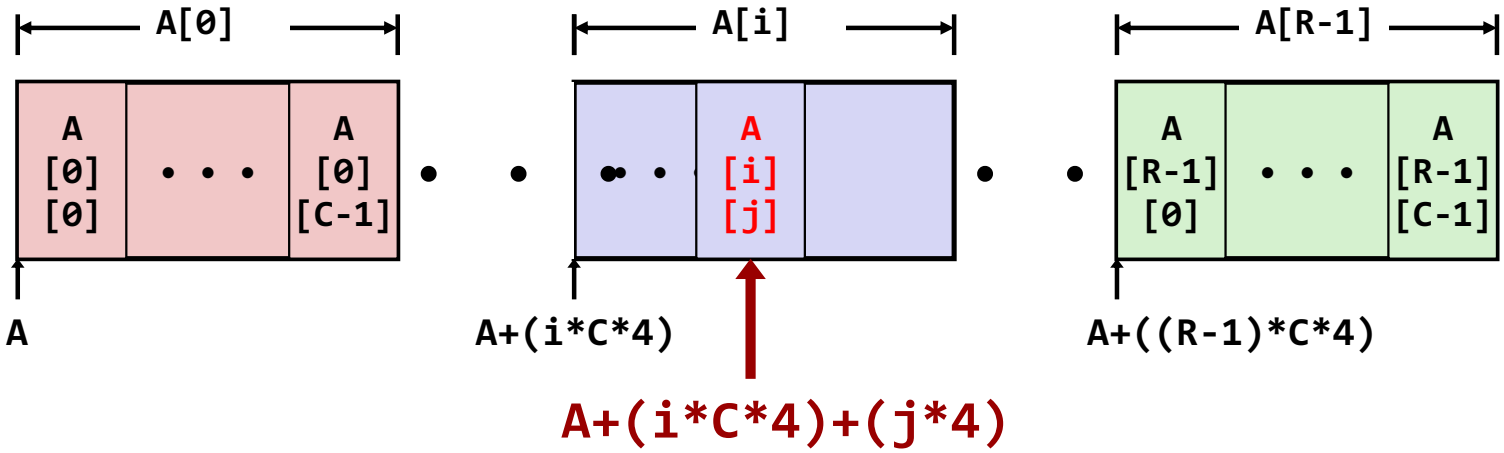
```
# %rdi = index
 leaq (%rdi,%rdi,4),%rax    # 5 * index
 leaq pgh(,%rax,4),%rax     # pgh + (20 * index)
```

- 行向量
  - **pgh[index]** 是一个 5 个整型数据的数组
  - 起始地址 **pgh+20*index**
- 机器码
  - 计算并返回地址
  - 计算 **pgh + 4*(index+4*index)**

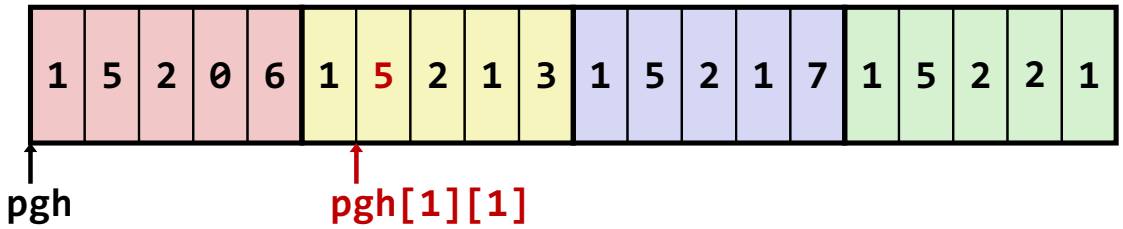
# 嵌套数组元素访问

- 数组元素
  - A[i][j] 是类型为 *T* 的元素，每个元素需要 *K* 个字节
  - A[i][j] 地址是 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)]
```

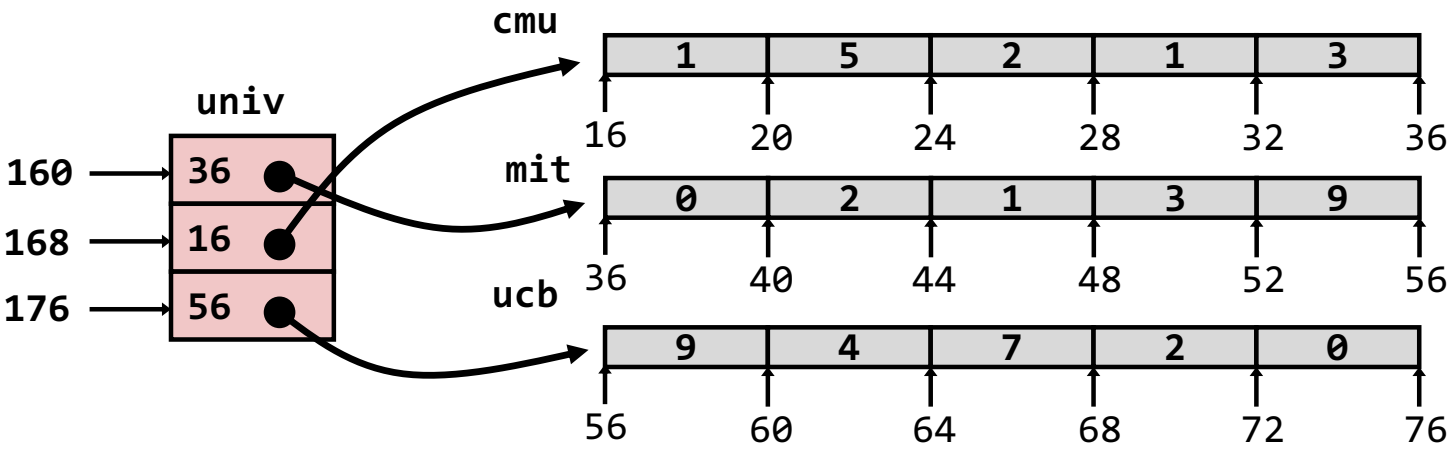
- 数组元素
  - **pgh[index][dig]** 是整型元素
  - 地址: **pgh + 20*index + 4*dig**
    **= pgh + 4*(5*index + dig)**

# 多级数组示例

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

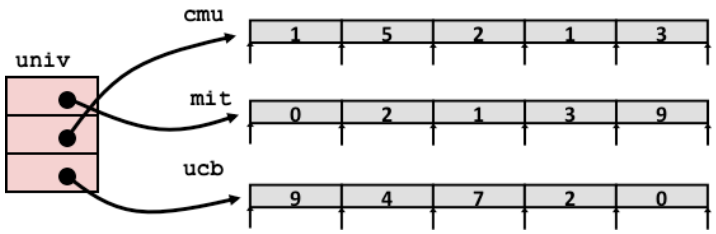
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- 变量 **univ** 表示有**3**个元素的数组
- 每个元素是指针类型
  - 8 个字节
- 每个指针指向一个整型数组



# 多级数组的元素访问

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



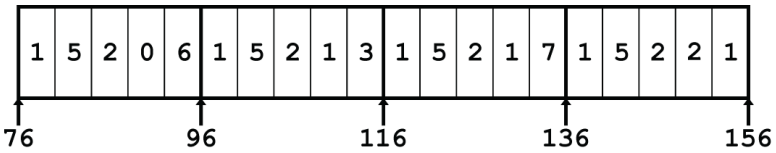

```
salq    $2, %rsi              # 4*digit
addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl    (%rsi), %eax          # return *p
ret
```

- 计算
  - 访问一个元素是要访问 **Mem[Mem[univ+8*index]+4*digit]**
  - 必须读取两次内存
    - 第一次获取指向行数组的指针
    - 第二次访问数组中的元素

# 访问数组元素

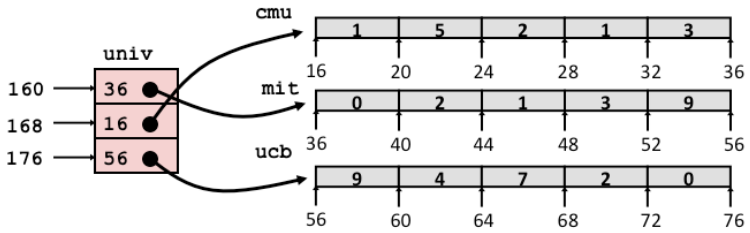
嵌套的数组

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




多级数组

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




在C语言中，访问看起来类似，但地址计算非常不同

**Mem[pgh+20*index+4*digit]**

**Mem[Mem[univ+8*index]+4*digit]**

# *N X N* 矩阵代码

- 固定的维数
  - 在编译时确定 *N* 的值

- 可变维度，显式索引
  - 实现动态数组的传统方法

- 可变维度，隐式索引
  - 现在 gcc 支持

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

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

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

# 16 X 16 矩阵访问

- 数组元素
  - `int A[i][j];`
  - 地址 `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   # Mem[A + 64*i + 4*j]
  ret
```

# *n* X *n* 矩阵访问

- 数组元素
  - size_t n;
  - int A[n][n];
  - 地址 A + i * (C * K) + j * K
  - C = n, K = 4
  - 必须执行整型乘法

```
/* 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

```
#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;
}
```

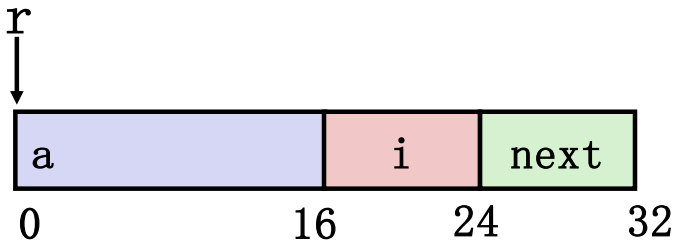
```
linux> ./array
result: 9
```

# 主要内容

- 数组
  - 一维数组
  - 多维数组（嵌套）
  - 变长数组
- 结构体
  - 分配
  - 存取
  - 数据对齐
- 联合体

# 结构描述

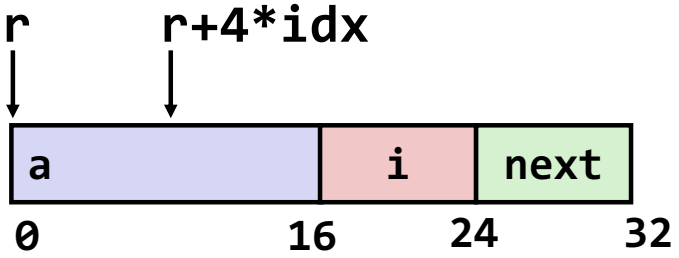
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```




- 结构是以一块连续的内存存储的
  - 大小可以容纳结构体所有的字段
- 字段按照声明的顺序排列
  - 即使另一种排列顺序可以产生更紧凑的表示
- 编译器决定字段的总体大小和位置
  - 机器级程序完全不知道源代码中的结构体

# 生成指向结构成员的指针

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




- 生成指向数组元素的指针
  - 在编译时确定每个结构成员的偏移量
  - 计算 **r + 4*idx**

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

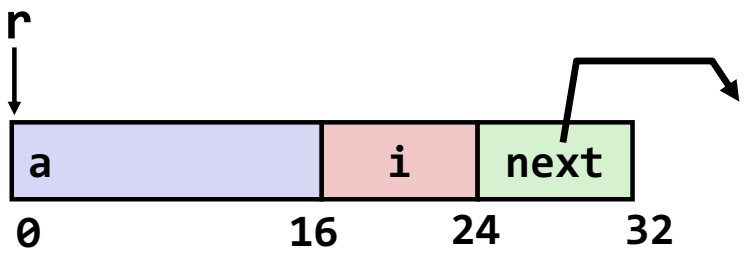
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

# 遍历链表 #1

- C 代码

```
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;
};
```




- 循环汇编代码

```
.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
```

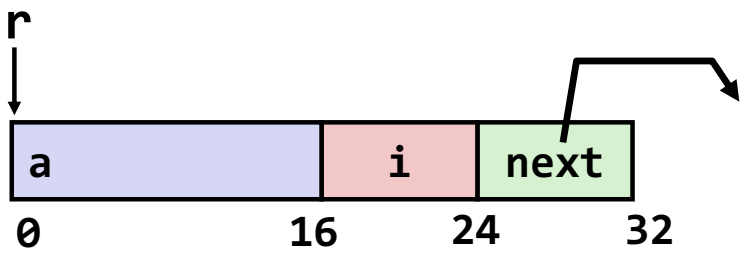
| Register | Value |
|---|---|
| %rdi | r |
| %rax | len |

# 遍历链表 #2

- C 代码

```
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;
};
```



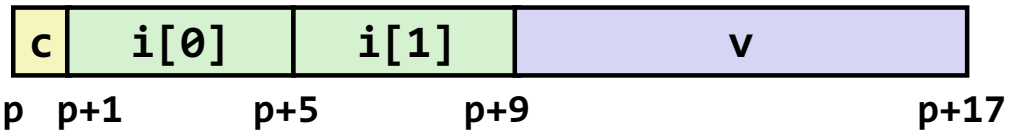

- 循环汇编代码

```
.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
```

| Register | Value |
|---|---|
| %rdi | r |
| %rax | len |

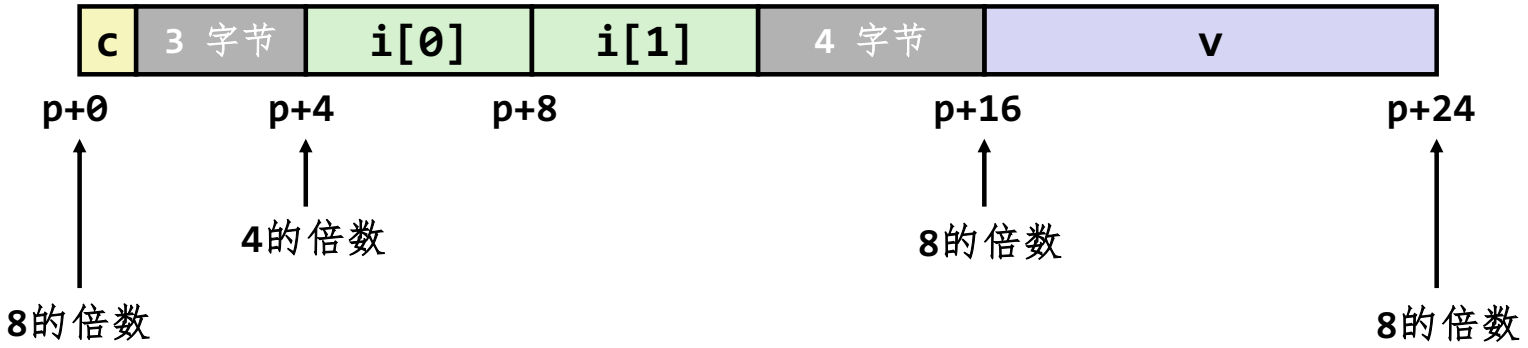
# 结构与对齐

- 非对齐（unaligned）的数据




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

- 对齐(aligned)的数据
  - 基本数据类型需要 K 字节表示，则地址必须是 K 的倍数



# 对齐原则

- **对齐的数据**
  - 基本数据类型需要 K 个字节
  - 地址必须是 K 的倍数
  - 在某些机器上严格要求满足；在 x86-64 上建议满足
- **数据对齐的目的**
  - 内存是以（对齐的）4 或 8 字节（取决于系统）为单位存取内存块的
    - 加载或存储块界限的数据，效率低
    - 虚拟内存中，数据跨越 2 个页面会更棘手
- **编译**
  - 在结构中插入缝隙，以确保字段的正确对齐

# 对齐的具体案例（x86-64）

- **1 字节：char，…**
  - 对地址无要求
- **2 字节：short，…**
  - 最低1位地址必须是 $0_2$
- **4 字节：int，float，…**
  - 最低2位地址必须是 $00_2$
- **8 字节：double，long，char *，…**
  - 最低3位地址必须是 $000_2$

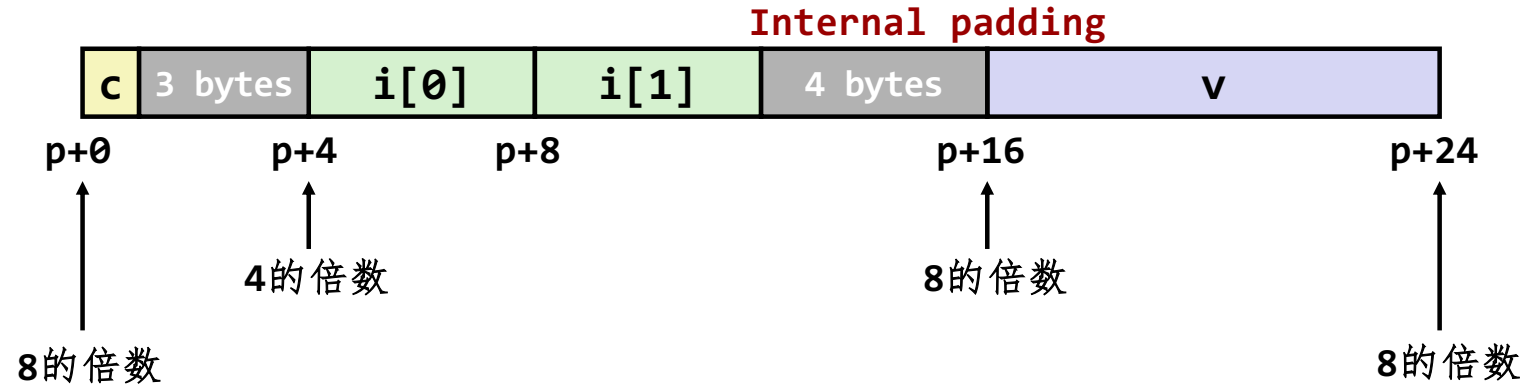
跳转表

```
.section   .rodata
  .align 8
.L4:
  .quad    .L8  # x = 0
  .quad    .L3  # x = 1
  .quad    .L5  # x = 2
  .quad    .L9  # x = 3
  .quad    .L8  # x = 4
  .quad    .L7  # x = 5
  .quad    .L7  # x = 6
```

# 结构体的对齐要求

- 结构体内部
  - 必须满足每个元素的对齐要求
- 总体结构放置
  - 每个结构都有对齐要求 K
    - K = 所有元素的最大对齐
  - 初始地址和结构长度必须是 K 的倍数
- 示例
  - K = 8, 因为是 double 类型元素

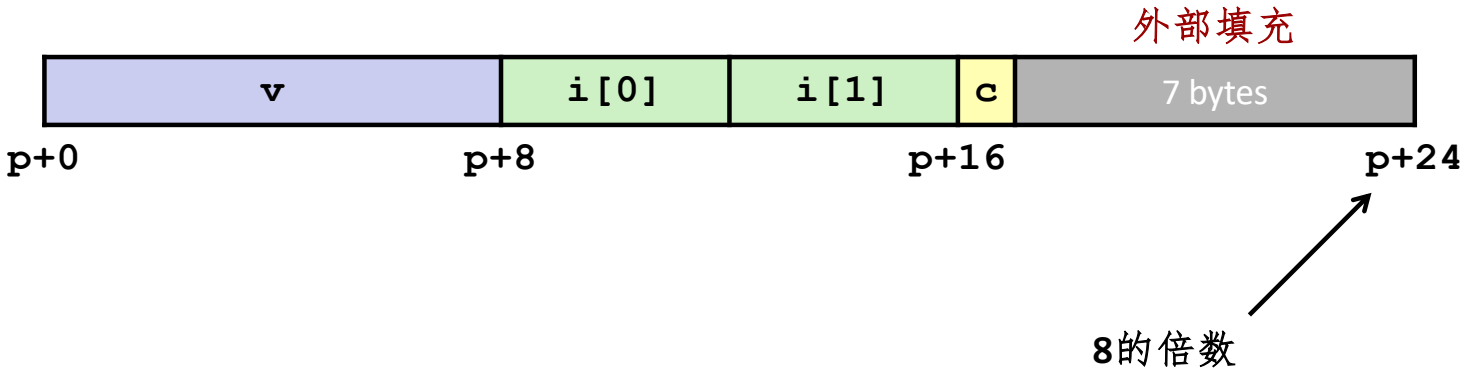
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



# 满足整体对齐要求

- 最大的对齐要求 **K**
- 整体结构必须是 **K** 的倍数

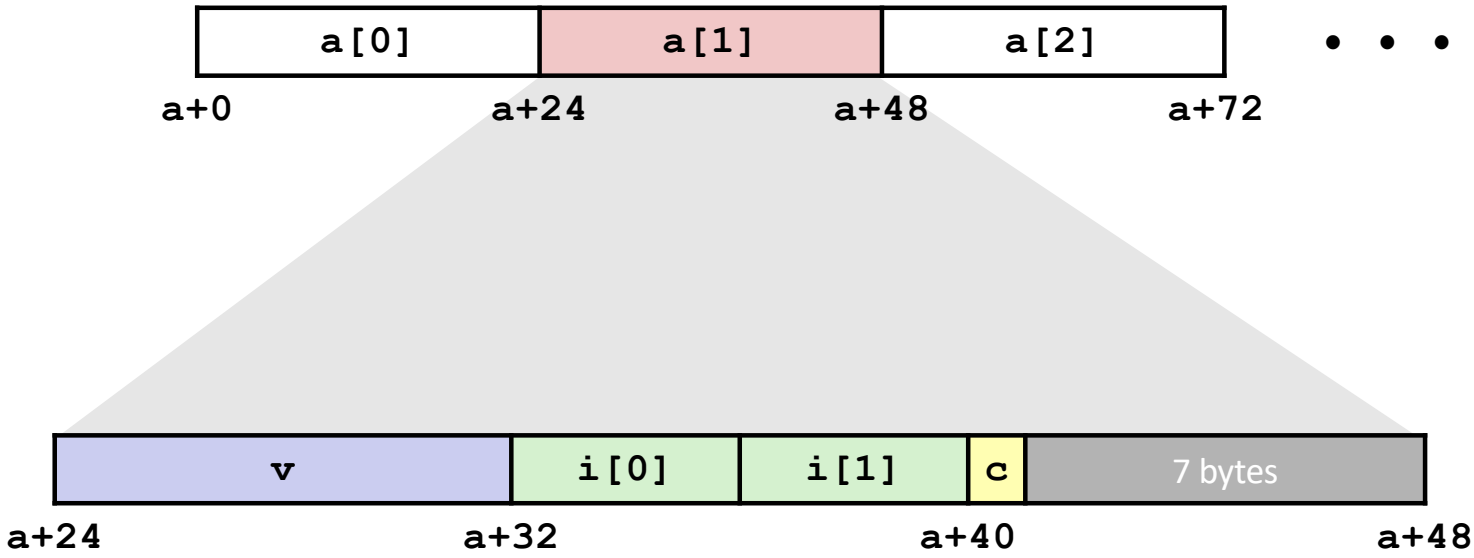
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



# 元素是结构体的数组

- 整体结构长度是 K 的倍数
- 满足每个基本元素的对齐要求

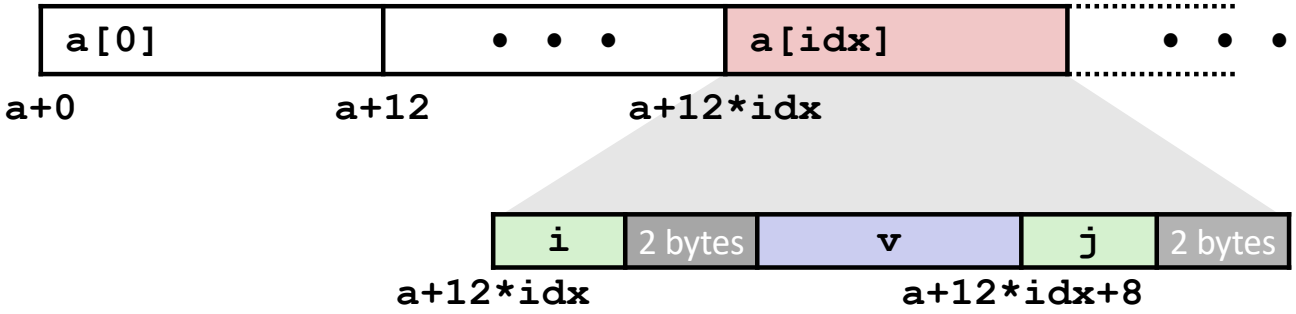
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



# 访问数组元素

- 计算数组偏移量 **12*idx**
  - sizeof(S3), 包括对齐所需的填充空间
- 元素 **j** 在结构中的偏移量为**8**
- 编译器给出偏移量 **a+8**

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




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

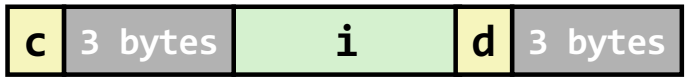
```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

# 节省空间

- 把大型数据类型放在前面

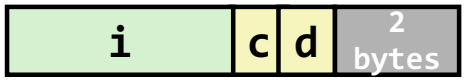
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

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




12 bytes

- 效果（最大对齐要求 K = 4）




8 bytes

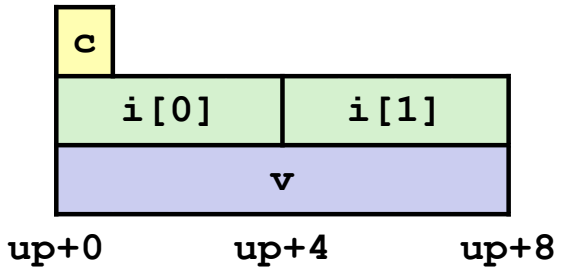
# 主要内容

- 数组
  - 一维数组
  - 多维数组（嵌套）
  - 变长数组
- 结构体
  - 分配
  - 存取
  - 数据对齐
- 联合体

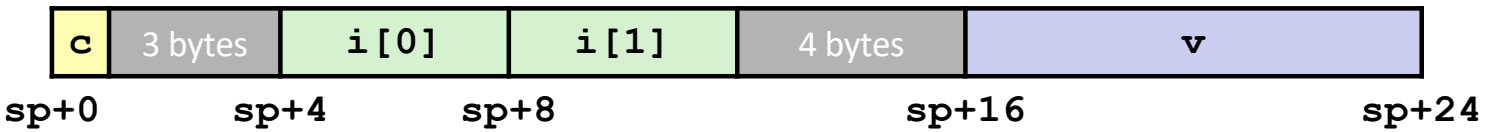
# 联合体分配

- 按最大元素分配
- 一次只能使用一个字段

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

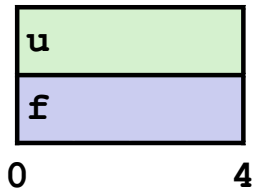



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



# 使用联合体访问位模式

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



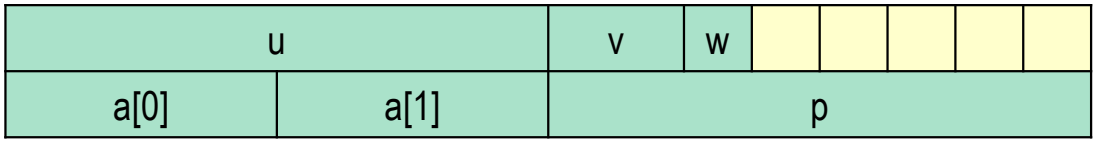

```
float bit2float(unsigned u) {
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

Same as (float) u ?

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

Same as (unsigned) f ?

# 练习题3.43



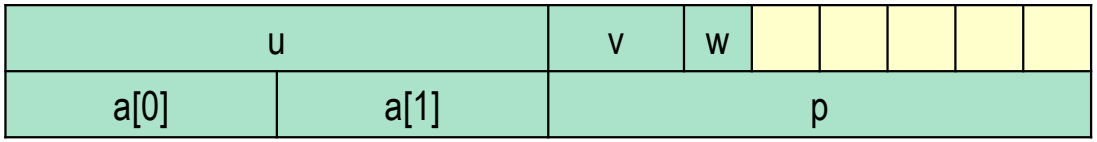

```
typedef union {
    struct {
        long                u;
        short               v;
        char                w;
    } t1;
    struct {
        int                 a[2];
        char                *p;
    } t2;
} u_type;

void get(u_type *up, type *dest) {
    *dest = expr;
}
```

| *expr* | *type* | Code |
|---|---|---|
| up->t1.u | | |
| up->t1.v | | |
| &up->t1.w | | |
| up->t2.a | | |
| up->t2.a[up->t1.u] | | |
| *up->t2.p | | |

| Instruction | | Effect | Description |
|---|---|---|---|
| MOV | *S*, *D* | *D* ← *S* | Move |
| movb | | | Move byte |
| movw | | | Move word |
| movl | | | Move double word |
| movq | | | Move quad word |
| movabsq | *I*, *R* | *R* ← *I* | Move absolute quad word |

**Figure 3.4** Simple data movement instructions.

# 练习题3.43



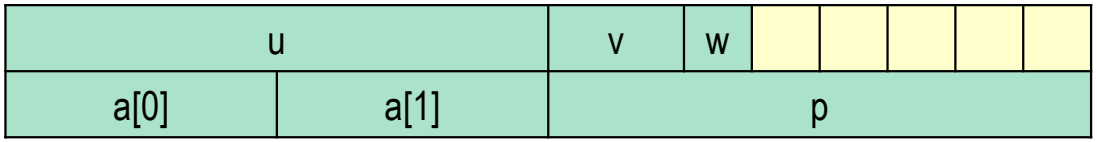

```
typedef union {
    struct {
        long                u;
        short               v;
        char                w;
    } t1;
    struct {
        int                 a[2];
        char                *p;
    } t2;
} u_type;

void get(u_type *up, type *dest) {
    *dest = expr;
}
```

| *expr* | *type* | Code |
|---|---|---|
| up->t1.u | long | |
| up->t1.v | short | |
| &up->t1.w | char * | |
| up->t2.a | int * | |
| up->t2.a[up->t1.u] | int | |
| *up->t2.p | char | |

| Instruction | | Effect | Description |
|---|---|---|---|
| MOV | $S, D$ | $D \leftarrow S$ | Move |
| movb | | | Move byte |
| movw | | | Move word |
| movl | | | Move double word |
| movq | | | Move quad word |
| movabsq | $I, R$ | $R \leftarrow I$ | Move absolute quad word |

**Figure 3.4** Simple data movement instructions.

# 练习题3.43




```
typedef union {
    struct {
        long                u;
        short               v;
        char                w;
    } t1;
    struct {
        int                 a[2];
        char                *p;
    } t2;
} u_type;

void get(u_type *up, type *dest) {
    *dest = expr;
}
```

| *expr* | *type* | Code |
|---|---|---|
| up->t1.u | long | movq (%rdi), %rax<br>movq %rax, (%rsi) |
| up->t1.v | short | movw 8(%rdi), %ax<br>movw %ax, (%rsi) |
| &up->t1.w | char * | lea 10(%rdi), %rax<br>movq %rax, (%rsi) |
| up->t2.a | int * | movq %rdi, (%rsi) |
| up->t2.a[up->t1.u] | int | movq (%rdi), %rax<br>movl (%rdi, %rax, 4), %eax<br>movl %eax, (%rsi) |
| *up->t2.p | char | movb 8(%rdi), %al<br>movb %al, (%rsi) |

| Instruction | | Effect | Description |
|---|---|---|---|
| MOV | $S, D$ | $D \leftarrow S$ | Move |
| movb | | | Move byte |
| movw | | | Move word |
| movl | | | Move double word |
| movq | | | Move quad word |
| movabsq | $I, R$ | $R \leftarrow I$ | Move absolute quad word |

**Figure 3.4** Simple data movement instructions.