汇编语言程序设计

80X86汇编语言与C语言-4

C程序在硬件层面的表示

- 数据
  - 整数（第二讲）
  - 浮点数（第三讲）
  - 数组、结构（第八讲）
- 代码
  - 基本概念/基本指令/寻址方式（第五讲）
  - 程序控制流与相关指令（第六讲）
  - 函数调用与相关指令（第七讲）

```
int array[2] = {1, 2};
int main()
{
    int val = sum(array, 2);
    return val;
}                          main.c
```

编译

```
0000000000000000 <array>:
  0:   01 00              add    %eax,(%rax)
  2:   00 00              add    %al,(%rax)
  4:   02 00              add    (%rax),%al
0000000000000000 <main>:
  0:   48 83 ec 08        sub    $0x8,%rsp
  4:   be 02 00 00 00     mov    $0x2,%esi
  9:   bf 00 00 00 00     mov    $0x0,%edi
                   a: R_X86_64_32 array
  e:   e8 00 00 00 00     callq  13 <main+0x13>
                   f: R_X86_64_PC32 sum-0x4
 13:   48 83 c4 08        add    $0x8,%rsp
 17:   c3                 retq          main.o
```

汇编指令

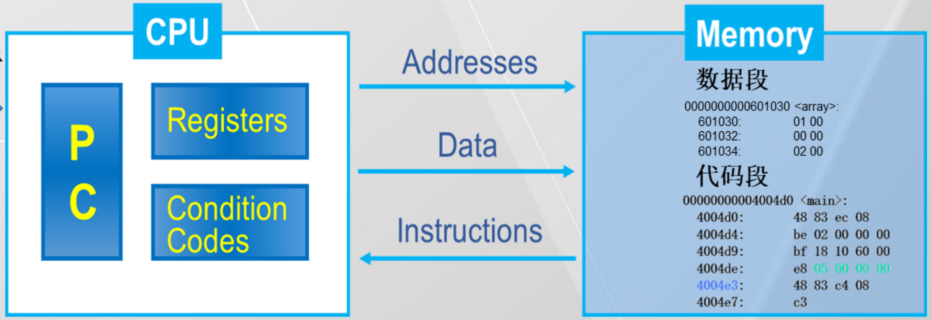
机器指令

链接

内存地址

```
00000000004004d0 <main>:
  4004d0:     48 83 ec 08
  4004d4:     be 02 00 00 00
  4004d9:     bf 18 10 60 00
  4004de:     e8 05 00 00 00
  4004e3:     48 83 c4 08
  4004e7:     c3
00000000004004e8 <sum>:
  4004e8:     b8 00 00 00 00
  4004ed:     ba 00 00 00 00
  4004f2:     eb 09
  4004f4:     48 63 ca
  4004f7:     03 04 8f
  4004fa:     83 c2 01
  4004fd:     39 f2
  4004ff:     7c f3
  400501:     f3 c3      #<array>没有给出
```

运行



程序在机器层面的表示与运行

基本数据类型

- **整型**

  如何区分无符号数与符号数?

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

- **浮点数**

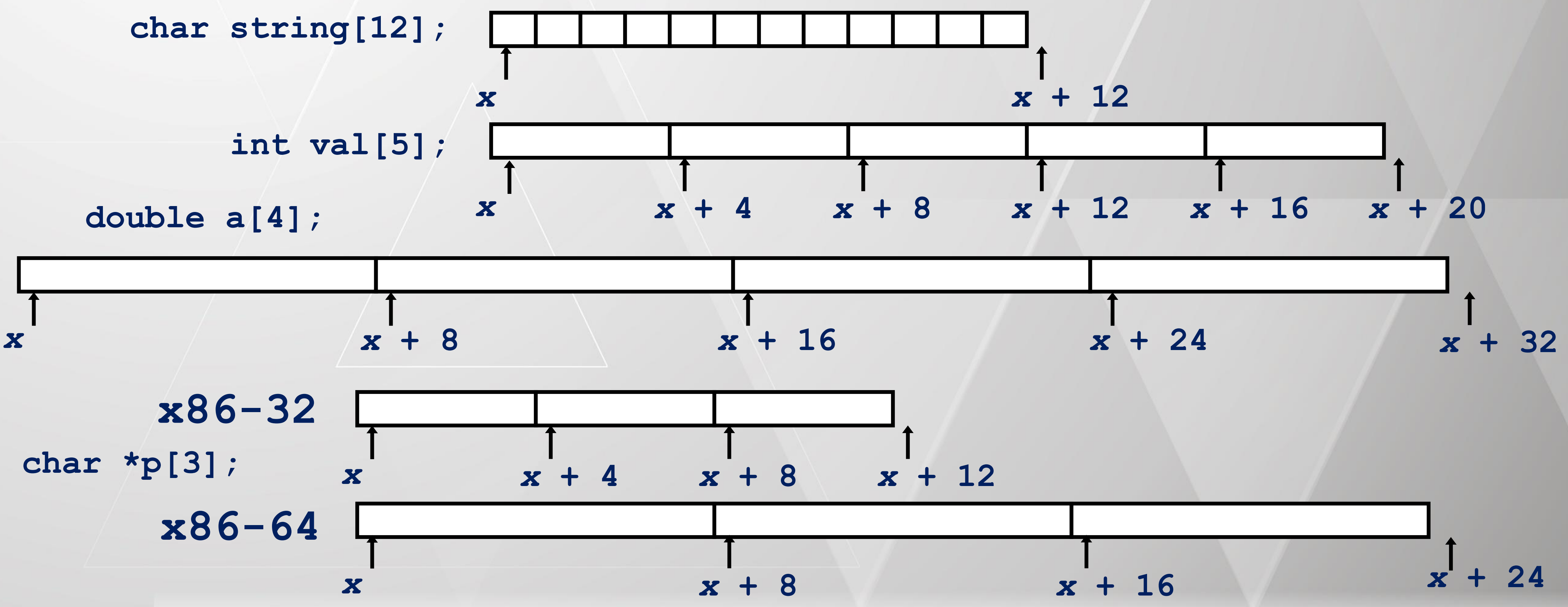
Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12/16	long double

数组的内存存储

- **基本原则**

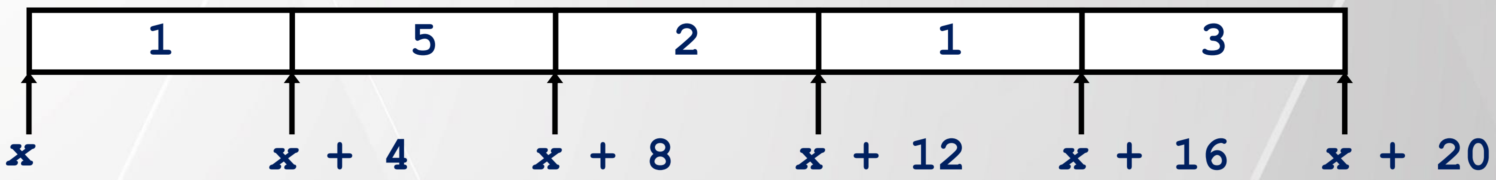
  T **A[*L*];**

  - 基本数据类型：***T*** ；　　数组长度：***L***
  - 连续存储在大小为 ***L* * sizeof(*T*)** 字节的空间内



数组访问

```
int val[5];
```



- Reference Type Value

Reference	Type	Value
`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 + i`	`int *`	*x* + 4 *i*

数组访问示例

```
zip_dig cmu;
```

1	5	2	1	3

16 20 24 28 32 36

#define zip_dig int[5]

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

X86-64

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

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

数组循环示例

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

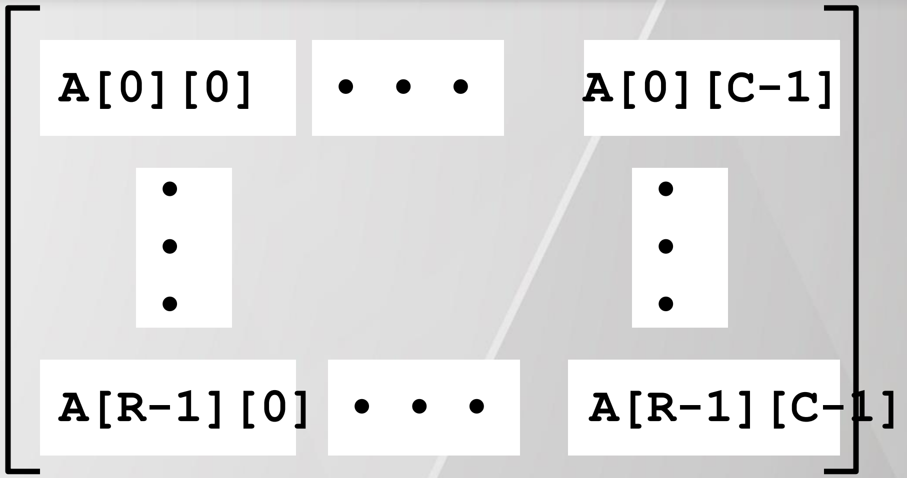
```
  # %rdi = z
  movl    $0, %eax          #   i = 0
  jmp     .L3               #   goto middle
.L4:                        # loop:
  addl    $1, (%rdi,%rax,4) #   z[i]++
  addq    $1, %rax          #   i++
.L3:                        # middle
  cmpq    $4, %rax          #   i:4
  jbe     .L4    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

嵌套数组（二维数组）

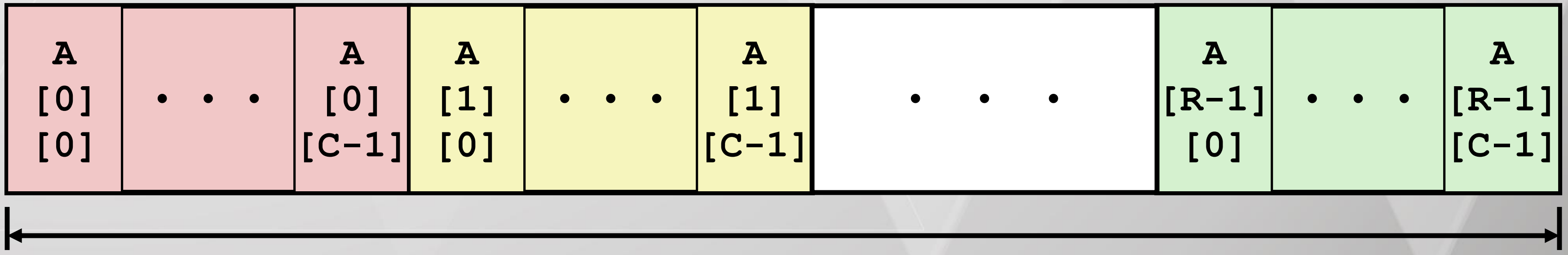
- **Declaration**

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

  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- **Array Size**
  - *R* * *C* * *K* bytes
- **Arrangement**
  - Row-Major Ordering



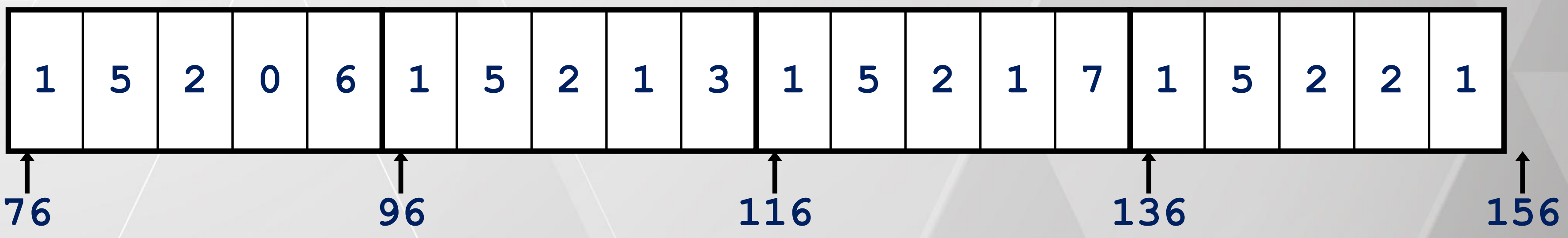

```
int A[R][C];
```



嵌套数组示例

```c
#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 }};
```

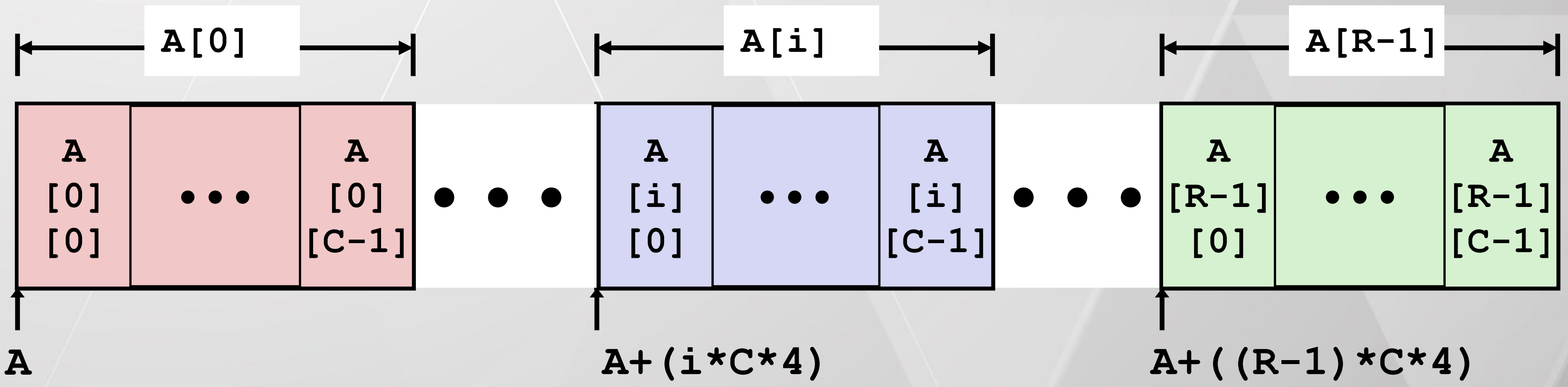
```c
zip_dig
pgh[4];
```



访问嵌套数组中的“行”

- Row Vectors
  - `A[i]` is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address `A` + *i* * (*C* * *K*)

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



访问嵌套数组中的“行”示例

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

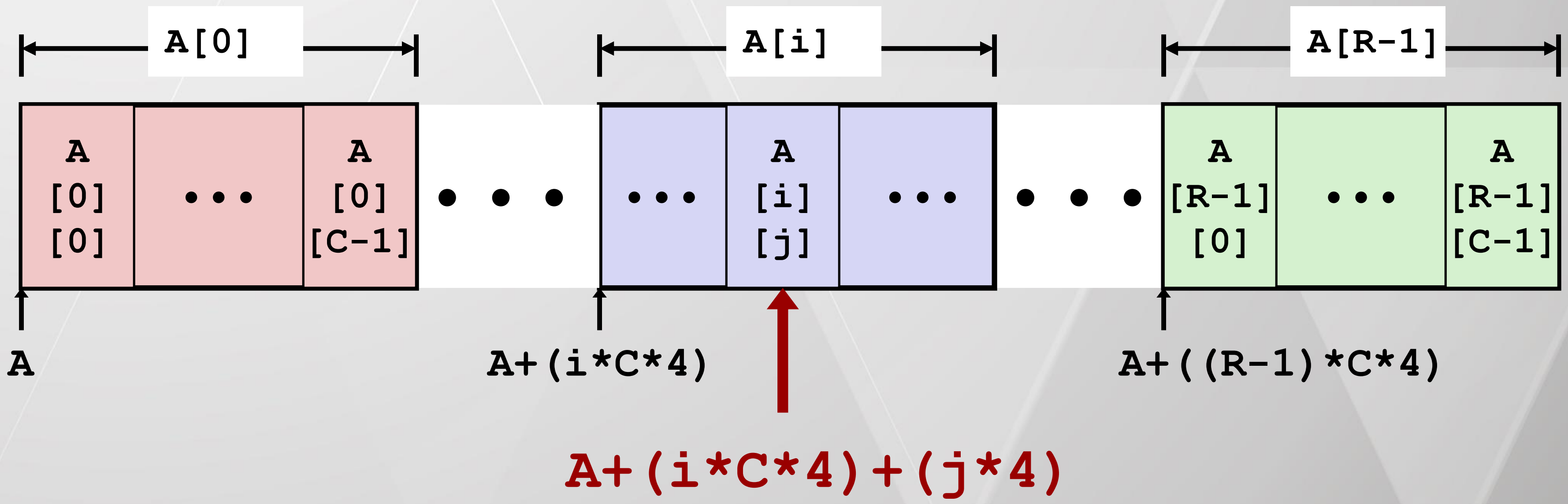
- 行地址计算
  - pgh[index]的数据类型是int[5]，即pgh每个元素的大小是 5*sizeof(int) = 20
  - 因此行地址是 pgh + (20 * index)
- 相关汇编代码
  - pgh + 4*(index+4*index)

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

访问嵌套数组的单个元素

- **Array Elements**
  - `A[i][j]` is element of type *T*, which requires *K* bytes
  - Address `A` + $i * (C * K) + j * K = A + (i * C + j) * K$

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



访问嵌套数组的单个元素—示例

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1

pgh

```
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)]
```

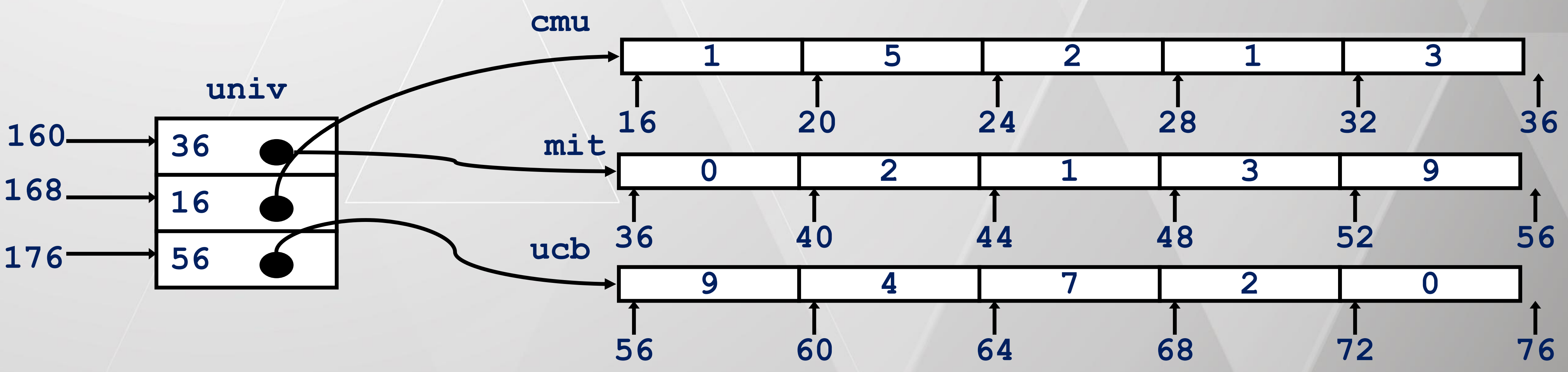
- **Array Elements**
  - `pgh[index][dig]` is `int`
  - Address: `pgh + 20*index + 4*dig`
    - `= pgh + 4*(5*index + dig)`

Multi-Level Array

- 变量**univ** 是一个指针数组，数组长度为3，数组元素长度为8字节
- 每个指针指向一个整数数组

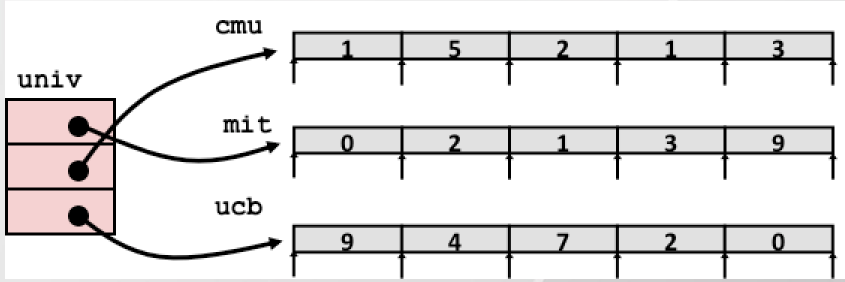
```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

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



访问Multi-Level Array中的元素

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



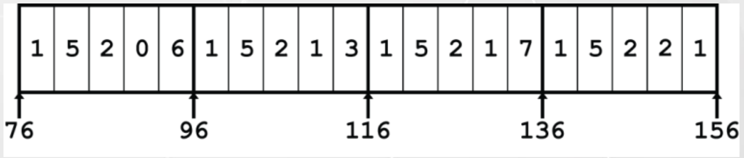

```
salq    $2, %rsi             # 4*digit
addq    univ(,%rdi,8), %rsi  # p = univ[index] + 4*digit
movl    (%rsi), %eax         # return *p
ret
```

- 数组元素的地址计算
  - **Mem[Mem[univ+8*index]+4*dig]**
  - 至少进行两次内存读取
    - 首先获得行地址
    - 再访问该行中的元素

与嵌套数组访问的不同

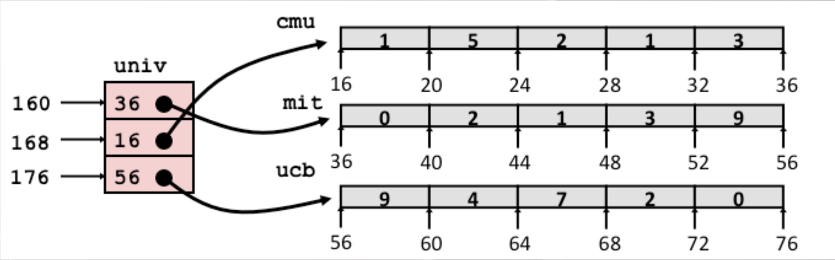
Nested array

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



Multi-level array

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



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

`Mem[pgh+20*index+4*digit]` `Mem[Mem[univ+8*index]+4*digit]`

N X N Matrix Code

Fixed dimensions

- 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, int i, int j)
{
  return a[i][j];
}
```

Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

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

Variable dimensions, implicit indexing

- Now supported by gcc (C99标准)

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

16 X 16 Matrix Access

- **Array Elements**
  - Address $A + i * (C * K) + j * K$
  - C = 16, K = 4

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, 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 Matrix Access

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

```
/* 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
```

Optimizing Fixed Array Access


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

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

Computation

- Step through all elements in column j

Optimization

- Retrieving successive elements from single column

Optimization

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

Register	Value
%rax	ajp
%rdx	dest

```
fix_column:
        movslq  %esi, %rcx #j
        salq    $2, %rcx   #j*4
        leaq    (%rdi,%rcx), %rax   #a+4*j
        leaq    1024(%rdi,%rcx),%rsi#结尾地址
.L2:
        movl    (%rax), %ecx
        addq    $64, %rax
        addq    $4, %rdx
        movl    %ecx, -4(%rdx)
        cmpq    %rsi, %rax
        jne     .L2
        ret
```

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

gcc -O2

Optimizing Variable Array Access

ZF set when a&b == 0
SF set when a&b < 0

- Compute ajp = &a[i][j]
  - Initially = a + 4*j
  - Increment by 4*n

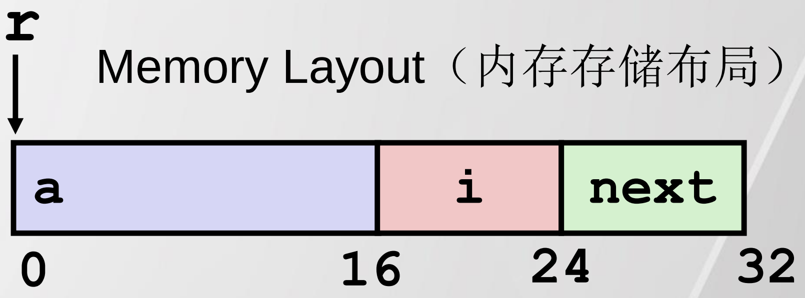
```
/* Retrieve column j from array */
void var_column
  (int n, int a[n][n],
   int j, int *dest)
{
  int i;
  for (i = 0; i < n; i++)
    dest[i] = a[i][j];
}
```

```
var_column:
        testl   %edi, %edi #n
        movslq  %edi, %r8  #r8 = n
        jle     .L1        #n<=0?

        movslq  %edx, %rdx #rdx = j
        salq    $2, %r8    #n*4   ①
        leaq    (%rsi,%rdx,4), %rax #a+4*j ③
        leal    -1(%rdi), %edx #n-1
        leaq    4(%rcx,%rdx,4),%rsi #dest+4*(n-1)+4
.L3:
        movl    (%rax), %edx  #④
        addq    $4, %rcx
        addq    %r8, %rax  #②
        movl    %edx, -4(%rcx)
        cmpq    %rsi, %rcx
        jne     .L3
.L1:
        ret
```

结构

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



连续分配的内存区域，内部各个域通过名字访问；

各个域依声明顺序存放；

结构的整体大小与各个域的位置布局由编译器决定

- 汇编层面不了解结构/域等信息

数据存储位置对齐

- 对齐的一般原则
  - 已知某种基本数据类型的大小为 K 字节
  - 那么，其存储地址必须是K的整数倍
  - X86-64的对齐要求基本上就是这样
- 为何需要对齐
  - 计算机访问内存一般是以内存块为单位的，块的大小是地址对齐的，如4、8、16字节对齐等
    - 如果数据访问地址跨越“块”边界会引起额外的内存访问
- 编译器的工作
  - 在结构的各个元素间插入额外空间来满足不同元素的对齐要求

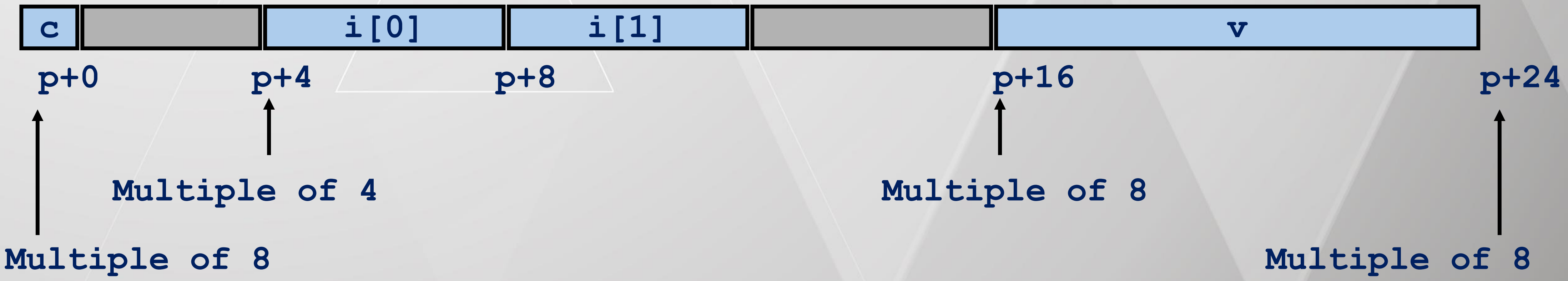
X86-64下不同元素的对齐要求

- 基本数据类型：
  - 1 byte (e.g., char)
    - N/A
  - 2 bytes (e.g., short)
    - 地址最后一位为0 / 2对齐
  - 4 bytes (e.g., int, float)
    - 地址最后二位为0 / 4对齐
  - 8 bytes (e.g., double, char *)
    - 地址最后三位为0 / 8对齐
  - 16 bytes (long double)
    - 地址最后四位为0 / 16对齐

结构的存储对齐要求

- 必须满足结构中各个元素的对齐要求
- 结构自身的对齐要求等同于其各个元素中对齐要求最高的那个，设为K字节
- 结构的起始地址与结构长度必须是K的整数倍
- 示例：
  - **K = 8**

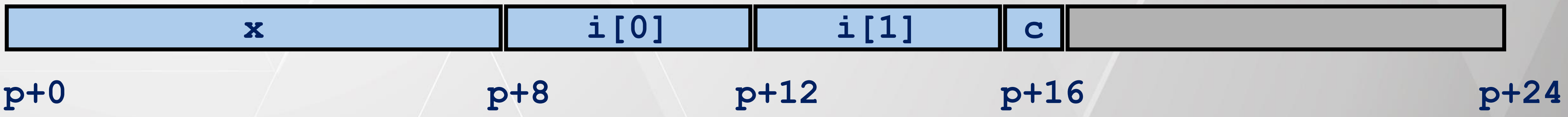
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



结构自身的对齐要求

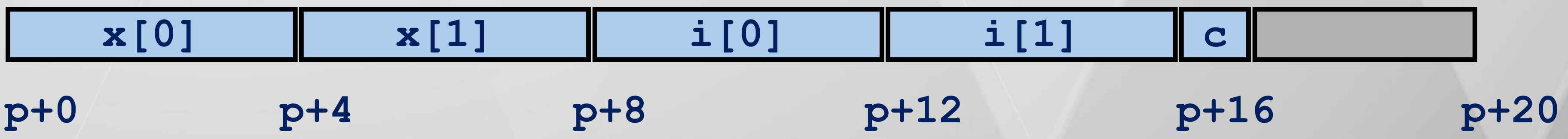
```c
struct S2 {
  double x;
  int i[2];
  char c;
} *p;
```

p必须是8的整数倍：



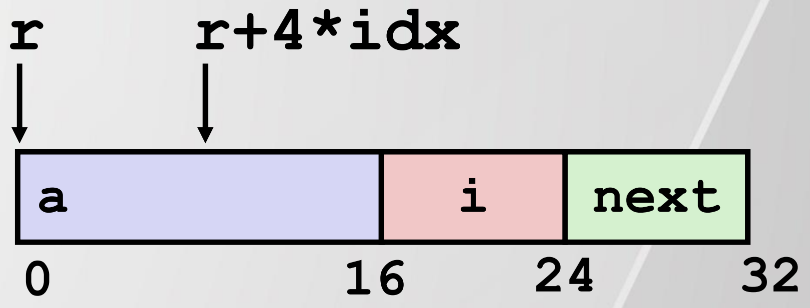

```c
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

p必须是4的整数倍



结构中各个域的地址

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



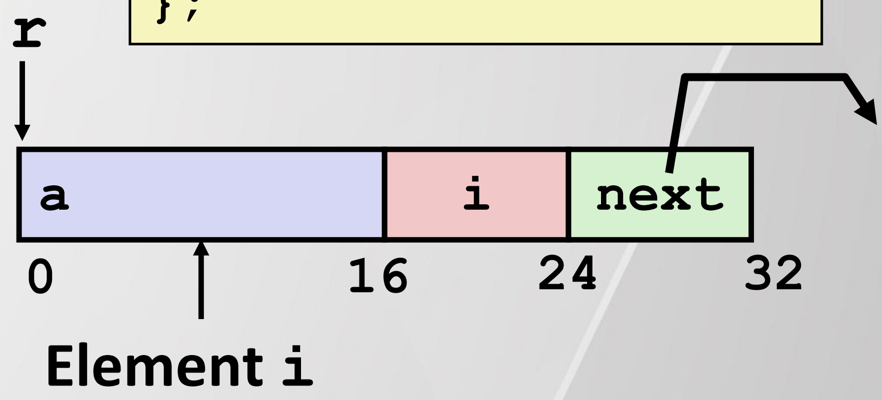
- 每个元素在结构中的相对地址在编译时就已确定
  - 地址为r + 4*idx

```
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
```

链表遍历

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




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

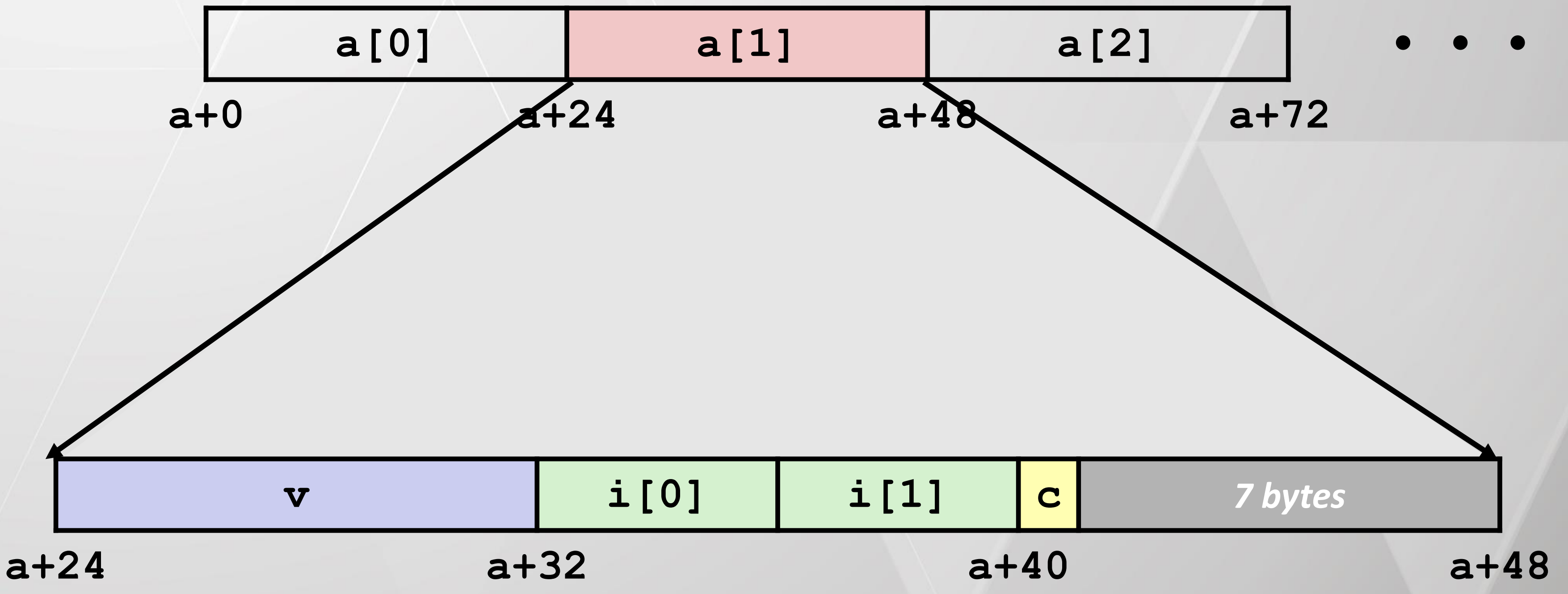
Register	Value
%rdi	r
%rsi	val

```
.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
```

结构数组

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

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



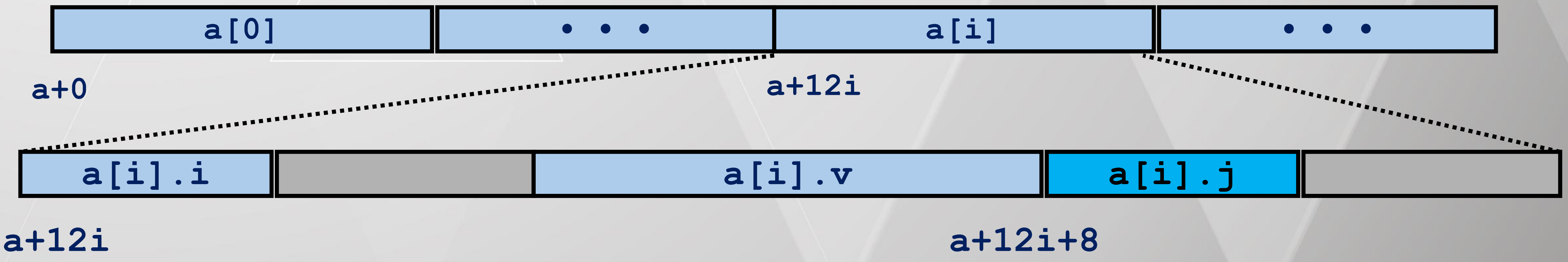
访问结构数组中的元素

- 首先计算数组元素（即结构）的地址
  - $12*i = 4*(i+2i)$
- 然后访问该结构中的元素
  - 偏移量为8

```c
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

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

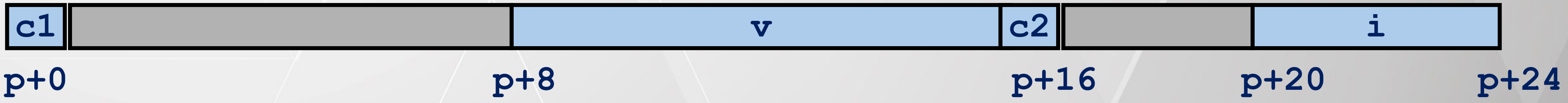
```
# %rdi= idx
leal  (%rdi,%rdi,2),%rax # 3*idx
movzwl  a+8(,%rax,4),%eax
```



结构内元素不同的先后顺序...

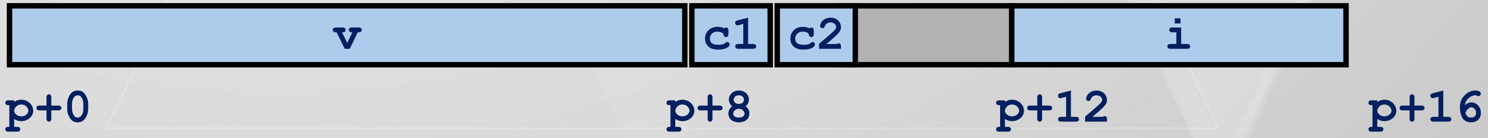
```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

在x86-64系统下，共有几个字节被浪费?




```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} *p;
```

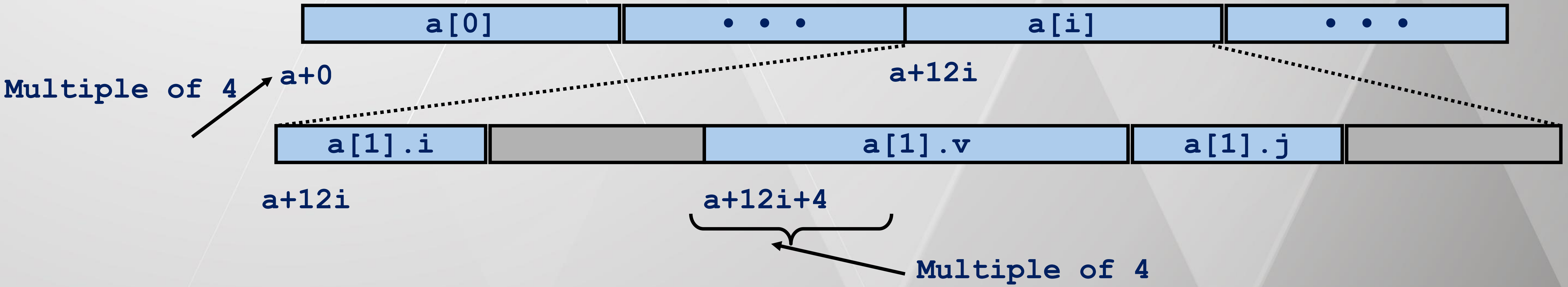
变换顺序后呢?



结构的存储对齐要求小结

- 结构起始地址的对齐要求等同于该结构各个元素中对齐要求最高的那个
  - a中每个元素的地址是4的整数倍
- 结构中元素的对齐要求必须满足该元素自身的对齐要求
  - v必须是4对齐
- 结构的长度必须是该结构各个元素中对齐要求最高的那个元素长度的整数倍

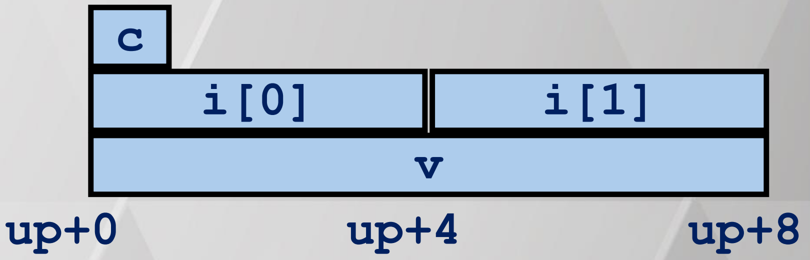
```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```



联合

- union中可以定义多个成员，union的大小由最大的成员的大小决定
- union成员共享同一块大小的内存，一次只能使用其中的一个成员

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




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

小结

- C语言数组的汇编访问
  - 连续存储
  - 访问代码优化
  - 无边界检查
- 结构
  - 对齐要求
  - 以及相应的汇编代码
- 联合

C函数返回struct类型是如何实现的?

```c
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;

int i = 2;

TEST_Struct __cdecl return_struct(int n)
{
    TEST_Struct local_struct;
    local_struct.age = n;
    local_struct.bye = n;
    local_struct.coo = 2*n;
    local_struct.ddd = n;
    local_struct.eee = n;

    i = local_struct.eee + local_struct.age *2 ;

    return local_struct;
}

int function1()
{
    TEST_Struct main_struct = return_struct(i);

    return 0;
}
```

```
return_struct:
        movq    %rdi, %rax
        movl    %esi, (%rdi)
        movl    %esi, 4(%rdi)
        leal    (%rsi,%rsi), %edx
        movl    %edx, 8(%rdi)
        movl    %esi, 12(%rdi)
        movl    %esi, 16(%rdi)
        addl    %edx, %esi
        movl    %esi, i(%rip)
        ret

function1:
        subq    $32, %rsp
        movl    i(%rip), %esi
        movq    %rsp, %rdi
        call    return_struct
        movl    $0, %eax
        addq    $32, %rsp
        ret
```

C函数是如何传入struct类型参数的?

```c
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;


int i = 2;
int input_struct(TEST_Struct in_struct)
{
    return in_struct.eee + in_struct.age*2 ;
}
int function2()
{
    TEST_Struct main_struct;
    main_struct.age = i;
    main_struct.bye = i;
    main_struct.coo = 2*i;
    main_struct.ddd = i;
    main_struct.eee = i;
    return input_struct(main_struct);
}
```

-Og →

```
input_struct:
        movl    8(%rsp), %eax    #age
        addl    %eax, %eax
        addl    24(%rsp), %eax   #eee
        ret

function2:
        subq    $56, %rsp
        movl    i(%rip), %eax
        movl    %eax, 24(%rsp)  #age
        movl    %eax, 28(%rsp)  #bye
        leal    (%rax,%rax), %edx
        movl    %edx, 32(%rsp)  #coo
        movl    %eax, 36(%rsp)  #ddd
        movq    24(%rsp), %rdx
        movq    %rdx, (%rsp)       #age/bye
        movq    32(%rsp), %rdx
        movq    %rdx, 8(%rsp)   #coo/ddd
        movl    %eax, 16(%rsp)  #eee
        call    input_struct
        addq    $56, %rsp
        ret
```

```c
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;


int i = 2;
int input_struct(TEST_Struct in_struct)
{
    return in_struct.eee + in_struct.age*2 ;
}
int function2()
{
    TEST_Struct main_struct;
    main_struct.age = i;
    main_struct.bye = i;
    main_struct.coo = 2*i;
    main_struct.ddd = i;
    main_struct.eee = i;
    return input_struct(main_struct);
}
```

-O1/2

```
input_struct:
        movl    24(%rsp), %eax
        movl    8(%rsp), %edx
        leal    (%rax,%rdx,2), %eax
        ret
function2:
        movl    i(%rip), %eax
        leal    (%rax,%rax,2), %eax
        ret
```

```c
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;


int i = 2;
static int input_struct(TEST_Struct in_struct)
{
    return in_struct.eee + in_struct.age*2 ;
}
int function2()
{
    TEST_Struct main_struct;
    main_struct.age = i;
    main_struct.bye = i;
    main_struct.coo = 2*i;
    main_struct.ddd = i;
    main_struct.eee = i;
    return input_struct(main_struct);
}
```

-O1/2

```
function2:
        movl    i(%rip), %eax
        leal    (%rax,%rax,2), %eax
        ret
```

```c
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;


int i = 2;
int input_struct(TEST_Struct in_struct)
{
    return in_struct.eee + in_struct.age*2 ;
}
int function2()
{
    TEST_Struct main_struct;
    main_struct.age = i;
    main_struct.bye = i;
    main_struct.coo = 2*i;
    main_struct.ddd = i;
    main_struct.eee = i;
    return input_struct(main_struct);
}
```

-O0

```
input_struct:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    32(%rbp), %eax
        movl    16(%rbp), %edx
        addl    %edx, %edx
        addl    %edx, %eax
        popq    %rbp
        ret
function2:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $32, %rsp
        movl    i(%rip), %eax
        movl    %eax, -32(%rbp)
        movl    i(%rip), %eax
        movl    %eax, -28(%rbp)
        movl    i(%rip), %eax
        addl    %eax, %eax
        movl    %eax, -24(%rbp)
        movl    i(%rip), %eax
        movl    %eax, -20(%rbp)
        movl    i(%rip), %eax
        movl    %eax, -16(%rbp)
        subq    $24, %rsp
        movq    %rsp, %rax
        movq    -32(%rbp), %rdx
        movq    %rdx, (%rax)
        movq    -24(%rbp), %rdx
        movq    %rdx, 8(%rax)
        movl    -16(%rbp), %edx
        movl    %edx, 16(%rax)
        call    input_struct
        addq    $24, %rsp
        leave
        ret
```

movq %rbp, %rsp
popq %rbp

```c
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;


int input_struct(TEST_Struct in_struct)
{
    return in_struct.eee + in_struct.age*2 ;
}
```

$\xrightarrow[\text{-Og/1/2/0}]{}$?

C99的VLA（variable-length array ）

```c
extern long read_val();

long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

```
read_and_process:
        pushq   %rbp                    } Set up
        movq    %rsp, %rbp

        pushq   %r14                    } Callee saved
        pushq   %r13
        pushq   %r12
        pushq   %rbx

        movl    %edi, %r13d             # %r13d = n
        movslq  %edi, %rax              # %rax = n
        leaq    22(,%rax,8), %rax       # 8*n + 22 //long 类型
        andq    $-16, %rax              # 16字节对齐
        subq    %rax, %rsp              # 栈上分配空间
        movq    %rsp, %r14              # VLA地址？

        testl   %edi, %edi
        jle     .L2                     # n<=0?
```

```c
extern long read_val();

long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

```
    movq    %rsp, %rbx              # VLA地址？
    leal    -1(%rdi), %eax          # n-1
    leaq    8(%rsp,%rax,8), %r12    # (n-1)*8+VLA地址+8 = r12， 结束地址

.L3:
    movl    $0, %eax
    call    read_val
    movq    %rax, (%rbx)
    addq    $8, %rbx
    cmpq    %r12, %rbx
    jne     .L3
```

Loop body

```
.L2:
    subl    $1, %r13d
    movslq  %r13d, %r13
    movq    (%r14,%r13,8), %rax
```

vals[n-1]

```
    leaq    -32(%rbp), %rsp
    ....
    ret
```

Finish