程序的机器级表示：数据

Machine-Level Programming : Data

本章内容

Topic

- 数组

  Arrays

  - 一维

    One dimensional

  - 多维（嵌套）

    Multi-dimensional(nested)

  - 多层

    Multi-level

- 结构体

  Structures

- 联合体

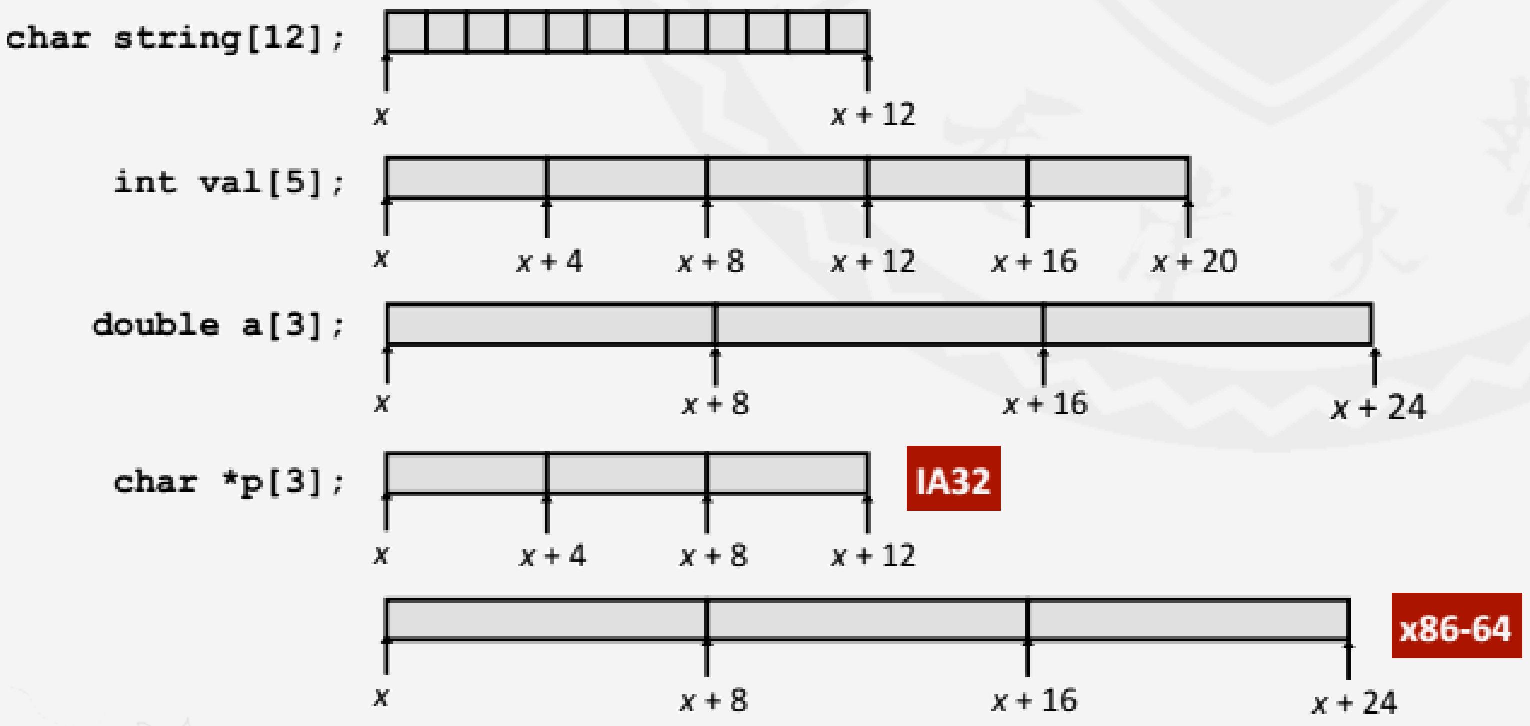
  Union

数组

Arrays

数组空间分配
Array Allocation

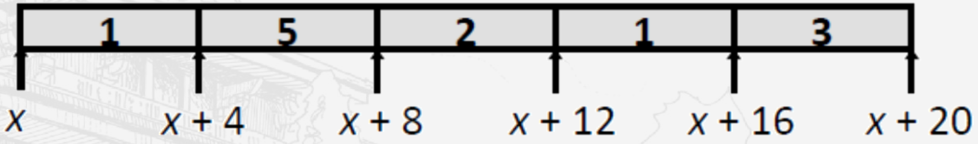
- 基本语法规则 Basic Principle　　*T*　A[*L*];
  - 元素数据类型为*T*，长度为*L*
    Array of data type T and length L
  - 内存中连续分配 *L**sizeof(*T*) 个字节
    Contiguously allocated region of *L**sizeof(*T*) bytes in memory



数组的访问
Array Allocation

- 基本语法规则 Basic Principle　　*T*　　A[*L*];
  - 元素数据类型为*T*，长度为*L*
    Array of data type T and length L
  - A可以被看做是第0个元素的指针，类型为*T* *
    Identifier A can be used as a pointer to array element 0: Type *T* *

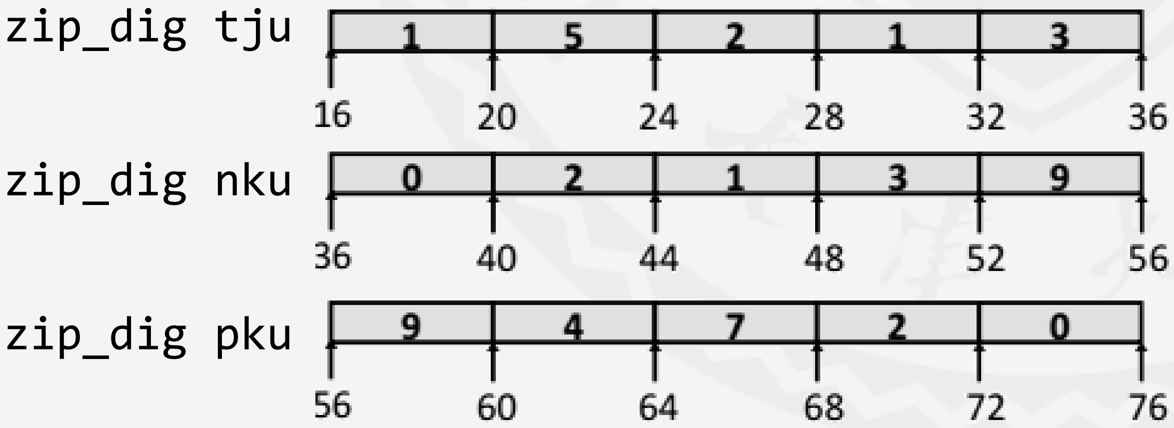
`int val[5];`



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 + 4i

举例：数组
Array Example

```
#define  ZLEN 5
typedef  int zip_dig[ZLEN];
zip_dig  tju = { 1, 5, 2, 1, 3 };
zip_dig  nku = { 0, 2, 1, 3, 9 };
zip_dig  pku = { 9, 4, 7, 2, 0 };
```



- "zip_dig tju;" 等价于 "int tju[5]"
  Declaration "zip_dig tju" equivalent to "int tju[5]"
- 示例中的每一个数组都被分配了20个字节的连续内存区域
  Example arrays were allocated in successive 20 byte blocks

数组

Arrays

举例：数组访问
Array Accessing Example

zip_dig tju


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

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

- %rdi 存储数组的起始地址
  Register %rdi contains starting address of array
- %rsi 存储数组的下标（索引）
  Register %rsi contains array index
- 目标数据地址为 %rdi + 4*%rsi
  Desired digit at %rdi + 4*%rsi
- 使用存储器寻址表示为(%rdi, %rsi, 4)
  Use memory reference (%rdi, %rsi, 4)

举例：数组循环遍历
Array Loop Example

```
void zincr(zip_dig z) {
   int 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
```

本章内容

Topic

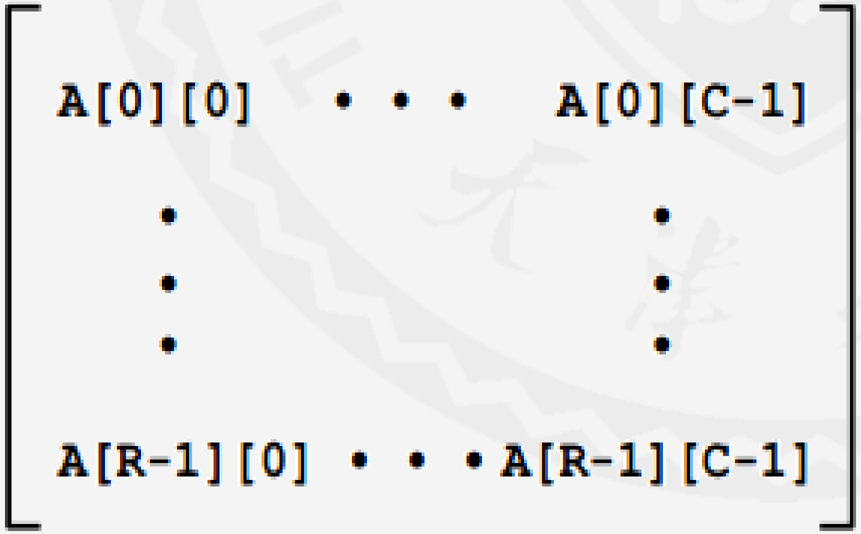
- 数组
  Arrays
  - 一维
    One dimensional
  - 多维（嵌套）
    Multi-dimensional(nested)
  - 多层
    Multi-level
- 结构体
  Structures
- 联合体
  Union

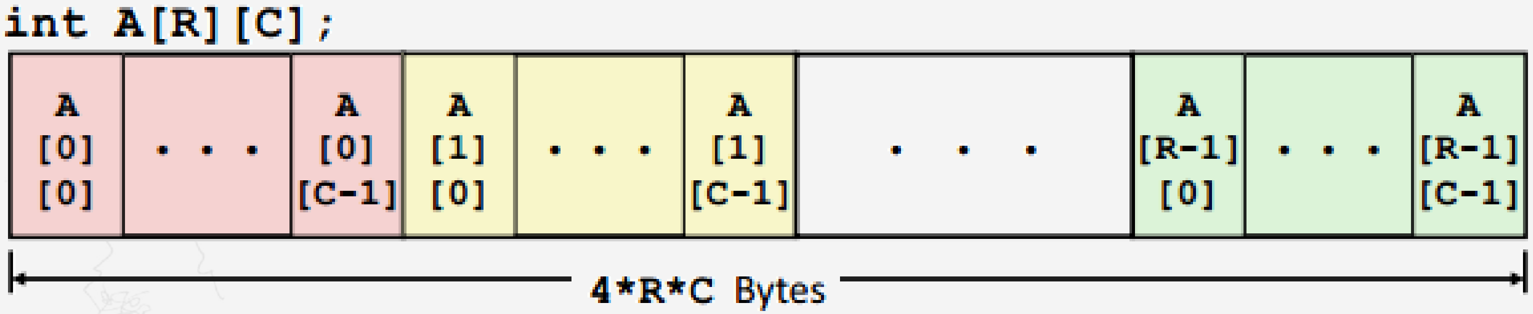
数组

Arrays

多维（嵌套）数组
Multi-dimensional (Nested) Arrays

- 基本语法规则 Basic Principle　　T A[R][C];
  - 元素类型为 T 的二维数组
    2D array of data type T
  - R 行，C 列
    R rows C columns
  - T类型的元素需要 K 个字节
    Type T element requires K bytes
- 数组大小：R * C * K bytes
  Array size：R * C * K bytes
- 排列方式：行优先
  Arrangement：Row-Major Ordering

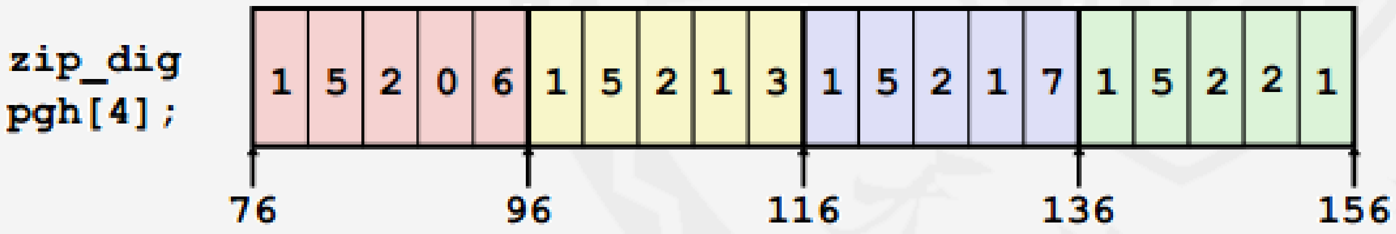




数组

Arrays

举例：嵌套数组
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 }};
```

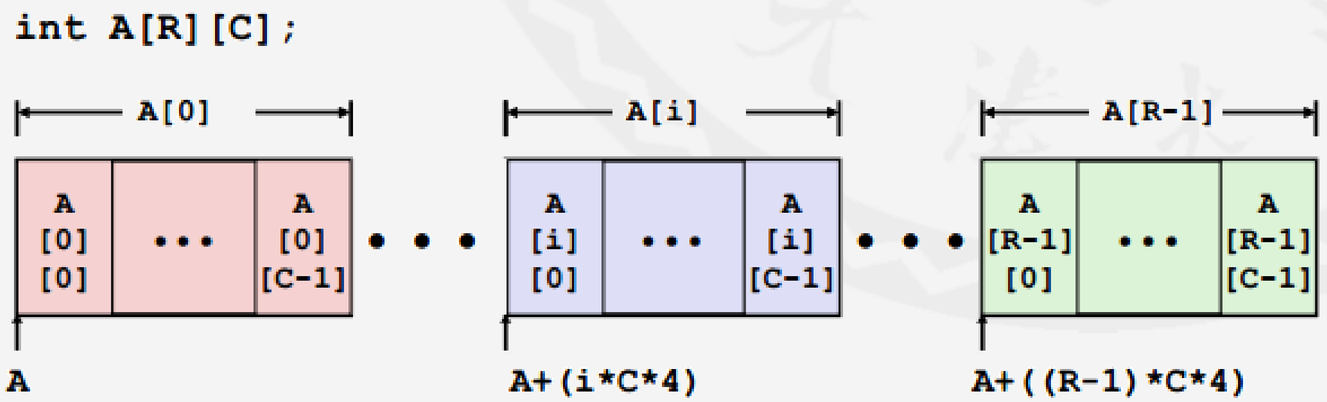
- “zip_dig pgh[4]” 等价于 “int pgh[4][5]”
  “zip_dig pgh[4]” equivalent to “int pgh[4][5]”
  - 变量 pgh：包含4个元素的数组，连续分配
    Variable pgh: array of 4 elements allocated contiguously
  - 其中每个元素是一个包含5个int类型数据的数组
    Each element is an array of 5 int’s allocated contiguously
- 所有的元素都是按照“行优先” 排列
  “Row-Major” ordering of all elements guaranteed

数组

Arrays

访问嵌套数组的行
Nested Array Row Access

- 行向量
  Row Vectors
  - A[i]是一个包含C个元素的数组
    A[i] is array of C elements
  - 元素类型为T，需要K个字节
    Each element of type T requires K bytes
  - 起始地址为 A + i*(C*K)
    Starting address A + i*(C*K)



数组

Arrays

代码：访问嵌套数组的行
Nested Array Row Access Code

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

```
#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 }};
```

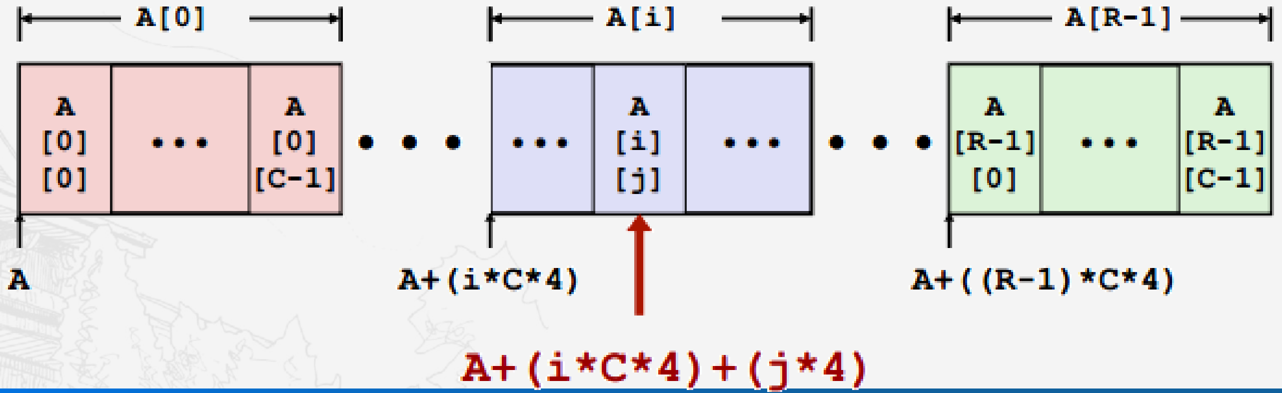
```
# %rdi = index
leaq (%rdi, %rdi, 4), %rax   # 5 * index
leaq pgh(, %rax, 4), %rax    # pgh + (20*index)
```

- 行向量
  Row Vectors
  - `pgh[index]`是一个包含5个int型数据的数组
    `pgh[index]` is array of 5 int's
  - 起始地址：`pgh + (20*index)`
    Starting address: `pgh + (20*index)`
- 汇编指令
  Machine Code
  - 计算并返回行向量的地址
    Computes and returns address
  - 进行如下计算：`pgh + 4*(index + 4*index)`
    Compute as `pgh + 4*(index + 4*index)`

嵌套数组元素的访问
Nested Array Element Access

- 数组元素
  Array Elements
  - A[i][j]数据类型为 T 的数组元素，需要 K 个字节
    A[i][j] is element of type T which requires K bytes
  - 地址：A + i*(C*K) + j*K = A + (i*C + j)*K
    Address: A + i*(C*K) + j*K = A + (i*C + j)*K

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



代码：嵌套数组元素的访问
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
  - pgh[index][dig] 是 int 类型
    pgh[index][dig] is int
  - 地址： **pgh + 20*index + 4*dig = pgh + 4*(5*index + dig)**
    Address: **pgh + 20*index + 4*dig = pgh + 4*(5*index + dig)**
- 汇编指令
  Machine Code
  - 地址计算方法如下： **pgh + 4*((index+4*index)+dig)**
    Compute address as pgh + 4*((index+4*index)+dig)

本章内容

Topic

- 数组

  Arrays

  - 一维

    One dimensional

  - 多维（嵌套）

    Multi-dimensional(nested)

  - 多层

    Multi-level

- 结构体

  Structures

- 联合体

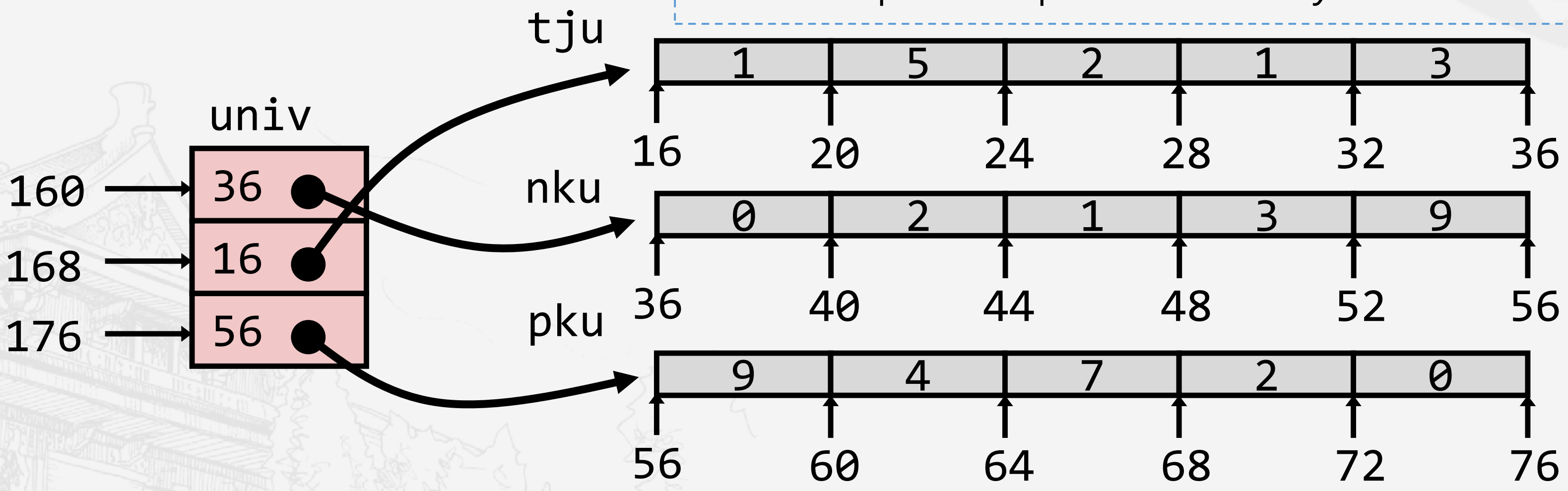
  Union

举例：多层数组
Multi-Level Array Example

```
zip_dig  tju = { 1, 5, 2, 1, 3 };
zip_dig  nku = { 0, 2, 1, 3, 9 };
zip_dig  pku = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {nku, tju, pku};
```

- 变量 univ 表示一个包含3个元素的数组
  Variable univ denotes array of 3 elements
- 每个元素是一个指针：8字节
  Each element is a pointer: 8 bytes
- 每个指针指向一个int型数组
  Each pointer points to array of int’s



数组

Arrays

代码：多层数组的元素访问
Element Access Code in Multi-Level Array

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

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

- 元素访问:
  Element access: **Mem[Mem[univ+8*index]+4*digit]**
- 需要进行两次存储器访问
  Must do two memory reads
  - 首先获得行数组的地址
    First get pointer to row array
  - 然后访问（行）数组中的元素
    Then access element within array

数组

Arrays

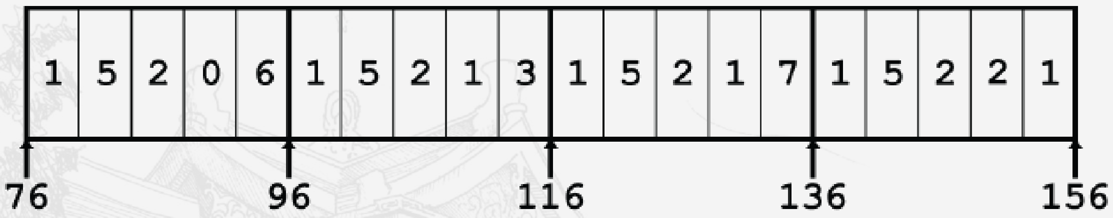
比较
Comparison

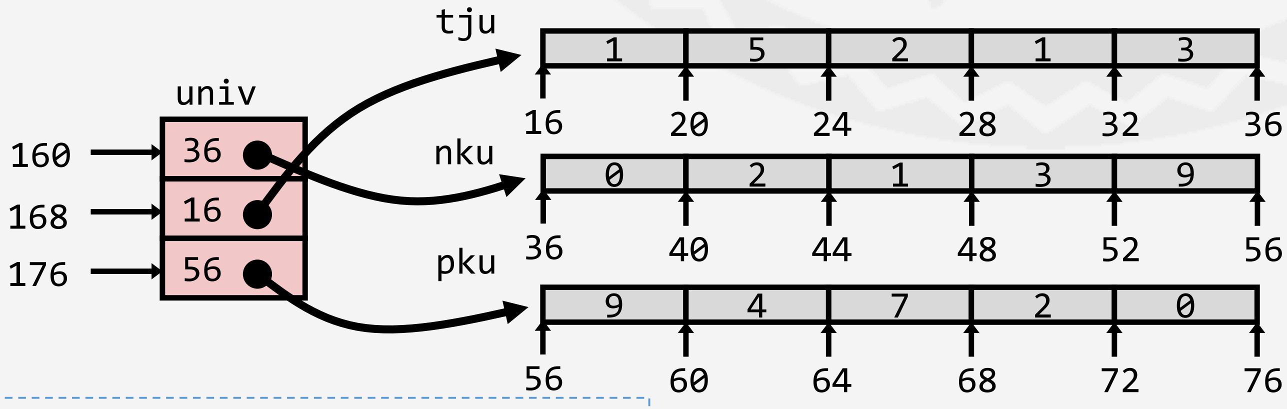
Nested array

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

Multi-level array

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





- **从C语法上看相似，但是寻址方式不同**

  Accesses looks similar in C but addresses very different

数组

Arrays

代码：n×n 矩阵
n x n Matrix Code

- 固定维度
  Fixed dimensions
  - 在编译时N的大小已经确定
    Know value of N at compile time
- 可变维度，显示索引
  Variable dimensions, explicit indexing
  - 传统的实现动态数组的方法
    Traditional way to implement dynamic arrays
- 可变维度，隐式索引
  Variable dimensions implicit indexing
  - 该语法已被最新的gcc所支持
    Now supported by gcc

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

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

```
int var_ele (size_t n, int a[n][n],
             size_t i, size_t j)
{
        return a[i][j];
}
```

数组

Arrays

16×16 矩阵
16×16 Matrix Access

- 数组元素

  Array Elements

  - 地址 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                # i*64
addq %rsi, %rdi              # a + 64*i
movl (%rdi, %rdx, 4), %eax   # M[a + 64*I + 4*j]
retq
```

数组

Arrays

n×n 矩阵
n×n Matrix Access

- 数组元素

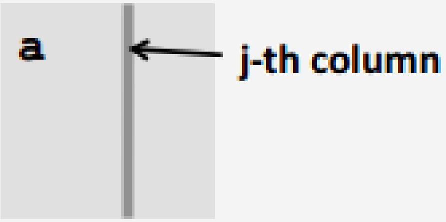
  Array Elements

  - 地址 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
retq
```

优化定长矩阵的访问
Optimizing Fixed Matrix Access



- 依次遍历第j列的所有元素
  Step through all elements in column j
- 优化
  Optimization
  - 从某一列查找连续元素
    Retrieving successive elements from single column
  - 计算
    Compute   `ajp = &a[i][j]`
    - 初始
      Initially   `ajp = a + 4*j`
    - 每次增加 4*N
      Increment by 4∗N   `ajp += 4*N`

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

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

本章内容

Topic

- 数组

  Arrays

- 结构体

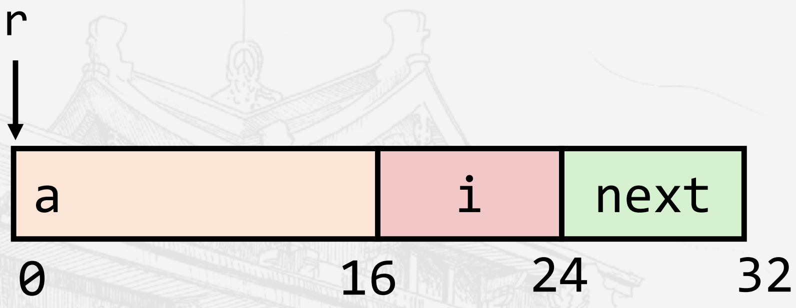
  Structures

- 联合体

  Union

结构体的表示
Structure Representation

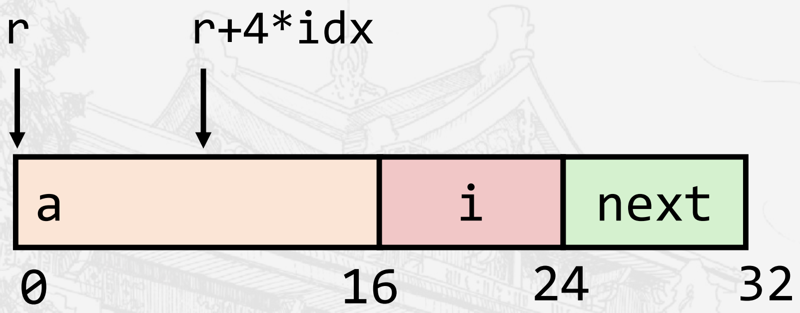
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- 结构体被看做是一块连续的内存区域
  Structure represented as block of memory
  - 足够大以容纳下结构体中所有的成员
    Big enough to hold all of the fields
- 成员在内存中组织的顺序和声明的顺序一致
  Fields ordered according to declaration
  - 即使另一种排序可以产生更紧凑的表示（也不会发生改变）
    Even if another ordering could yield a more compact representation
- 编译器决定了结构体的大小和每个成员在内存中的位置
  Compiler determines overall size + positions of fields
  - 在机器级程序不了解高级语言源代码中的结构体
    Machine-level program has no understanding of the structures in the source code

获得结构体成员的指针
Generating Pointer to Structure Member

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



- 计算数组成员的指针 **`r->a[idx]`**
  Generating Pointer to Array Element
  - 每个结构体成员的偏移量在编译期决定
    Offset of each structure member determined at compile time
  - 计算结果 **`r + 4*idx`**
    Compute as

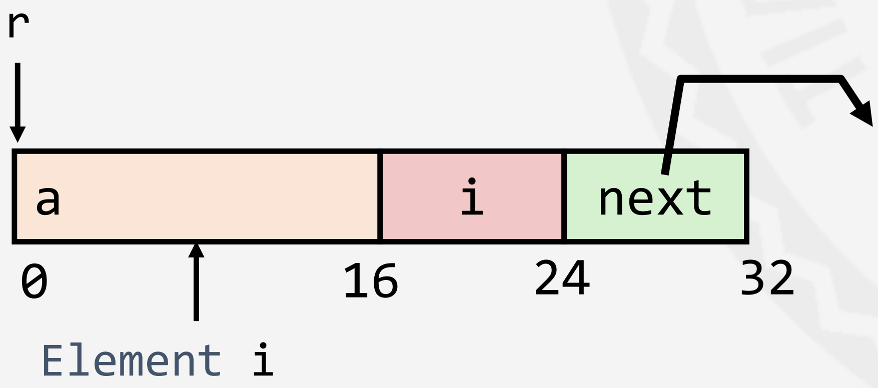
```
int *get_ap (struct rec *r, size_t idx) {
  return &r->a[idx];
}
```

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

链表遍历
Following Linked List

- C Code

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



Register	Use(s)
%rdi	r
%rsi	val

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

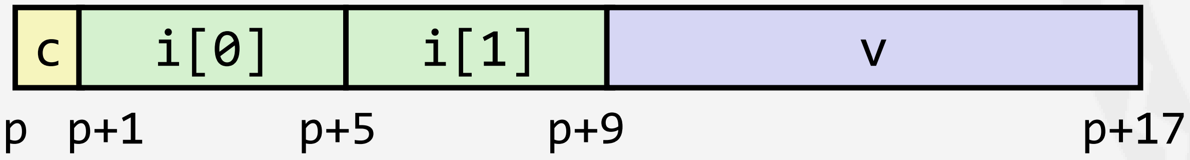
```
.L11:                                  # loop:
  movslq  16(%rdi), %rax               #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)          #   M[r+4*i] = val
  movq    24(%rdi), %rdi               #   r = M[r+24]
  testq   %rdi, %rdi                   #   Test r
  jne     .L11                 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

结构体

Structures

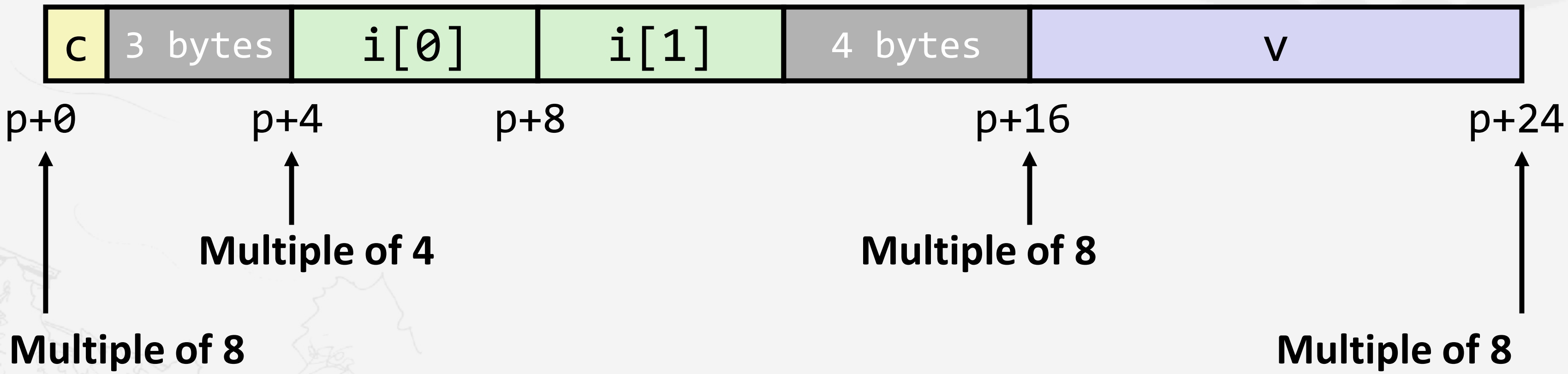
结构体对齐
Structures Alignment

- 数据未对齐
  Unaligned Data




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

- 数据对齐
  Unaligned Data
  - 若基本数据类型需要K个字节
    Primitive data type requires K bytes
  - 则地址必须是K的整数倍
    Address must be multiple of K

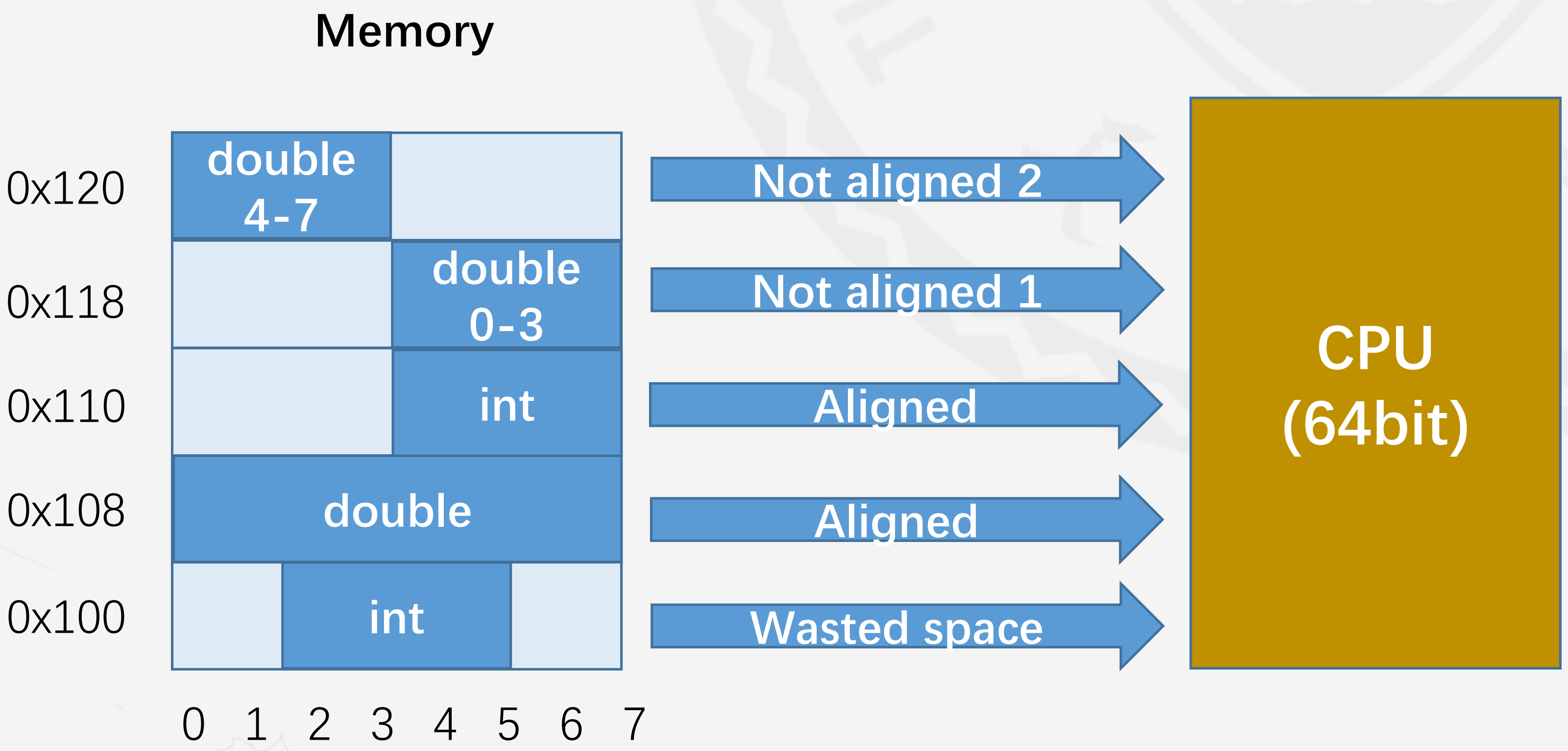


结构体

Structures

为什么需要结构体对齐？
Why Alignment Is Needed?

- 物理上，内存是以连续4或8字节块的方式进行访问（依赖于系统）
  Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - 如果数据跨过四字的边界，数据的访问效率低
    Inefficient to load or store datum that spans quad word boundaries
  - 当数据横跨2个内存页，虚拟内存在处理上十分复杂
    Virtual memory trickier when datum spans 2 pages



对齐规则
Alignment Principles

- 数据对齐
  Unaligned Data
  - 若基本数据类型需要K个字节
    Primitive data type requires K bytes
  - 则地址必须是K的整数倍
    Address must be multiple of K
  - 在某些处理器上是必须的使用的，x86-64处理器是推荐使用的（不必须）
    Required on some machines; advised on x86-64
- 编译器会在结构体中插入间隙以保证各成员正确对齐
  Compiler Inserts gaps in structure to ensure correct alignment of fields

具体的对齐案例（x86-64）
Specific Cases of Alignment (x86-64)

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

具体的对齐案例（IA32）
Specific Cases of Alignment (IA32)

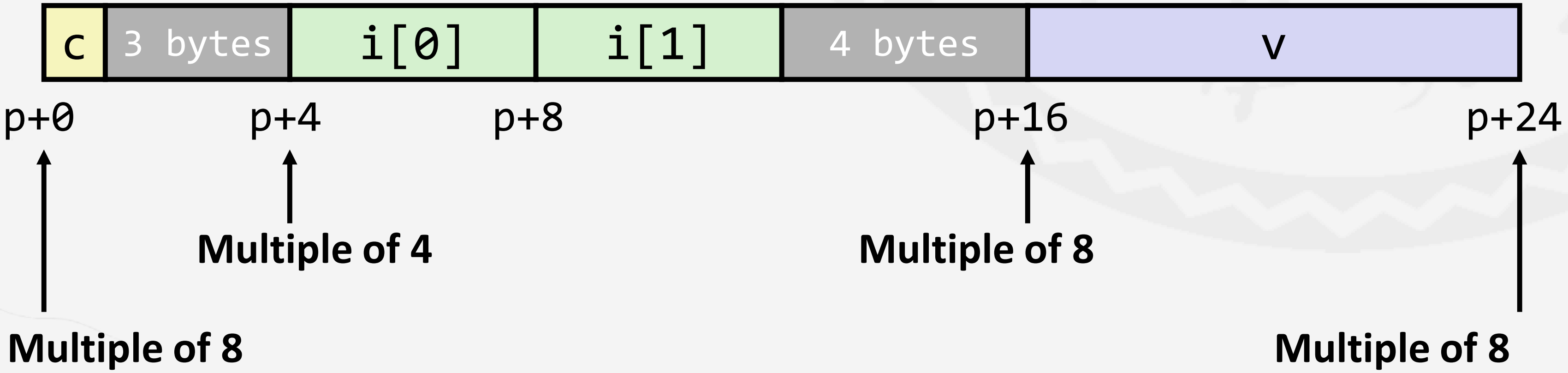
- 1 byte: char, …
  - 地址没有限制
    no restrictions on address
- 2 bytes: short, …
  - 地址的最低位必须为0_2
    lowest 1 bit of address must be 0_2
- 4 bytes: int, float, long, char *, …
  - 地址的最低2位必须为00_2
    lowest 2 bits of address must be 00_2
- 8 bytes: double …
  - Windows 和其他的大多数操作系统的指令集
    - 地址的最低3位必须为000_2
      lowest 3 bits of address must be 000_2
  - Linux
    - 地址的最低2位必须为00_2
      lowest 3 bits of address must be 000_2
    - 视为一个4字节的基本数据类型
      treated the same as a 4-byte primitive data type

满足结构体的对齐
Satisfying Alignment with Structures

- 在结构体内
  Within structure:
  - 每个成员都需要对齐
    Must satisfy each element's alignment requirement
- 整个结构体的放置（结构体起始地址要求）
  Overall structure placement
  - 每个结构体有一个对齐要求K
    Each structure has alignment requirement K
  - K 为结构体中所有元素中的最大对齐需求
    K = Largest alignment of any element
  - 结构体的初始地址和大小必须为K的整数倍
    Initial address & structure length must be multiples of K

举例：满足结构体的对齐 #1
Satisfying Alignment with Structures Example #1

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



结构体的对齐需求

Structure Alignment

K = 8

举例：满足结构体的对齐 #2
Satisfying Alignment with Structures Example #2

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

v	i[0]	i[1]	c	7 bytes

p+0 p+8 p+16 p+24

Multiple of K=8

最大的对齐需求为 8

For largest alignment requirement 8

结构体的整体大小必须为8的整数倍

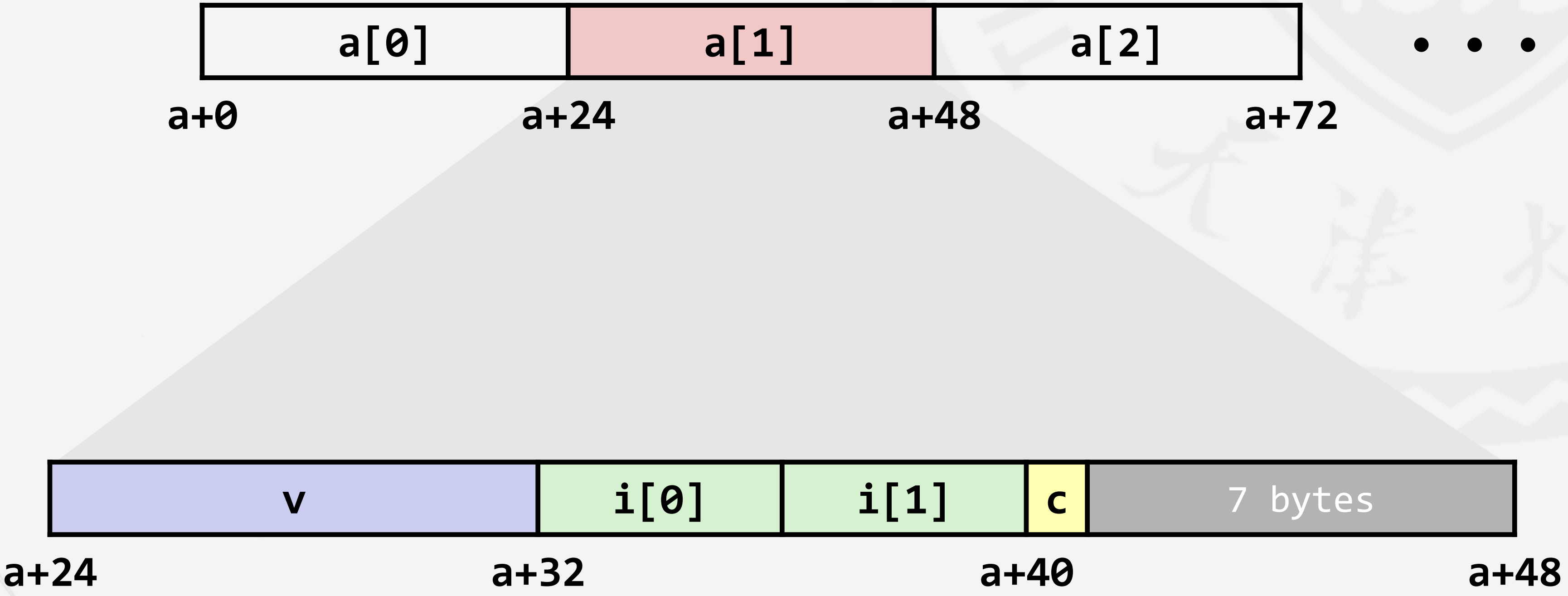
The size of overall structure must be multiple of 8

结构体

Structures

举例：结构体数组
Arrays of Structures

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



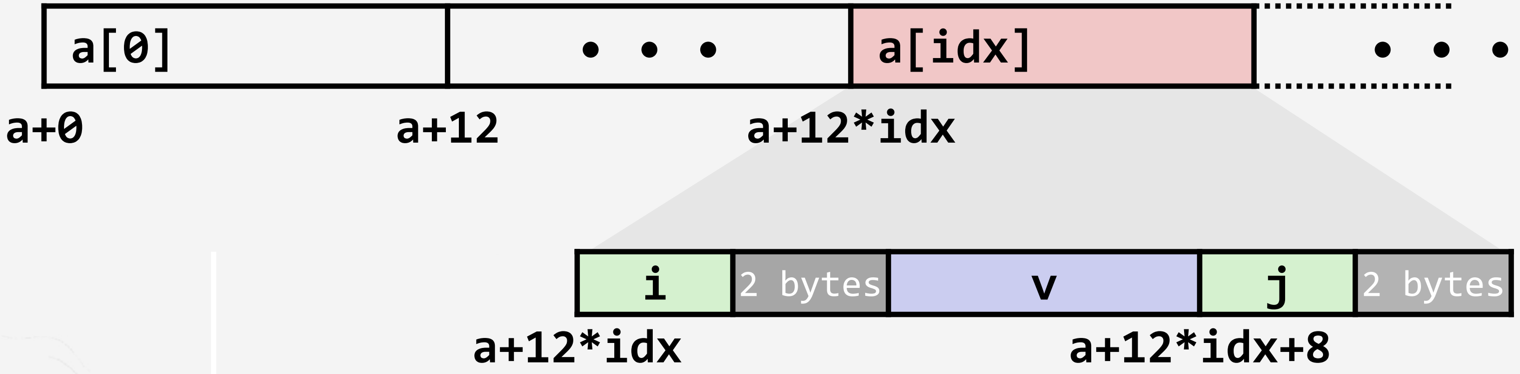
结构体的整体大小必须为8的整数倍

The size of overall structure must be multiple of 8

结构体

Structures

访问结构体数组中的元素
Accessing Structures Array Elements



- 计算结构体元素的偏移量
  Compute array offset `12*idx`
  - `sizeof(struct S3) = 12`
- 成员 j 在结构体中的偏移量为8
  Field j is at offset 8 within structure

```
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
```

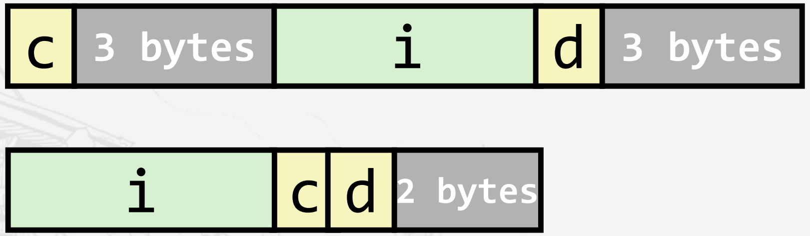
节约空间
Saving Space

把尺寸大的成员放在结构体的前面
Put large data types first

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

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

K=4



本章内容

Topic

- 数组

  Arrays
- 结构体

  Structures
- 联合体

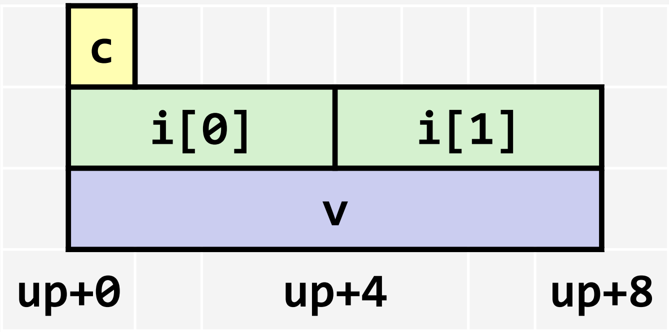
  Union

联合体

Union

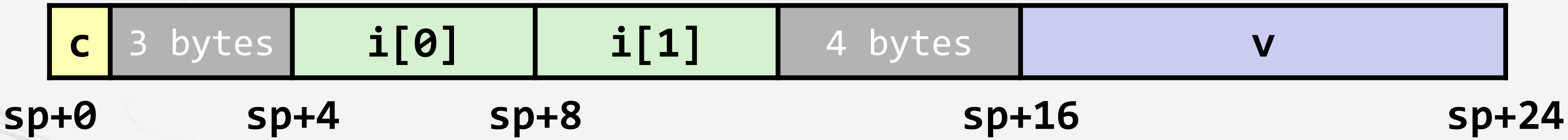
联合体的空间分配
Union Allocation

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



- 基于最大的成员分配空间
  Allocate according to largest element
- 一次只能使用其中的一个成员
  Can only use one field at a time

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



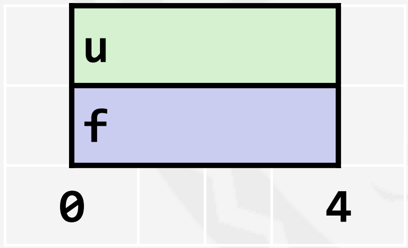
联合体

Union

使用联合体获得数据的编码

Union Allocation

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




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

与 (float) u 相同吗?

Same as (float) u **?**

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

与(unsigned) f 相同吗?

Same as (unsigned) f **?**

联合体

Union

重新审视字节序
Byte Ordering Revisited

- 思想
  Idea
  - short/long/quad words 在内存中以 2/4/8个连续字节存储
    Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - 哪一个字节是最高（低）字节
    Which byte is most (least) significant?
  - 这会导致在计算机间传输二进制数据的出现问题
    Can cause problems when exchanging binary data between machines

- 大端
  Big Endian
  - 最高字节在最低地址
    Most significant byte has lowest address
  - Sparc
- 小端
  Little Endian
  - 最低字节在最低地址
    Least significant byte has lowest address
  - Intel x86 and IOS
- 双端
  - 可以通过某种方式进行配置
    Can be configured either way
  - ARM

联合体

Union

举例：字节序
Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

32-bit

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

64-bit

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

联合体

Union

举例：字节序
Byte Ordering Example

```
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]);
```

联合体

Union

IA32字节序
Byte Ordering on IA32

Little Endian

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

LSB MSB LSB MSB

Print

Output:

```
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]
```

联合体

Union

Sparc字节序
Byte Ordering on Sparc

Big Endian

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

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

Output on Sun:

```
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]
```

联合体

Union

x86-64字节序
Byte Ordering on x86-64

Little Endian

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

Output on x86-64:

```
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]
```

总结
Summary

- 数组
  Arrays
  - 连续内存分配
    Contiguous allocation of memory
  - 每个元素都要满足对齐要求
    Aligned to satisfy every element’s alignment requirement
  - 数组的名称是指向第一个元素的指针
    Pointer to first element
  - 没有边界检查
    No bounds checking
- 结构体
  Structures
  - 按照声明的顺序连续分配字节
    Allocate bytes in order declared
  - 在中间和最后插入间隙以满足对齐要求
    Pad in middle and at end to satisfy alignment
- 联合体
  Unions
  - 空间重叠的声明
    Overlay declarations
  - 一种绕过类型检查的方法
    Way to circumvent type system