计算机系统基础

程序的机器级表示(5)

王晶

jwang@ruc.edu.cn，信息楼124

2024年11月

提纲

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

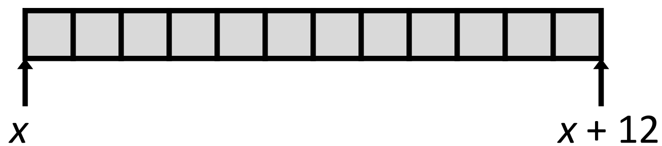
Array Allocation

- Basic Principle

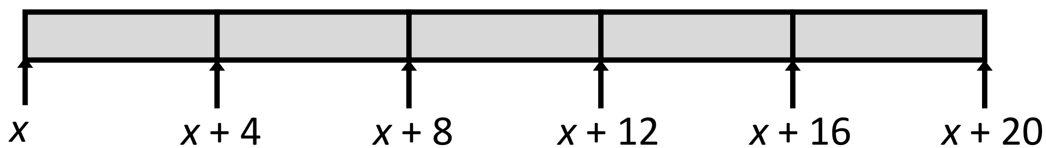
  T **A[*L*]**;

  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof**(*T*) bytes in memory

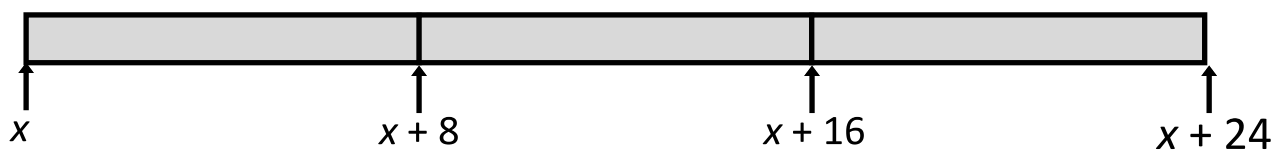
`char string[12];`



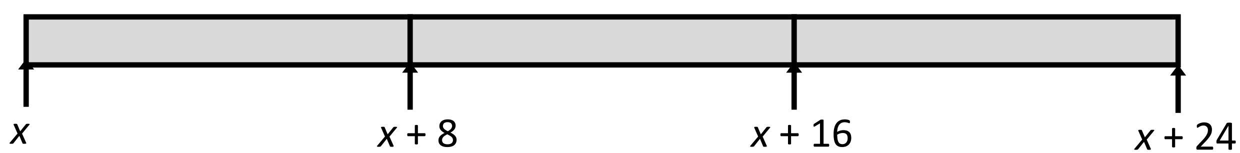
`int val[5];`



`double a[3];`



`char *p[3];`



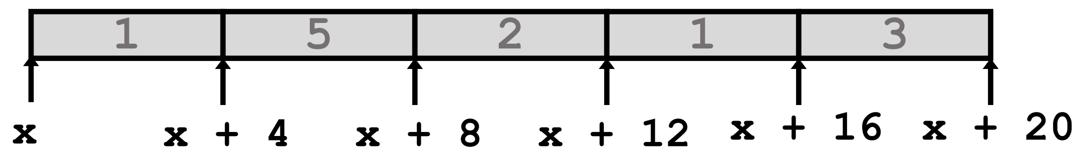
Array Access

- Basic Principle

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

  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**

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



- Reference Type Value

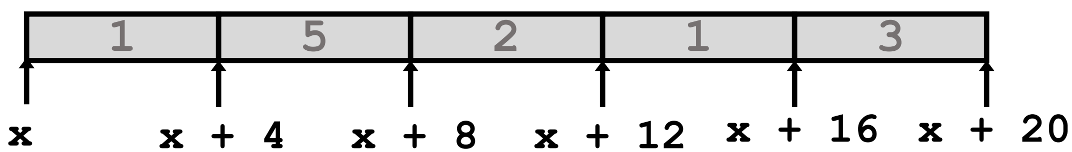
Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	x
`val+1`	`int *`	x+4
`&val[2]`	`int *`	x+8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	x+4*i

Array Access

- Basic Principle

  T **A[*L*];**
  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**

`int val[5];`



- Reference Type Value

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	
`val+1`	`int *`	
`&val[2]`	`int *`	
`val[5]`	`int`	
`*(val+1)`	`int`	
`val + i`	`int *`	

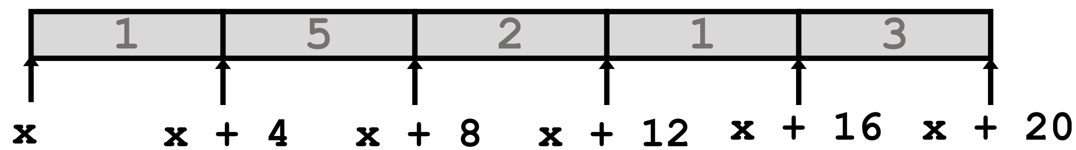
Array Access

- Basic Principle

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

  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**

`int val[5];`

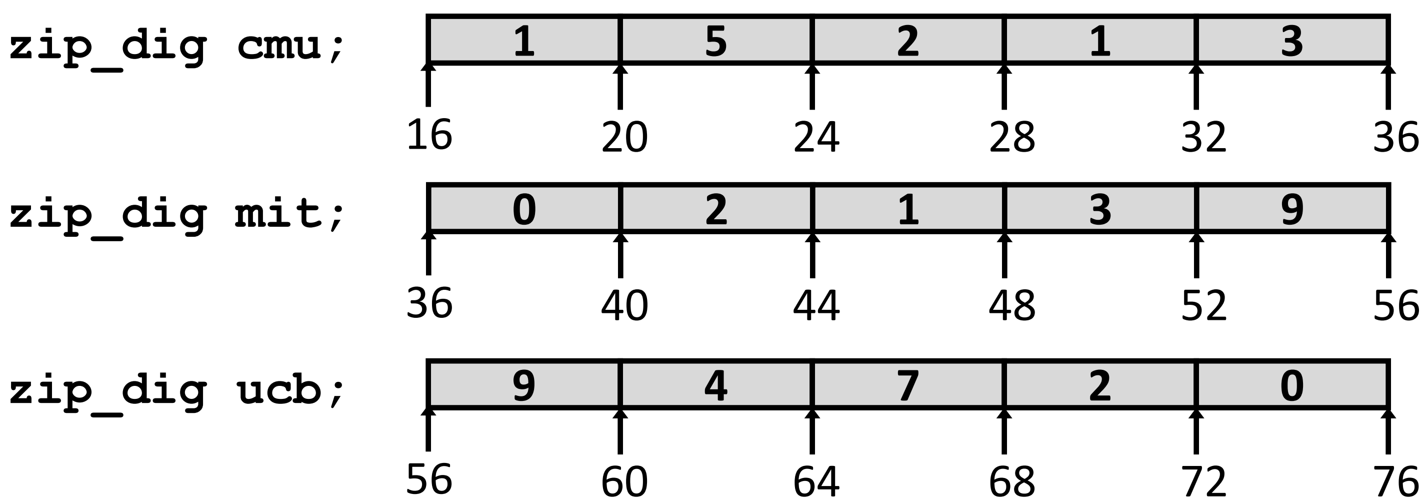


- Reference / Type / Value

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	
`val+1`	`int *`	
`&val[2]`	`int *`	
`val[5]`	`int`	
`*(val+1)`	`int`	
`val + i`	`int *`	

Array Example

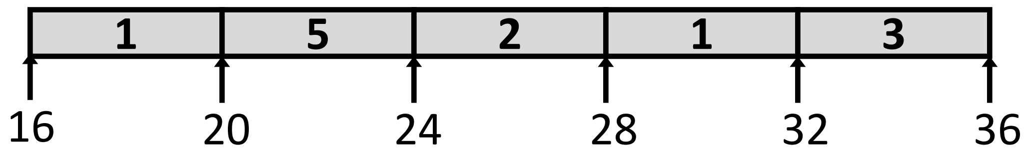
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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



- Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

```
zip_dig cmu;
```




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

IA32

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

%rdi数组起始
%rsi索引
4为数据元素大小

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

Array Loop Example

```
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
```

size_t i 无符号

Multidimensional (Nested) Arrays

- 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

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

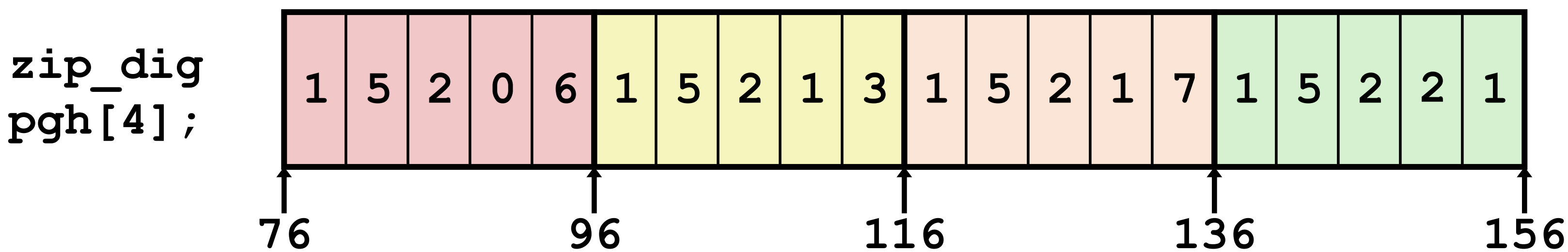
```
int A[R][C];
```

A [0] [0]	• • •	A [0] [C-1]	A [1] [0]	• • •	A [1] [C-1]	• • •	A [R-1] [0]	• • •	A [R-1] [C-1]

4*R*C Bytes

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 }};
```

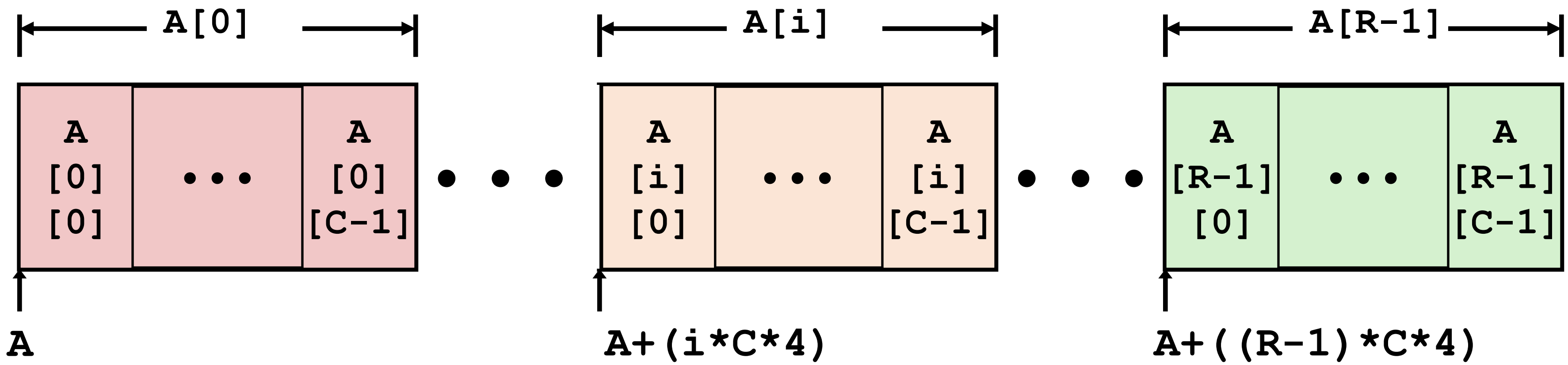


- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

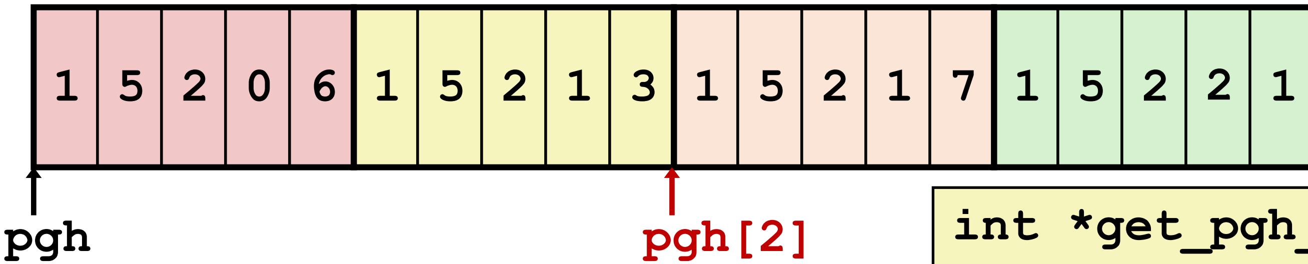
Nested Array Row Access

- 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];
```



Nested Array Row Access Code




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

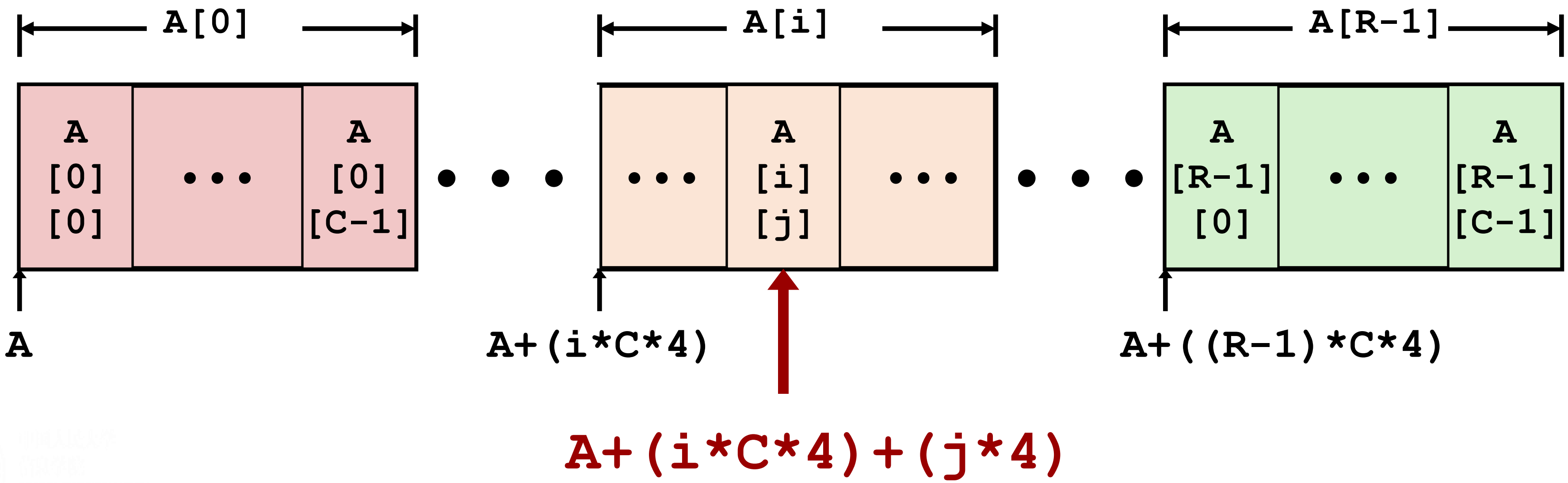
```
  # %rdi = index
  leaq (%rdi,%rdi,4),%rax # 5 * index
  leaq pgh(,%rax,4),%rax  # pgh + (20 * index)
```

- Row Vector
  - **`pgh[index]`** is array of 5 **`int`**'s
  - Starting address **`pgh+20*index`**
- Machine Code
  - Computes and returns address
  - Compute as **`pgh + 4*(index+4*index)`**

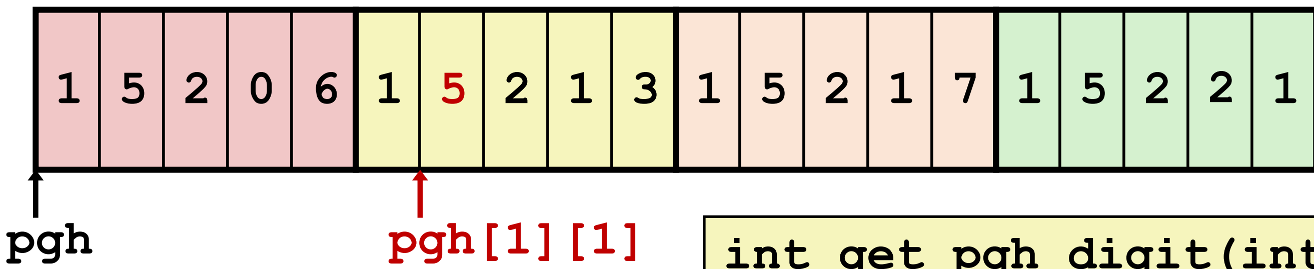
Nested Array Element Access

- 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];
```



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)]
```

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

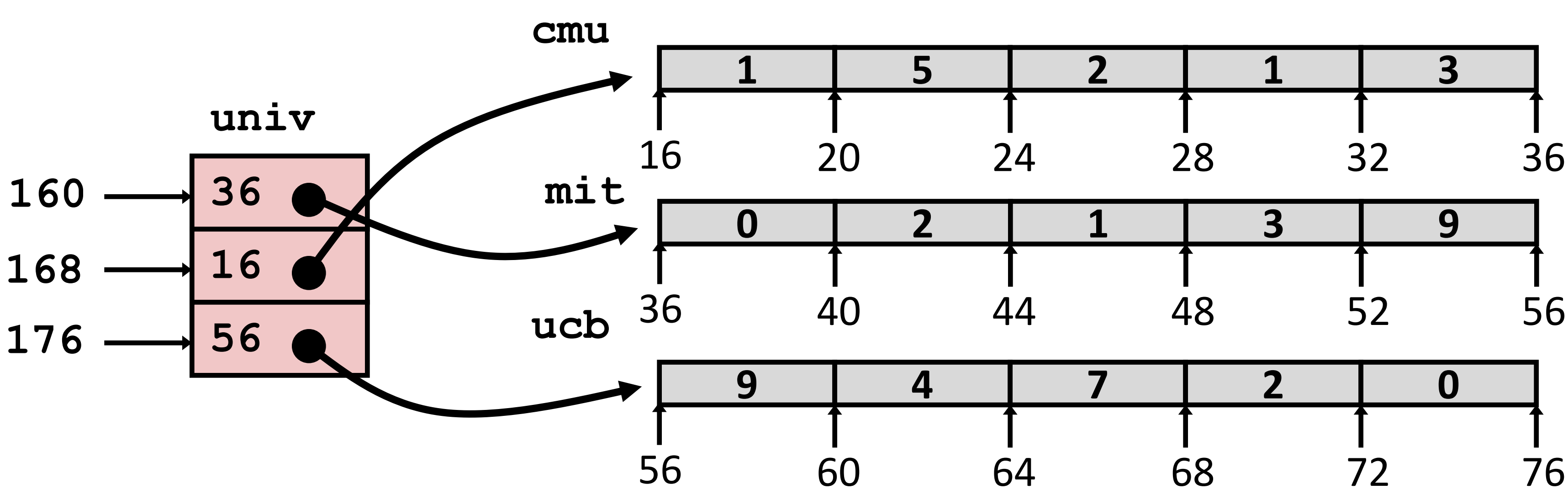
    = `pgh + 4*(5*index + dig)`

Multi-Level Array Example

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

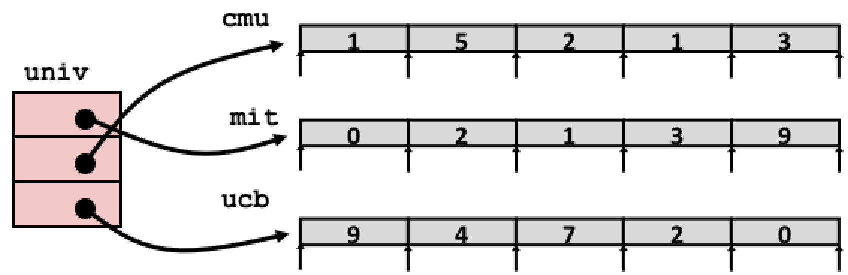
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of `int`'s



Element Access in Multi-Level Array

```
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
```

左移

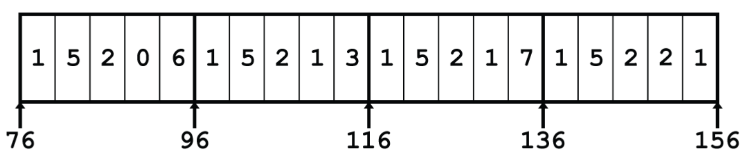
存储器访问慢——第一种方法好

- Computation
  - Element access **`Mem[Mem[univ+8*index]+4*digit]`**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

Array Element Accesses

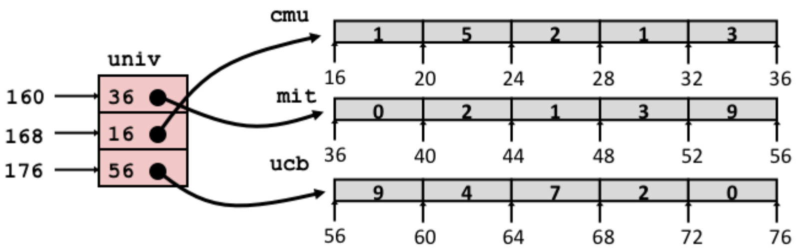
Nested array

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

- Variable dimensions, implicit indexing
  - Now supported by gcc

```
/* 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 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
  - Must perform integer multiplication

```
/* 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
```

Example: Array Access

```c
#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
```

填空题 1分

设置

```
#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;
}
```

强制转化为int* 指针

9

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

[填空1]

作答

Example: Array Access

```
#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;
}
```

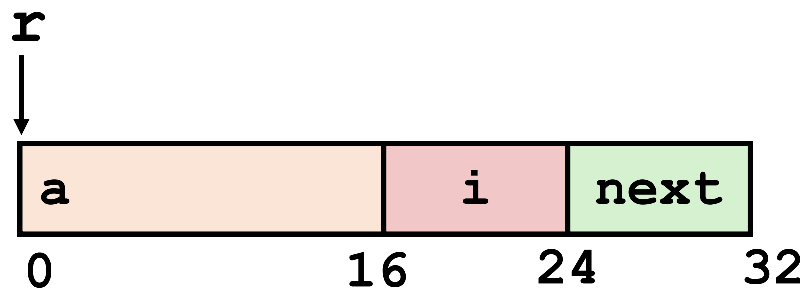
```
linux> ./array
result: 9
```

提纲

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

Structure Representation

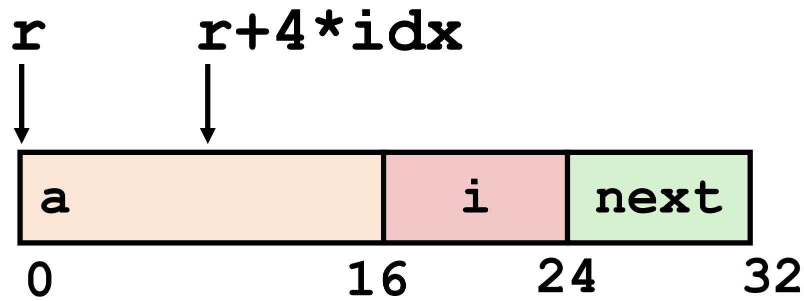
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

Generating Pointer to Structure Member

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as **`r + 4*idx`**

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

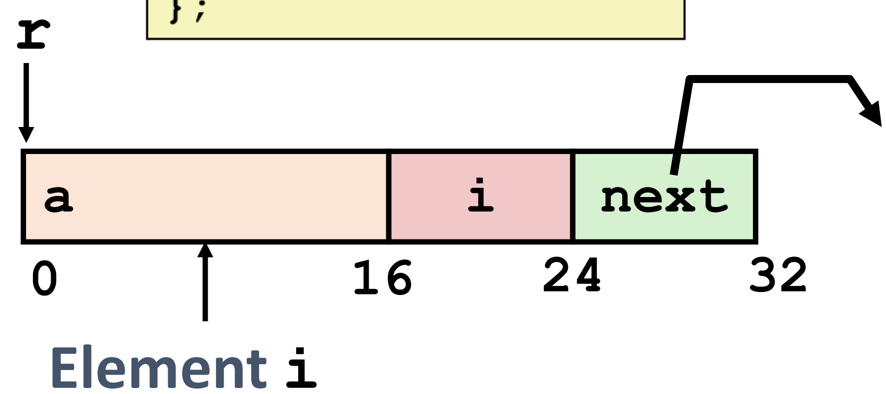
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Following Linked List

- C Code

```
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;
  }
}
```



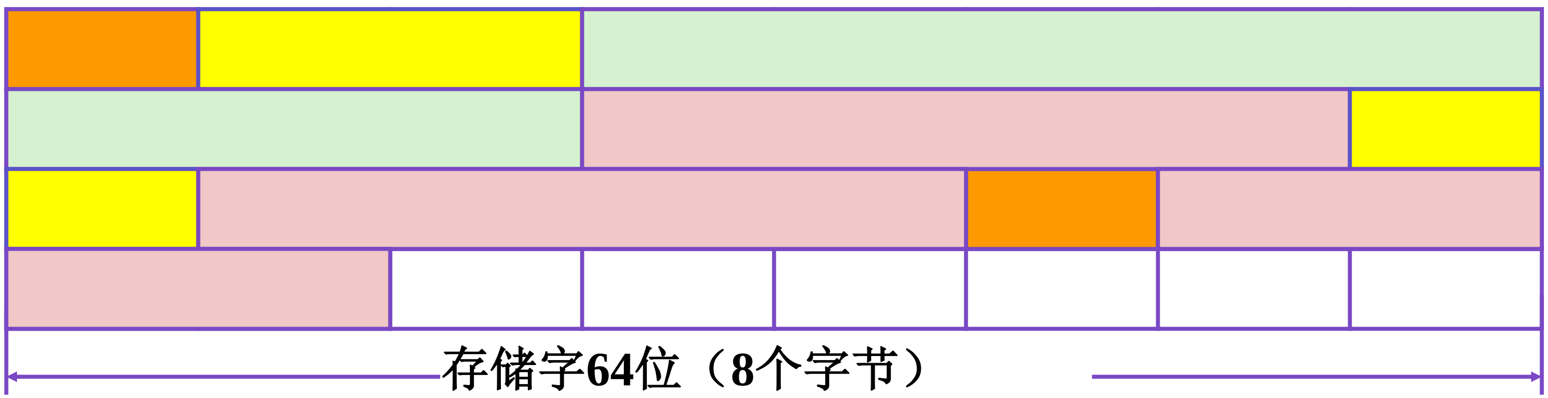
Register	Value
%rdi	r
%rsi	val

```
.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
```

对齐问题

不浪费存储器资源的存放方法

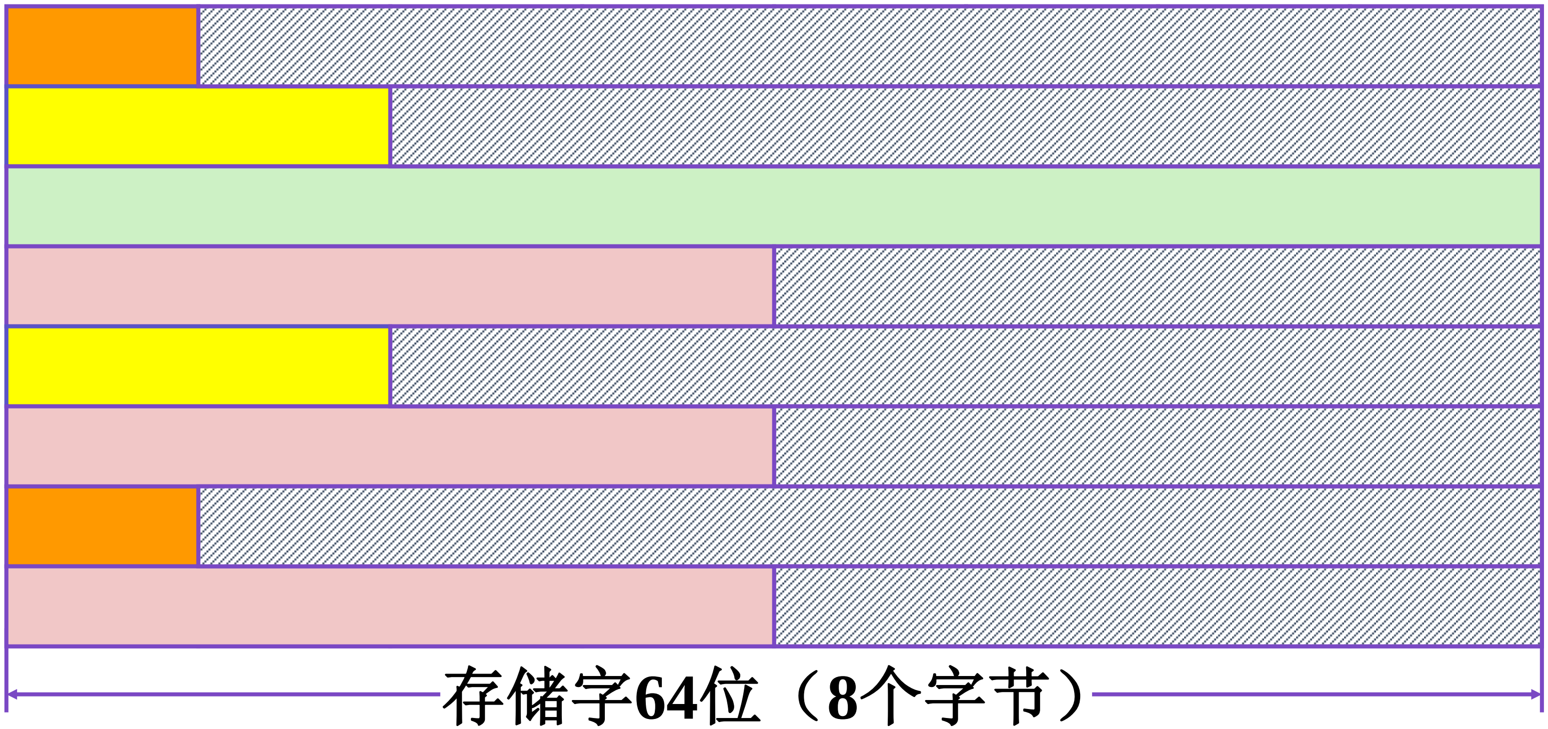
现有一批数据，它们依次为：字节、半字、双字、单字、半字、单字、字节、单字。4种不同长度的数据一个紧接着一个存放。



优点：不浪费宝贵的主存资源，

缺点：当访问的一个双字、单字或半字跨越两个存储单元时，存储器的工作速度降低了一半，而且读写控制比较复杂。

从存储字的起始位置开始存放

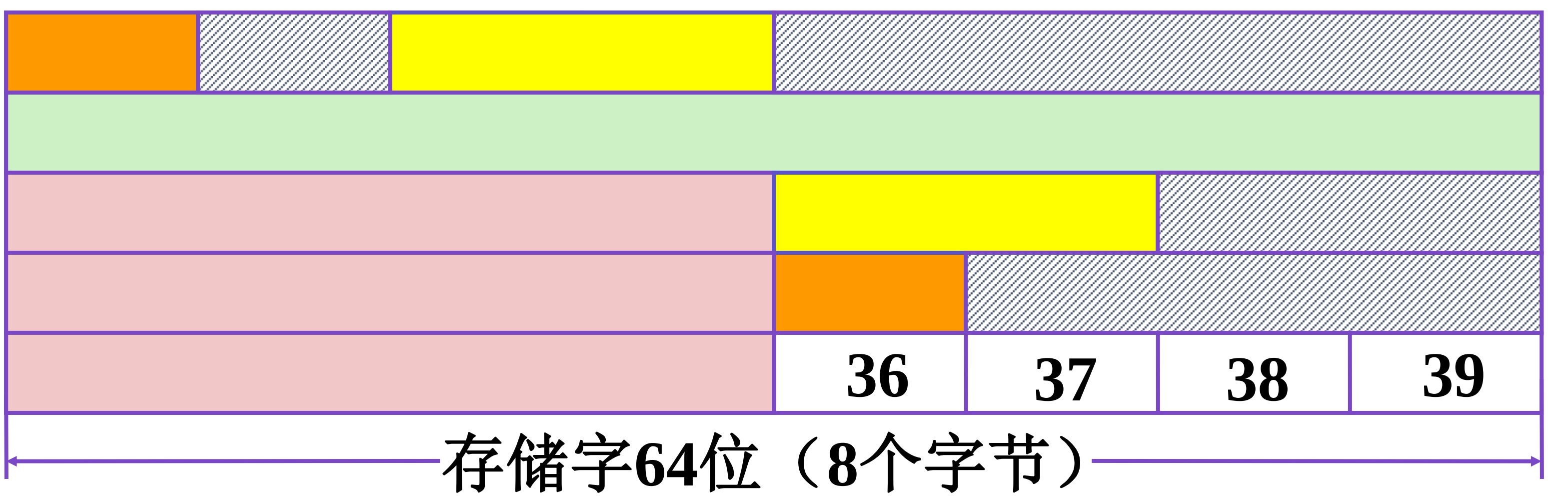


优点：无论访问一个字节、半字、单字或双字都可以在一个存储周期内完成，读写数据的控制比较简单。

缺点：浪费了宝贵的存储器资源。

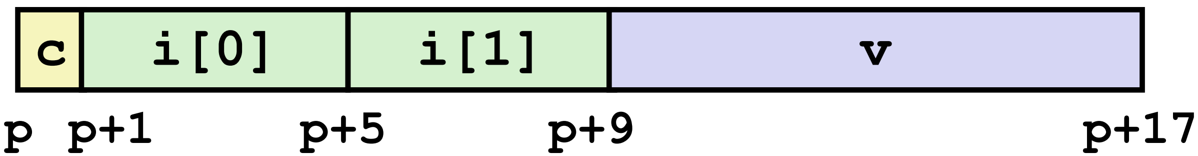
边界对齐的数据存放方法

此方法规定，双字地址的最末3个二进制位必须为000，单字地址的最末两位必须为00，半字地址的最末一位必须为0。它能够保证无论访问双字、单字、半字或字节，都在一个存取周期内完成，尽管存储器资源仍然有浪费。



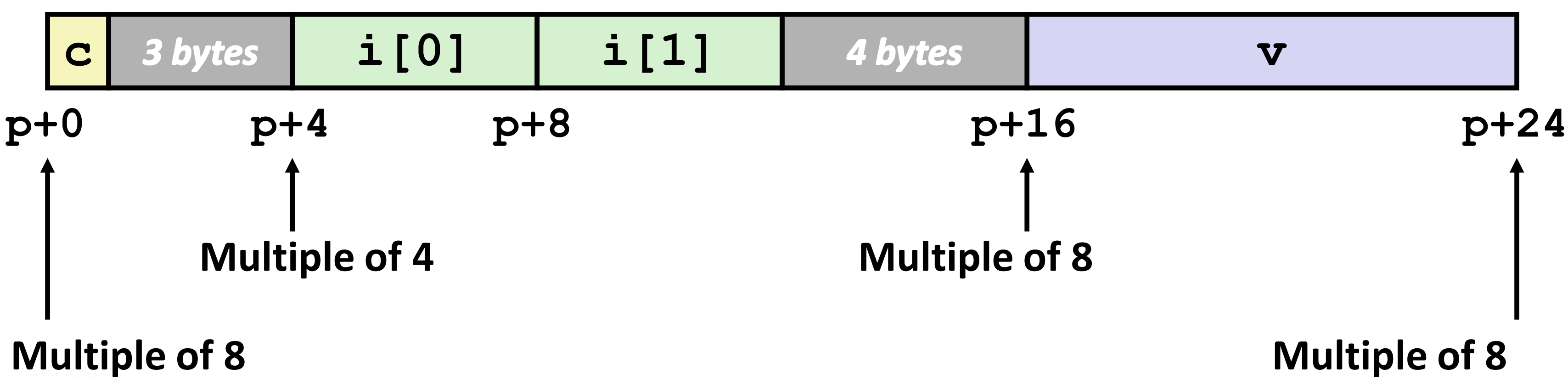
Structures & Alignment

- Unaligned Data




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

- Aligned Data
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***



Alignment Principles

- Aligned Data
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***
  - Required on some machines; advised on x86-64

编译会加入填充

- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

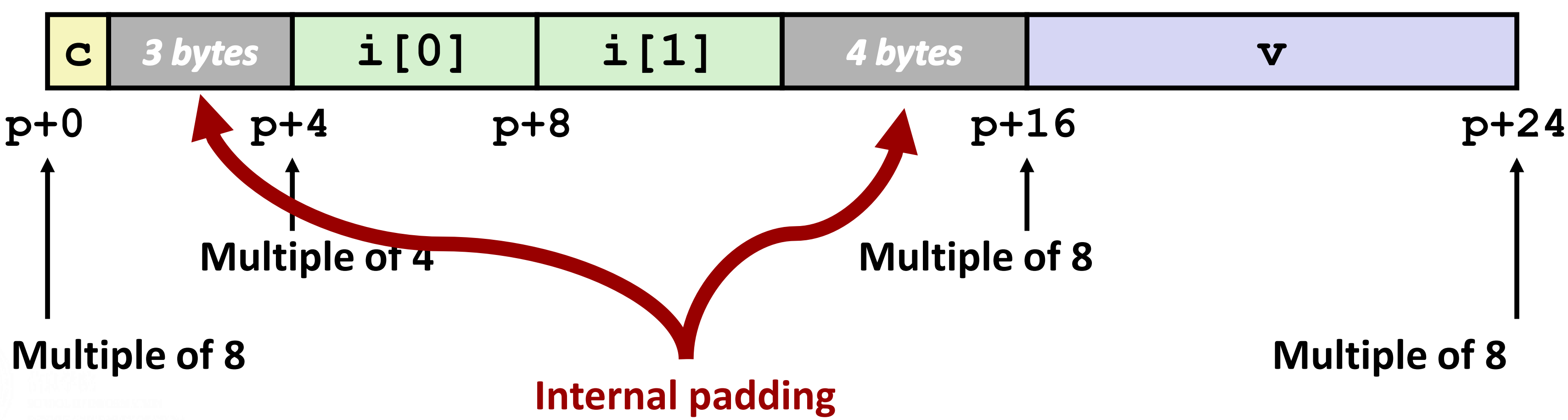
Specific Cases of Alignment

- 1 byte: **`char`**, …
  - no restrictions on address
- 2 bytes: **`short`**, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: **`int`**, **`float`**, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: **`double`**, `long`, **`char *`**, …
  - lowest 3 bits of address must be 000_2
- 16 bytes: **`long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- Example:
  - **K** = 8, due to **`double`** element
    NOTE: K < sizeof(struct S1)

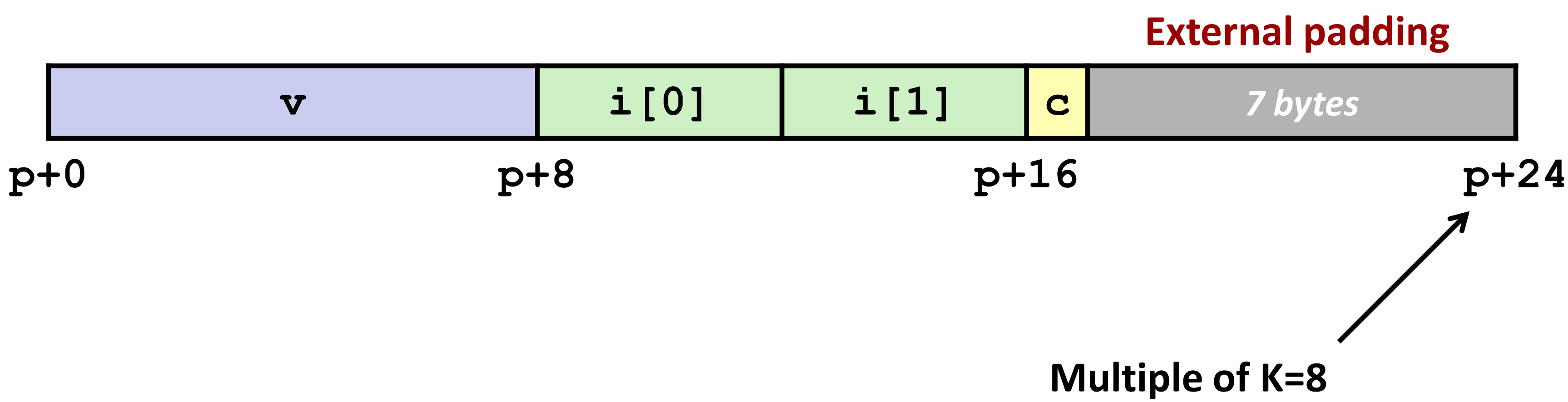
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Meeting Overall Alignment Requirement

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

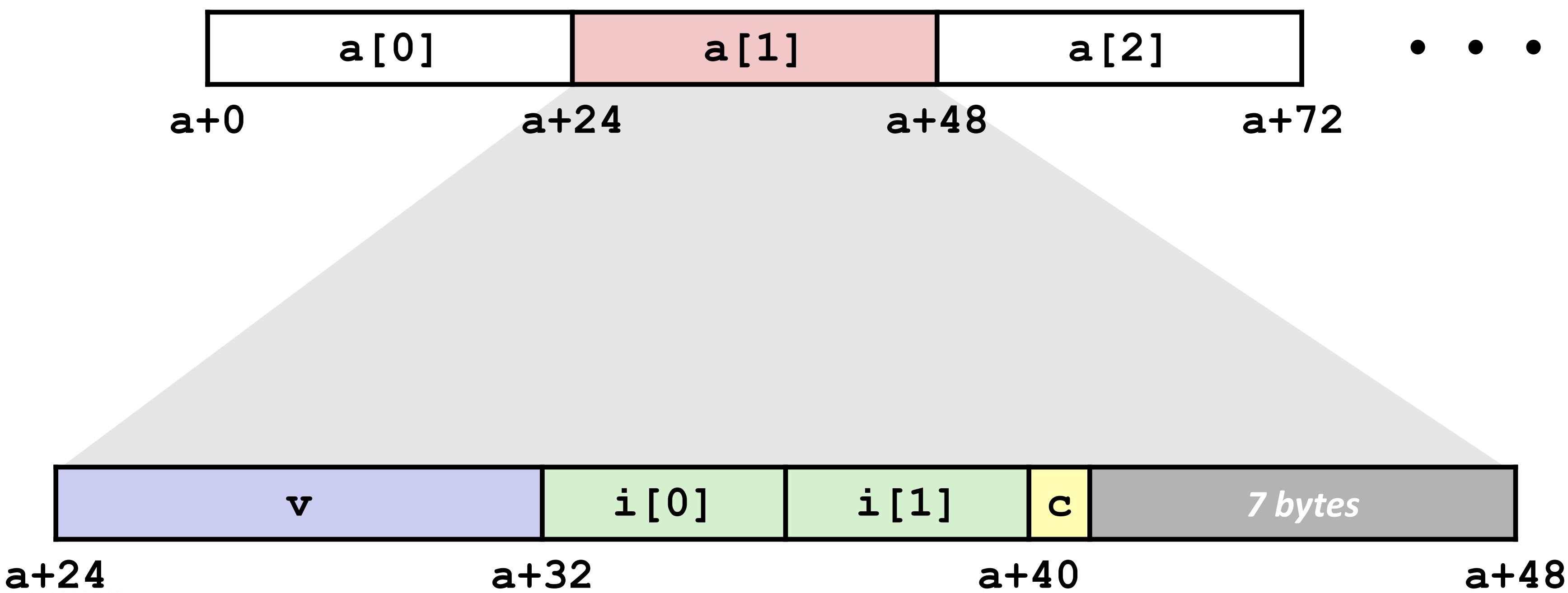
- For largest alignment requirement K
- Overall structure must be multiple of K



Arrays of Structures

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

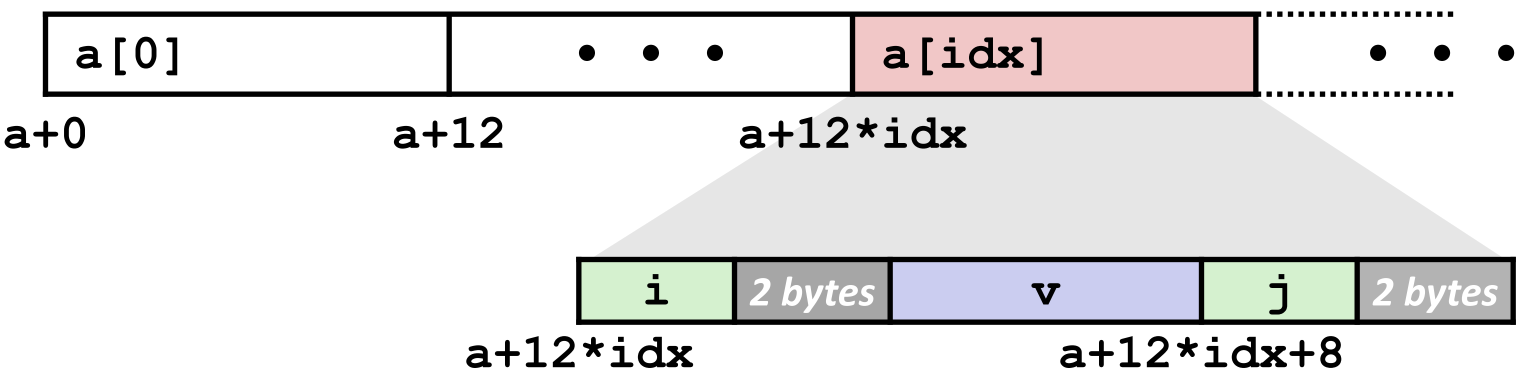
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

- Compute array offset 12*idx
  - **`sizeof(S3)`**, including alignment spacers
- Element **`j`** is at offset 8 within structure
- Assembler gives offset **`a+8`**
  - Resolved during linking

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




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

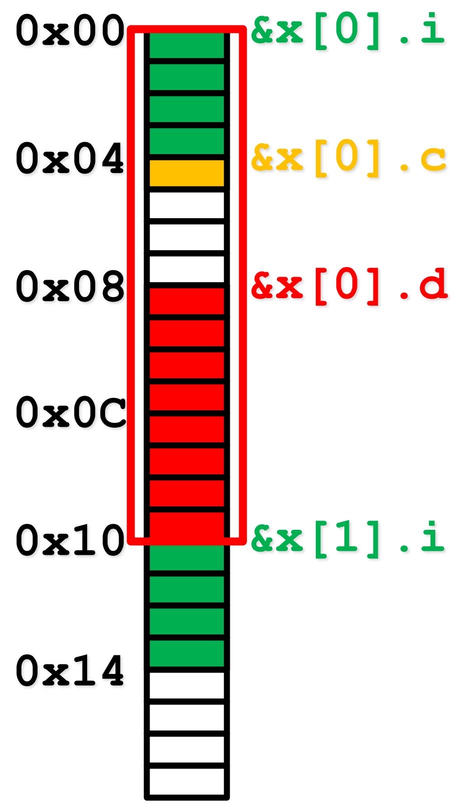
```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

Alignment

- Structure data type
  - may need to insert gaps in the field allocation
  - may need to add padding to the end of the structure

Simple Example

```
struct xxx {
  int i;
  char c;
  double d;
};

struct xxx x[2];
```

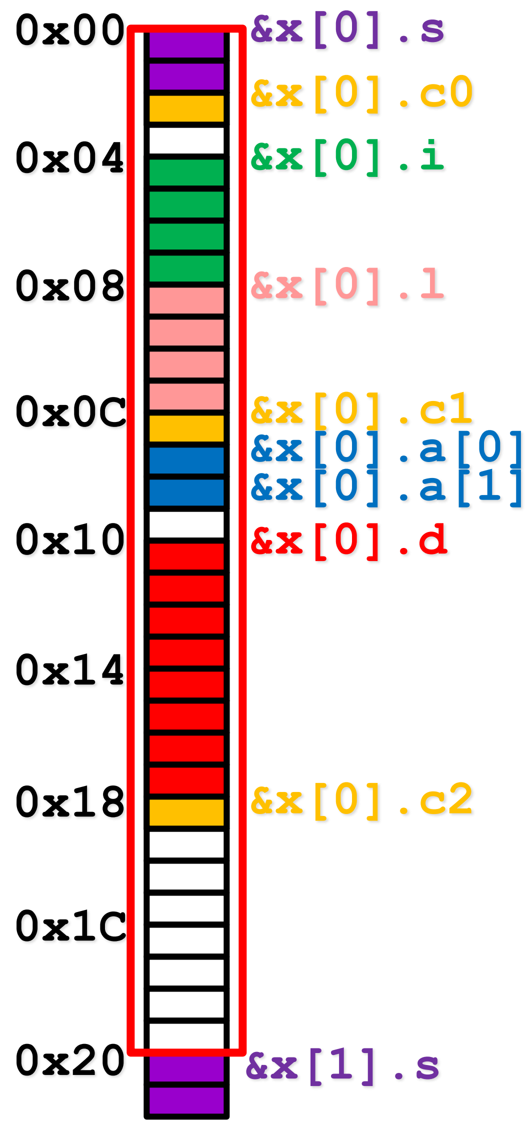


Struct整体对齐规则由其中最大的元素决定（此例为8字节）

40

Complex Example

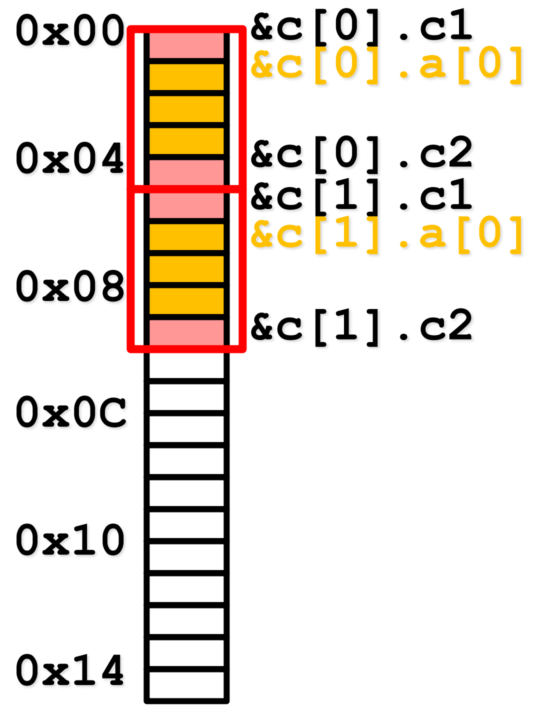
```
struct xxx {
    short s;
    char c0;
    int i;
    long l;
    char c1;
    char a[2];
    double d;
    char c2;
};

struct xxx x[2];
```



41

Array

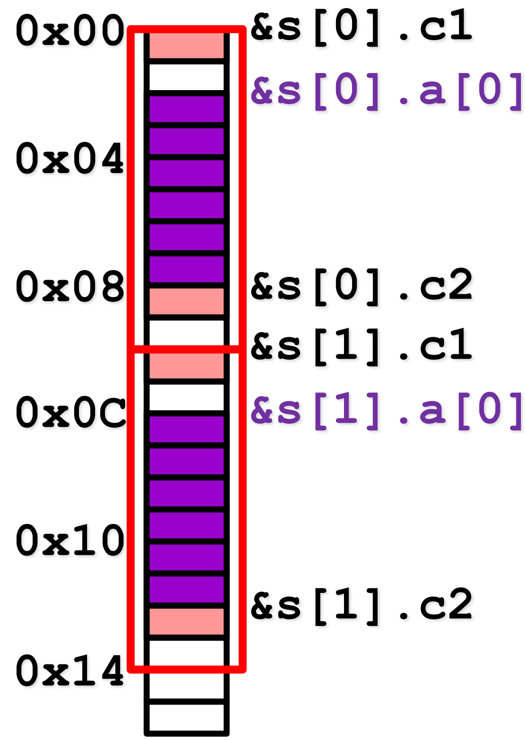
```
struct ccc {
    char c1;
    char a[3];
    char c2;
};

struct ccc c[2];
```



42

Array

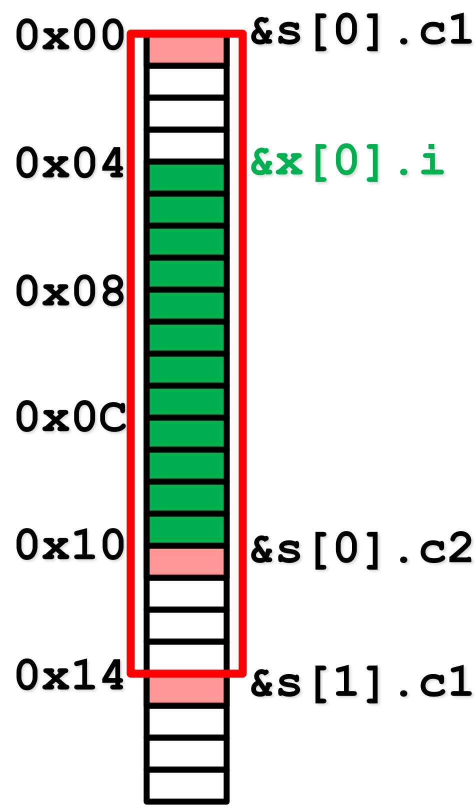
```
struct ccc {
    char c1;
    short a[3];
    char c2;
};

struct sss s[2];
```



43

Array

```
struct iii {
    char c1;
    int a[3];
    char c2;
};

struct iii i[2];
```



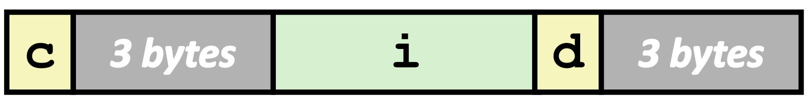
44

Saving Space

- Put large data types first

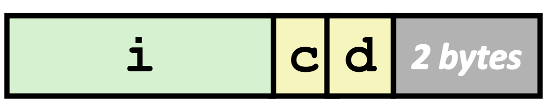
```c
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

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



12 bytes

- Effect (largest alignment requirement K=4)



8 bytes

Union

- 两个/多个部分，同一段内存有不同的解读方式
- A single object can be referenced by using different data types
- The syntax of a union declaration is identical to that for structures, but its semantics are very different
- Rather than having the different fields reference different blocks of memory, they all reference the same block

Union

```
struct S3 {
  char c;
  int i[2];
  double v;
};
union U3 {
  char c;
  int i[2];
  double v;
};
```

The offsets of the fields, as well as the total size of data types S3 and U3, are:

Type	c	i	v	size
S3	0	4	12	20
U3	0	0	0	8

提纲

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

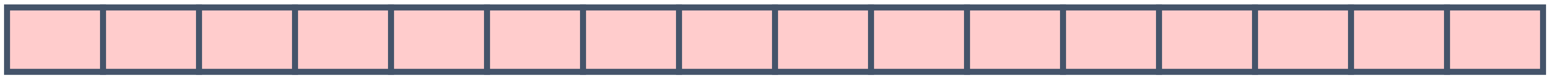
Background

- History
  - x87 FP
    - Legacy, very ugly
  - SSE FP
    - Supported by server machines
    - Special case use of vector instructions
  - AVX FP
    - Newest version
    - Similar to SSE
    - Documented in book

Programming with SSE3

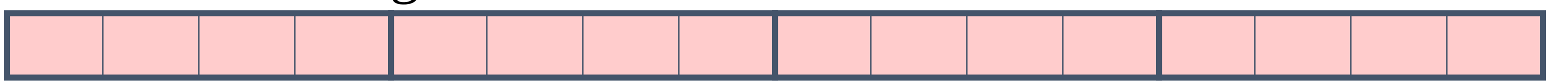
XMM Registers

- 16 total, each 16 bytes
- 16 single-byte integers

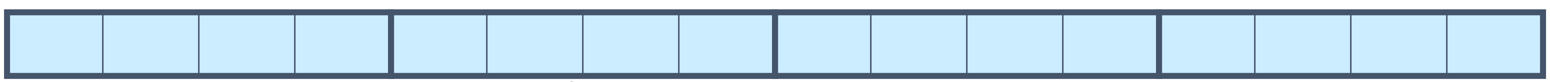


- 8 16-bit integers

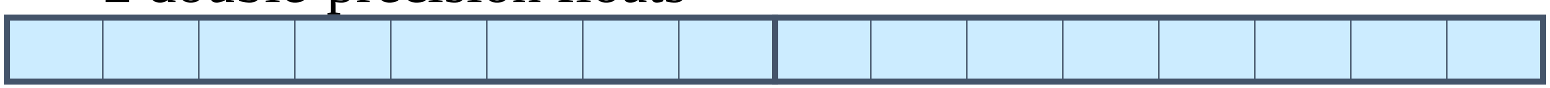
- 4 32-bit integers



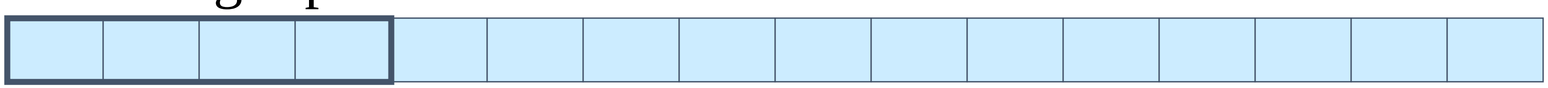
- 4 single-precision floats



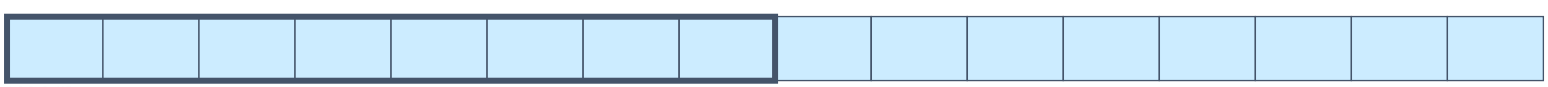
- 2 double-precision floats



- 1 single-precision float

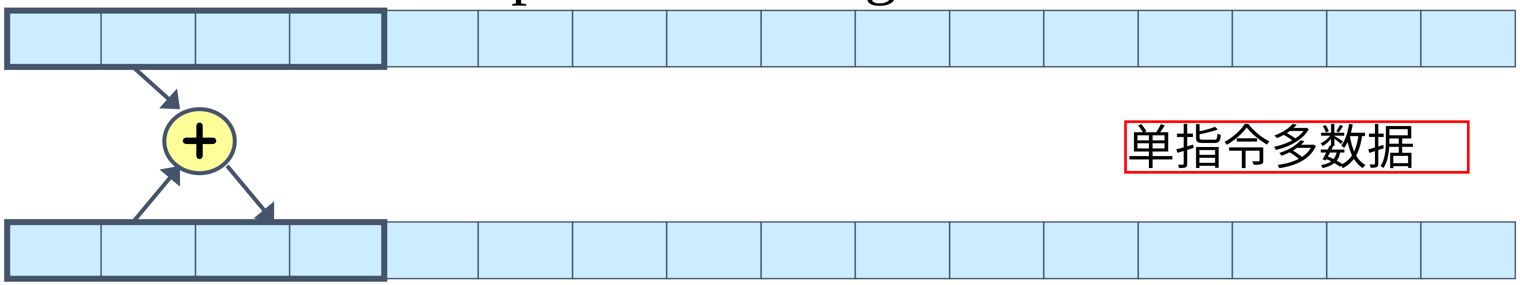


- 1 double-precision float



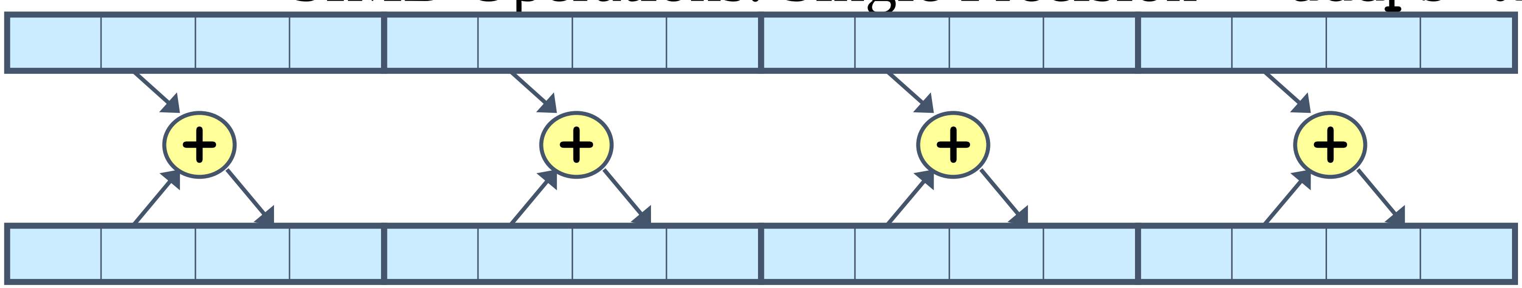
Scalar & SIMD Operations

- Scalar Operations: Single Precision `addss %xmm0,%xmm1`

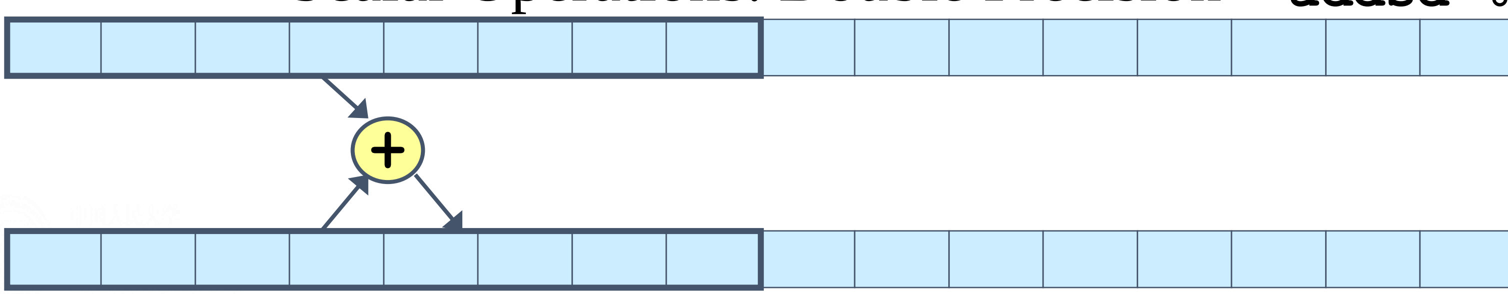


单指令多数据

- SIMD Operations: Single Precision `addps %xmm0,%xmm1`



- Scalar Operations: Double Precision `addsd %xmm0,%xmm1`



FP Basics

- Arguments passed in %xmm0, %xmm1, ...
- Result returned in %xmm0
- All XMM registers caller-saved

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

```
  # x in %xmm0, y in %xmm1
  addss    %xmm1, %xmm0
  ret
```

```
  # x in %xmm0, y in %xmm1
  addsd    %xmm1, %xmm0
  ret
```

FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different mov instructions to move between XMM registers, and between memory and XMM registers

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
  # p in %rdi, v in %xmm0
  movapd  %xmm0, %xmm1   # Copy v
  movsd   (%rdi), %xmm0  # x = *p
  addsd   %xmm0, %xmm1   # t = x + v
  movsd   %xmm1, (%rdi)  # *p = t
  ret
```

Other Aspects of FP Code

- *Lots* of instructions
  - Different operations, different formats, ...
- Floating-point comparisons
  - Instructions **`ucomiss`** and **`ucomisd`**
  - Set condition codes ZF, PF and CF
  - Zeros OF and SF

Parity Flag

UNORDERED: ZF,PF,CF←111
GREATER_THAN: ZF,PF,CF←000
LESS_THAN: ZF,PF,CF←001
EQUAL: ZF,PF,CF←100

- Using constant values
  - Set XMM0 register to 0 with instruction **`xorpd %xmm0, %xmm0`**
  - Others loaded from memory

课堂练习

- 下面的定义声明了一类结构，用来构建二叉树：

```
typedef struct ELE *tree_ptr;          // 表示tree_ptr实际上是ELE*类型
struct ELE {
        tree_ptr    left;
        tree_ptr    right;
        long        val;
}
```

- 对于如下函数原型 long trace(tree_ptr tp); GCC产生了下面的x86-64代码：

```
trace:                          ; tp in %rdi
        movl    $0, %eax
        testq   %rdi, %rdi
        je      .L3
.L5
        movq    16(%rdi), %rax
        movq    (%rdi), %rdi
        testq   %rdi, %rdi
        jne     .L5
.L3
        ret     ;函数返回，返回值一般放在%rax中
```

- （1）请写出该函数的最简洁的C语言版本，使用while循环；（2）用自然语言解释该函数的功能。

课堂练习

你负责维护一个大型的C程序，遇到下面的代码：　编译时常数CNT和结构a_struct的声明是在一个你没有访问权限的文件中。

```c
typedef struct {
    int first;
    a_struct a[CNT];
    int last;
}b_struct;
void test(long i, b_struct *bp) {
     int n = bp->first + bp->last;
     a_struct *ap = &bp->a[i];
     ap->x[ap->idx] = n;
}
```

幸好你有代码的.o版本，反汇编的代码为：

void test(long i, b_struct *bp) {

i in %rdi, bp in %rsi

```
0000000000000 <test>:
0: 8b 8e 20 01 00 00     mov 0x120(%rsi), %ecx
6: 03 0e                 add (%rsi), %ecx
8: 48 8d 04 bf           lea (%rdi, %rdi, 4), %rax
c: 48 8d 04 c6           lea (%rsi, %rax, 8), %rax
10: 48 8b 50 08          mov 0x8(%rax), %rdx
14: 48 63 c9             movslq %ecx, %rcx
17: 48 89 4c d0 10       mov %rcx, 0x10(%rax, %rdx, 8)
1c: c3.                  retq
```

请推断：

A. CNT的值；　$\boxed{40？}$

B. 结构体a_struct的完整声明，假设其中只有字段idx和x，且都是有符号数

课堂练习

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

```
// 32位机环境下
//up@eax, dest@edx
void get(u_type *up, TYPE *dest) {
    *dest = EXPR;
}
```

EXPR分为为以下值时，求TYPE和get函数的汇编代码:

1. up->t1.s
2. up->t1.v
3. &up->t1.d
4. up->t2.a
5. up->t2.a[up->t1.s]
6. *up->t2.p

练习答案

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

```
// 32位机环境下
//up@eax, dest@edx
void get(u_type *up, TYPE *dest) {
    *dest = EXPR;
}
```

EXPR分为为以下值时，求TYPE和get函数的汇编代码:

1. up->t1.s
2. up->t1.v
3. &up->t1.d
4. up->t2.a
5. up->t2.a[up->t1.s]
6. *up->t2.p

1）int, movl 4(%eax), %eax　　movl %eax, (%edx)

2）short, movw (%eax),%ax　　movw %ax, (%edx)

3）short *, leal 2(%eax), %eax　　movl %eax, (%edx)

4）int *, movl %eax, (%edx)

5）int, movl 4(%eax), %ecx

movl (%eax, %ecx, 4), %eax　　movl %eax, (%edx)

6）char, movb 8(%eax), %al　　movb al, (%edx)

Summary

- Arrays
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements
- Structures
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment
- Combinations
  - Can nest structure and array code arbitrarily
- Floating Point
  - Data held and operated on in XMM registers