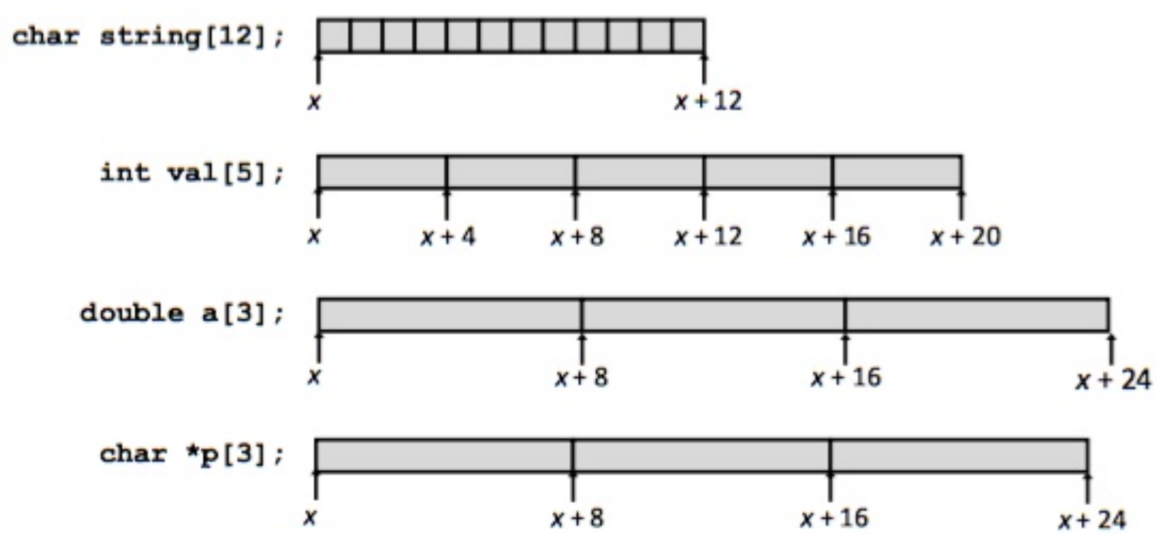
Machine-Level Programming IV: Data

arrays,assembly,C,C++,structs

Array Allocation

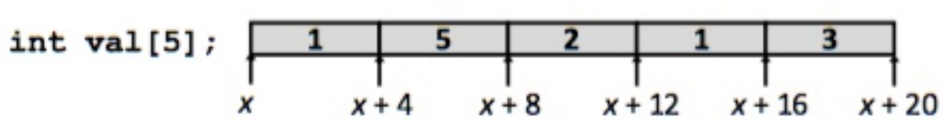
- `T A[L];`
  - array of data type T and length L
  - contiguously allocated region of `L * sizeof(T)` bytes in memory



- nested arrays: `A[n][n]`
  - static
    - compiler can optimize this very well
  - dynamic
    - hard for compiler to optimize
- multi-level
  - i.e. array of pointers to arrays (like hash tables)

Array Access

- identifier A can be used as a pointer to array element 0: Type T*



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 *`	$x+4i$

Array Loop Example

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

```asm
; %rdi = z

movl    $0, %eax
jmp     .L3

.L4:
    addl    $1, (%rdi,%rax,4)
    addq    $1, %rax
.L3:
    cmpq    $4, %rax
    jbe     .L4
    ret
```

Multidimensional (Nested) Arrays

Declaration

`T A[R][C]`

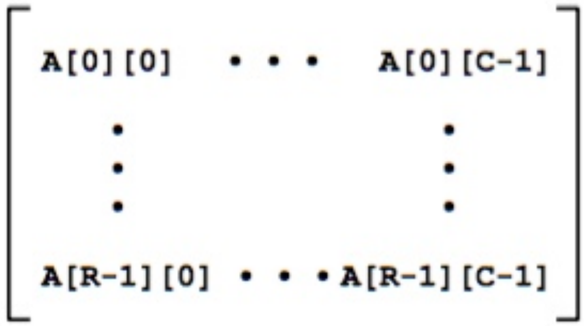
- 2D array of data type T
- R rows, C columns
- type T element requires K bytes

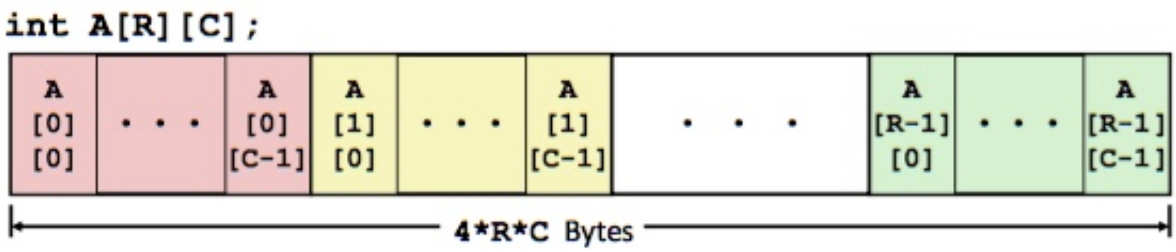
Array Size

- R * C * K bytes

Arrangement

- row-major ordering

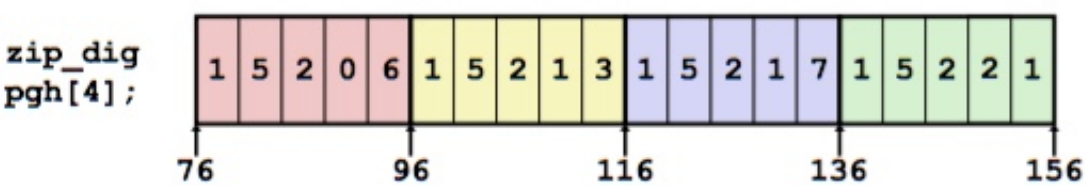




- **each color is a separate row**

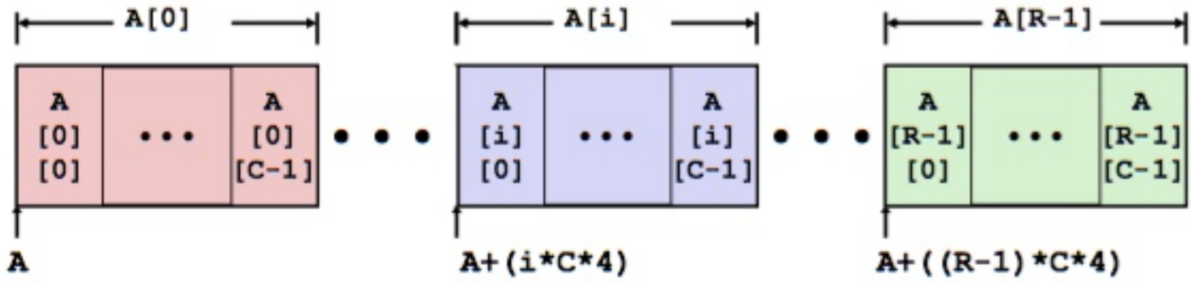
Nested Array Example

```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}};
```



Nested Array Row Access

- `A[i]` is array of C elements
- each element of type T requires K bytes
- `A + i * (C * K)`



Access Code

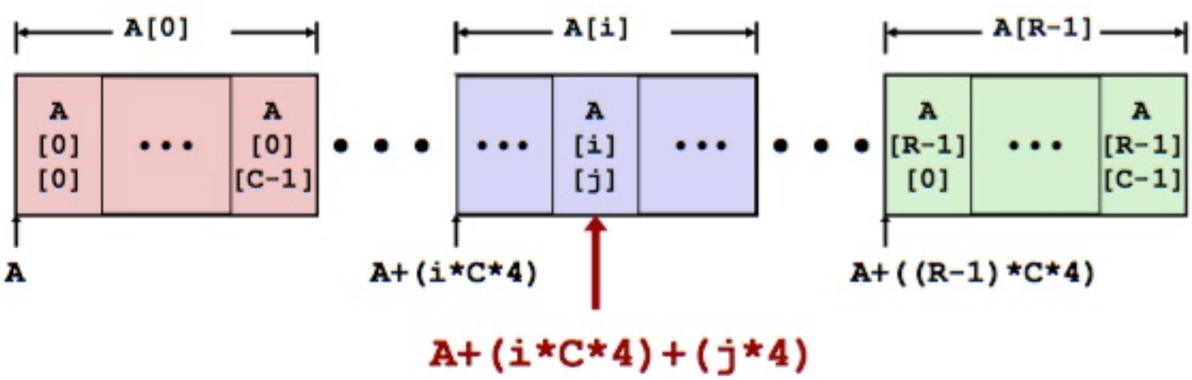
```c
int *get_pgh_zip(int index) {
    return pgh[index];
}
```

```
; %rdi = index
; leaq doesn't dereference the contents at the address
; just puts memory location into %rax
leaq    (%rdi,%rdi,4), %rax     ; 5 * index into %rax
leaq    pgh(,%rax,4), %rax      ; pgh + (20 * index) into %rax
```

Nested Array Element Access

- `A[i][j]` is element of type T, which requires K bytes
- **address**: `A + (i*(C*K)) + (j*K) = A + (i*C+j)*K`

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

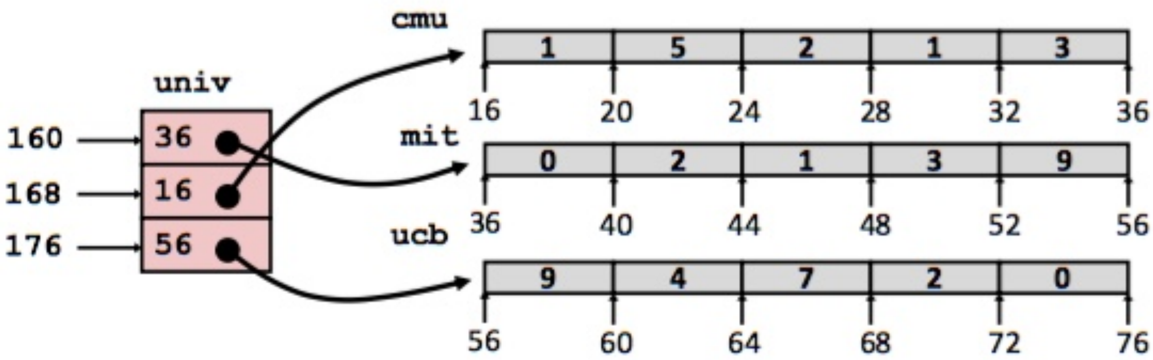


Access Code

```c
int get_pgh_digit(int index, int dig) {
    return pgh[index][dig];
}
```

```
leaq    (%rdi,%rdi,4), %rax     ; 5*index into %rax
addl    %rax, %rsi              ; (5*index) + dig
movl    pgh(,%rsi,4), %eax      ; M[pgh + 4*(5*index)+dig]
```

Multi-Level Array Example




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

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

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

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

Must do two memory reads

1. to get pointer to row array
2. then access element within array

Less efficient than nested memory access, which only requires ONE memory load

N x N Matrix Access

TAKE ANOTHER LOOK AT THESE

```
/* 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
```

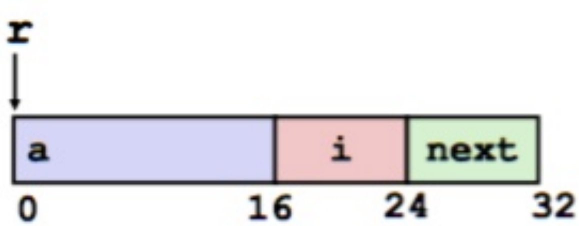
Questions

1. Comparison between nested arrays, multi-level arrays
2. N x N matrix

Structures

Structure Representation

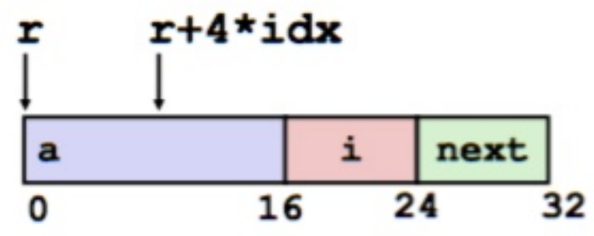
```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- 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 + position of fields
  - machine-level program has no understanding of the structures in the source code
- can access different elements of struct using pointer arithmetic

- data laid out linearly, the same way every time

Generating Pointer to Structure Member




```c
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
```

Generating Pointer to Array Element

- offset of each structure member determined at compile time
- compute as `r + 4*idx`

Following Linked List

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

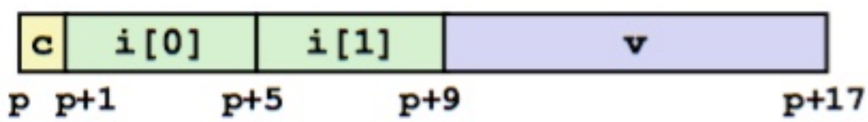
```
; r in %rdi, val in %rsi
.L11:                                   ; loop:
    movslq  16(%rdi), %rax              ;   i = M[r+16]
    movl    %esi, (%rdi,%rax,4)         ;   M[r+4*i] = val
    movq    24(%rdi), %rdi              ;   r = M[r+24]
    testq   %rdi, %rdi                  ;   Test r
 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

Structures & Alignment

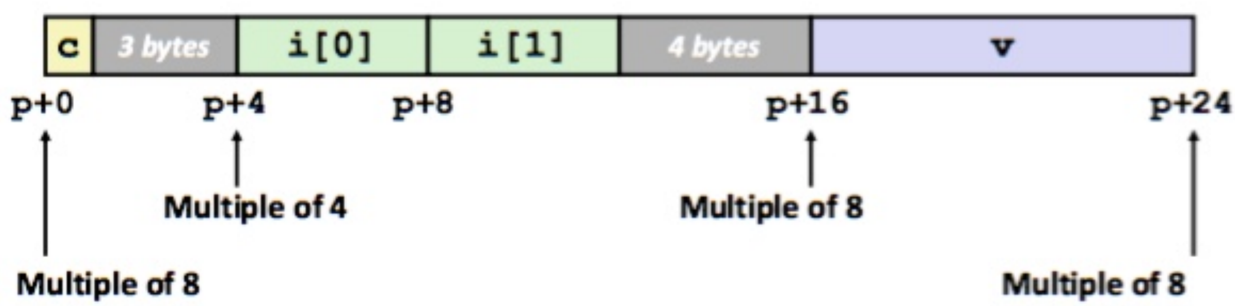
```c
struct S1 {
    char c;     // 1 1-byte quantity
```

```c
    int i[2];   // 2 4-bytes quantities
    double v;   // 8-byte quantities
}
```

Unaligned



Aligned



Aligned Data

- primitive data type requires K bytes
- address must be mutliple of K
  - char requires 1 byte, can be anywhere
  - int requires 4 bytes, must be at address multiple of 4
  - double requires 8 bytes, must be at address multiple of 8
- rqeuired on some machines; advised on x86-64

Motivation for Aligning Data

- memory access 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

Satisfying Alignment with Structures

Within Structure

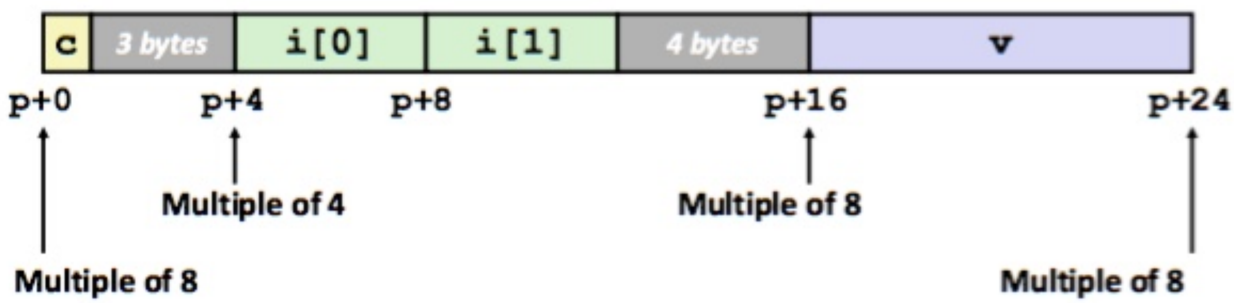
- must satisfy each element's alignment requirement

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

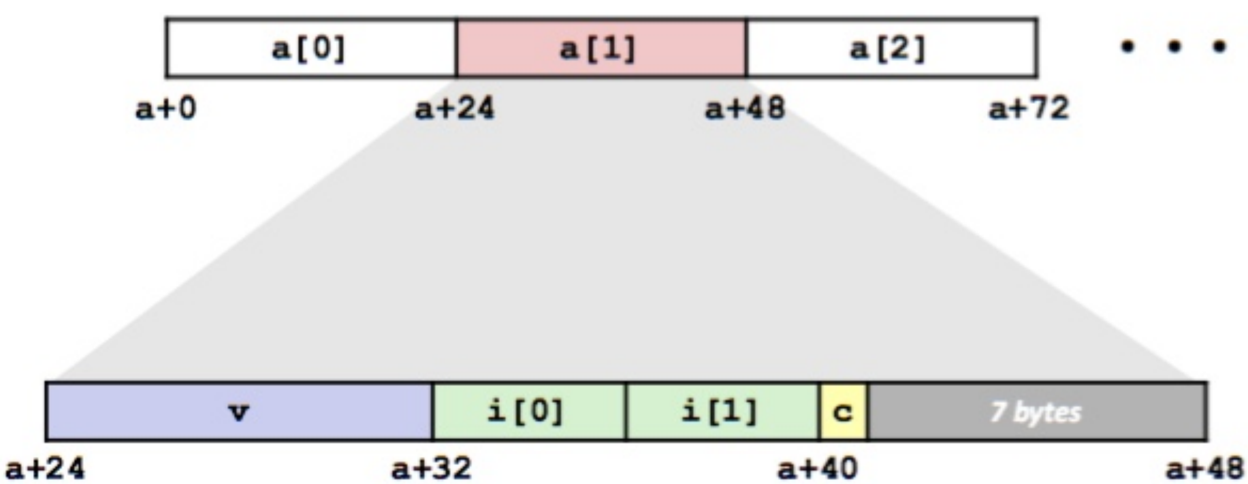
- K = 8, due to `double` element



Arrays of Structures

- overall structure length multiple of K
    - **array has 7 bytes of padding to meet K=8 requirement**
- satisfy alignment requirement for every element

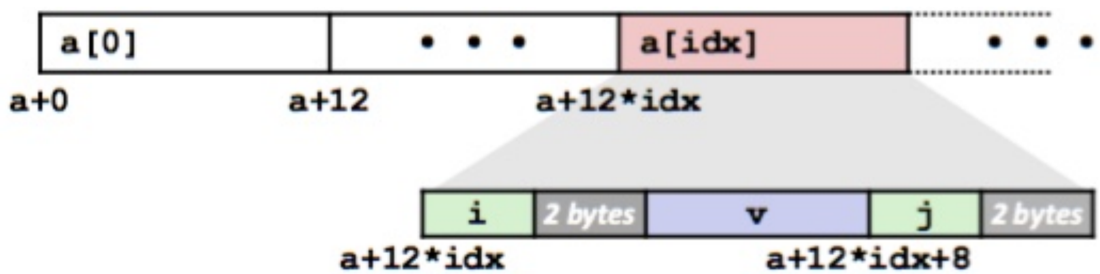
```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



Accessing Array Elements

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

- 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




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

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

Saving Space

- **put large data types first**