

# 15-213

*“The course that gives CMU its Zip!”*

## **Structured Data I: Homogenous Data Sept. 16, 1999**

### **Topics**

- **Arrays**
  - Single
  - Nested
- **Pointers**
  - Multilevel Arrays
- **Optimized Array Code**

# Basic Data Types

## Integral

- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int

## Floating Point

- Stored & operated on in floating point registers

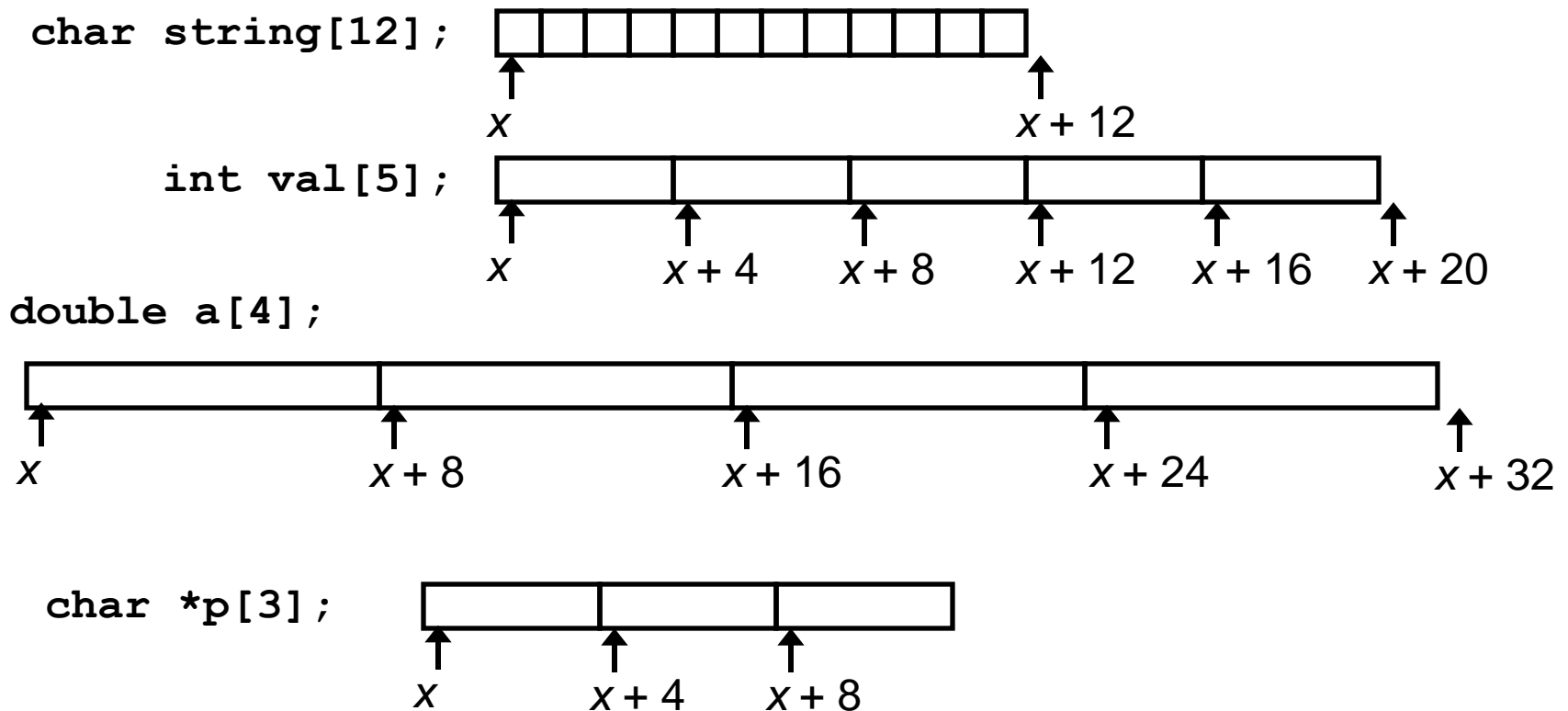
Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended		10	--

# Array Allocation

## Basic Principle

$T$   $A[L]$ ;

- Array of data type  $T$  and length  $L$
- Contiguously allocated region of  $L * \text{sizeof}(T)$  bytes

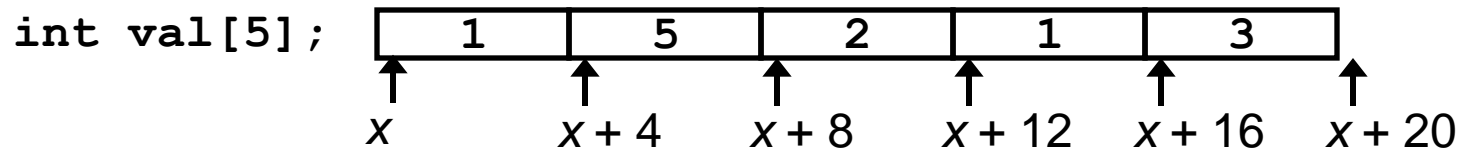


# Array Access

## Basic Principle

$T$  A[L];

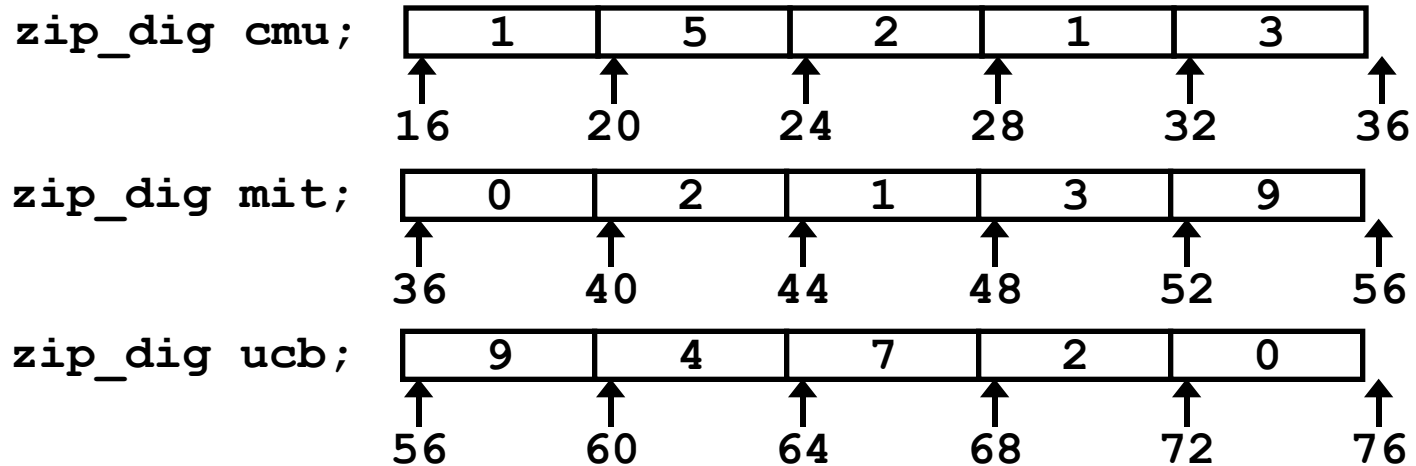
- Array of data type  $T$  and length  $L$
- Identifier  $A$  can be used as a pointer to starting element of the array



Reference	Type	Value
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	$x$
<code>val+1</code>	<code>int *</code>	$x+4$
<code>&amp;val[2]</code>	<code>int *</code>	$x+8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	$x+4i$

# Array Example

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



## Notes

- 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

## Computation

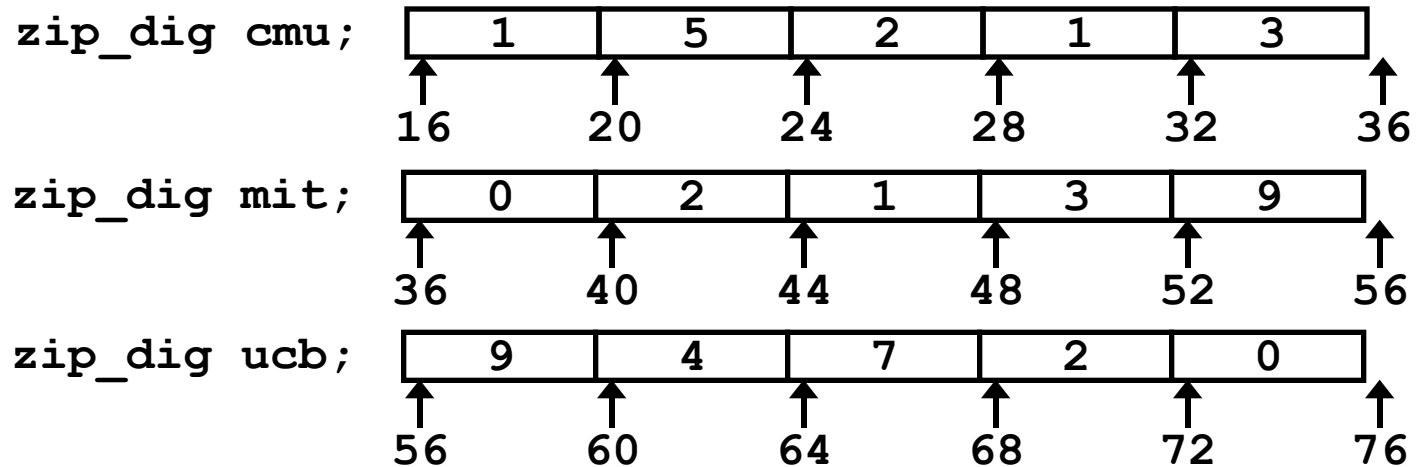
- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at  $4 * \%eax + \%edx$
- Use memory reference  
`(%edx, %eax, 4)`

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

## Memory Reference Code

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```

# Referencing Examples



## Code Does Not Do Any Bounds Checking!

Reference	Address	Value	Guaranteed?
<code>mit[3]</code>	$36 + 4 * 3 = 48$	3	Yes
<code>mit[5]</code>	$36 + 4 * 5 = 56$	9	No
<code>mit[-1]</code>	$36 + 4 * -1 = 32$	3	No
<code>cmu[15]</code>	$16 + 4 * 15 = 76$	??	No

- **Out of range behavior implementation-dependent**
  - No guranteed relative allocation of different arrays

# Array Loop Example

## Original Source

```
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

## Transformed Version

- Eliminate loop variable *i*
- Convert array code to pointer code
- Express in do-while form
  - No need to test at entrance

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

# Array Loop Implementation

## Registers

`%ecx` `z`  
`%eax` `zi`  
`%ebx` `zend`

## Computations

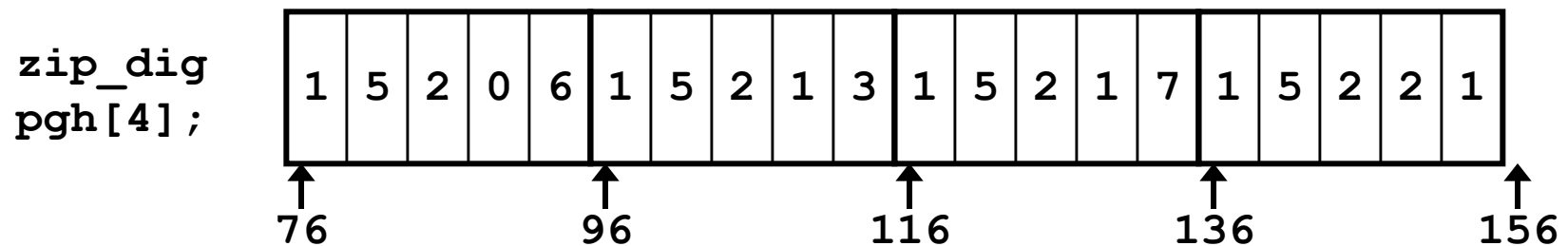
- $10 * z_i + *z$   
implemented as  
 $*z + 2 * (z_i + 4 * z_i)$
- `z++` increments by 4

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

```
# %ecx = z
xorl %eax,%eax          # zi = 0
leal 16(%ecx),%ebx      # zend = z+4
.L59:
leal (%eax,%eax,4),%edx # 5*zi
movl (%ecx),%eax       # *z
addl $4,%ecx           # z++
leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
cmpl %ebx,%ecx         # z : zend
jle .L59               # if <= goto loop
```

# 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}};
```



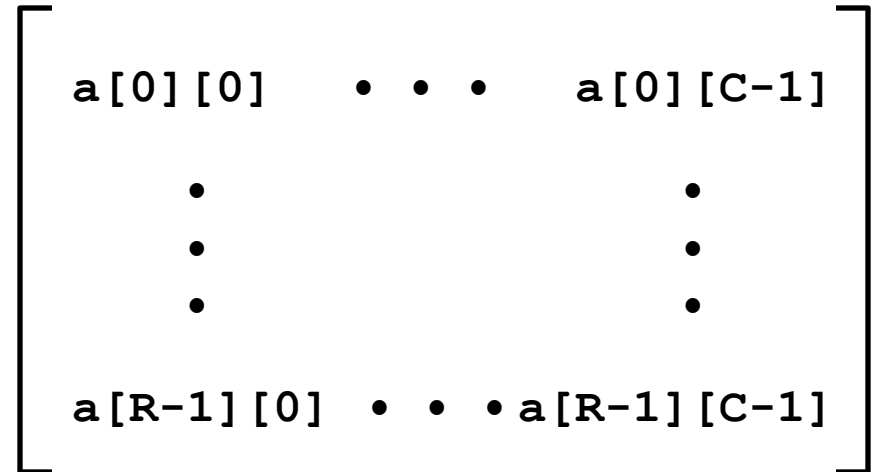
- Declaration “zip\_dig pgh[4]” equivalent to “int pgh[4][5]”
  - Variable `pgh` denotes array of 4 elements
    - » Allocated contiguously
  - Each element is an array of 5 `int`'s
    - » Allocated contiguously
- “Row-Major” ordering of all elements guaranteed

# Nested Array Allocation

## Declaration

```
T A[R][C];
```

- Array of data type  $T$
- $R$  rows
- $C$  columns
- Type  $T$  element requires  $K$  bytes



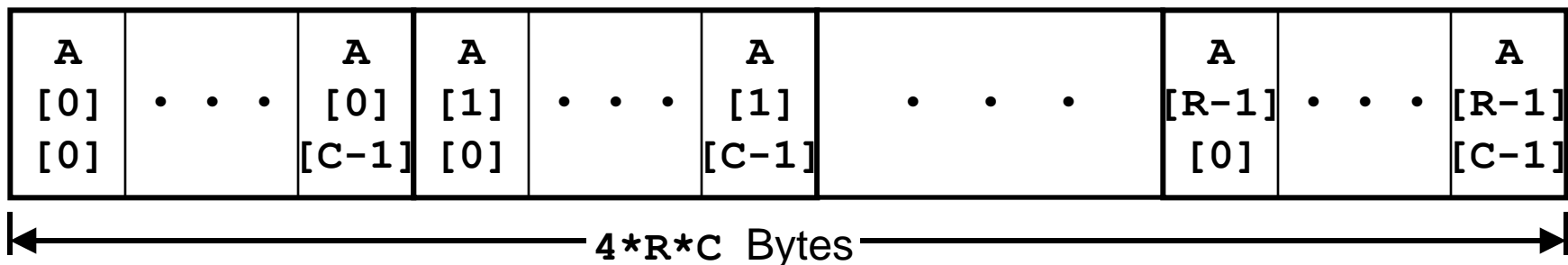
## Array Size

- $R * C * K$  bytes

## Arrangement

- Row-Major Ordering

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

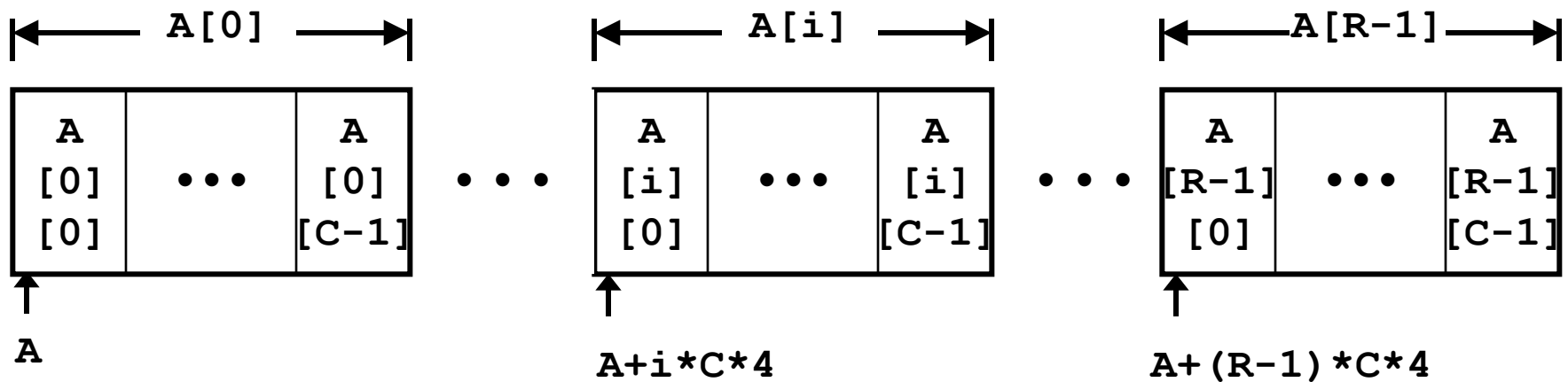


# Nested Array Row Access

## Row Vectors

- $A[i]$  is array of  $C$  elements
- Each element of type  $T$
- Starting address  $A + i * C * K$

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



# Nested Array Row Access Code

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

## Row Vector

- `pgh[index]` is array of 5 int's
- Starting address `pgh+20*index`

## Code

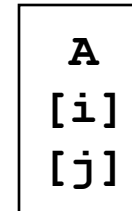
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

```
# %eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal pgh(,%eax,4),%eax # pgh + (20 * index)
```

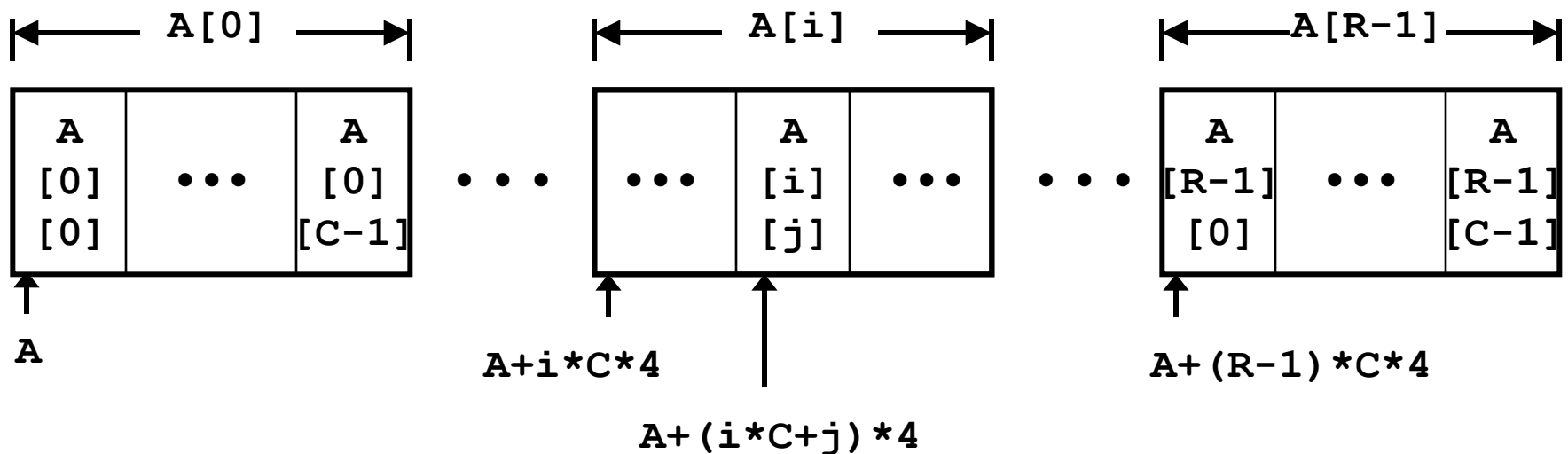
# Nested Array Element Access

## Array Elements

- $A[i][j]$  is element of type  $T$
- Address  $A + (i * C + j) * K$



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



# Nested Array Element Access Code

## Array Elements

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

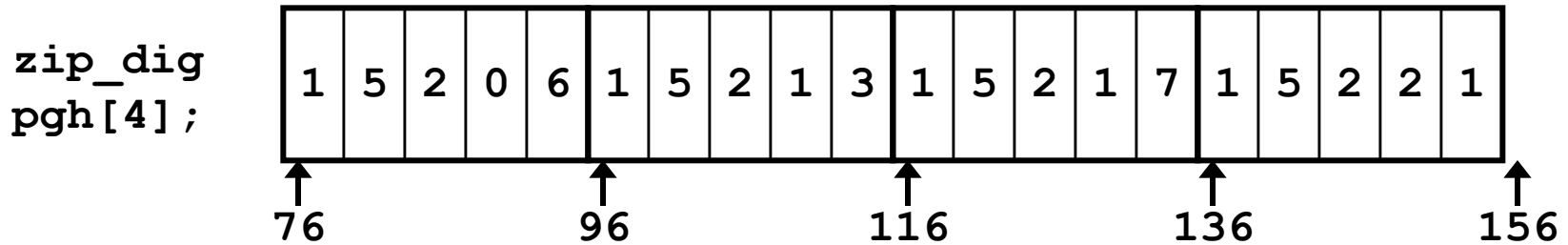
## Code

- Computes address  
 $pgh + 4 * dig + 4 * (index + 4 * index)$
- `movl` performs memory reference

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

```
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx          # 4*dig
leal (%eax,%eax,4),%eax      # 5*index
movl pgh(%edx,%eax,4),%eax   # *(pgh + 4*dig + 20*index)
```

# Strange Referencing Examples



Reference	Address	Value	Guaranteed?
pgh[3][3]	$76+20*3+4*3 = 148$	2	Yes
pgh[2][5]	$76+20*2+4*5 = 136$	1	Yes
pgh[2][-1]	$76+20*2+4*-1 = 112$	3	Yes
pgh[4][-1]	$76+20*4+4*-1 = 152$	1	Yes
pgh[0][19]	$76+20*0+4*19 = 152$	1	Yes
pgh[0][-1]	$76+20*0+4*-1 = 72$	??	No

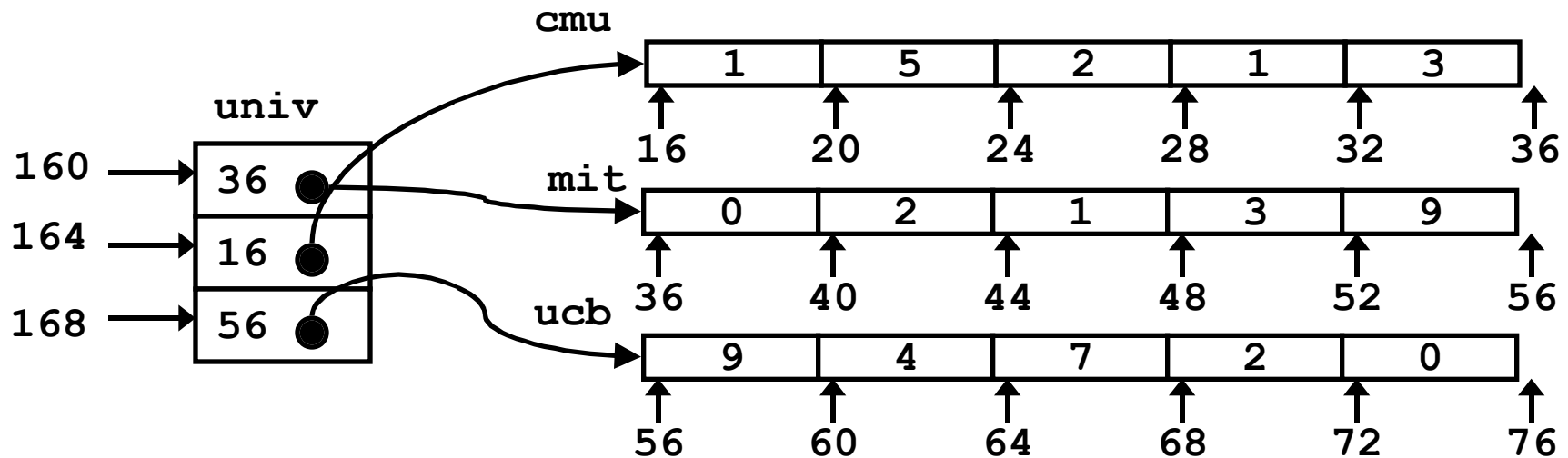
- Code does not do any bounds checking
- Ordering of elements within array guaranteed

# Multi-Level Array Example

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

```
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};
```



# Referencing “Row” in Multi-Level Array

## Row Vector

- `univ[index]` is pointer to array of `int`'s
- Starting address `Mem[univ+4*index]`

```
int* get_univ_zip(int index)
{
    return univ[index];
}
```

## Code

- Computes address within `univ`
- Reads pointer from memory and returns it

```
# %edx = index
leal 0(,%edx,4),%eax # 4*index
movl univ(%eax),%eax # *(univ+4*index)
```

# Accessing Element in Multi-Level Array

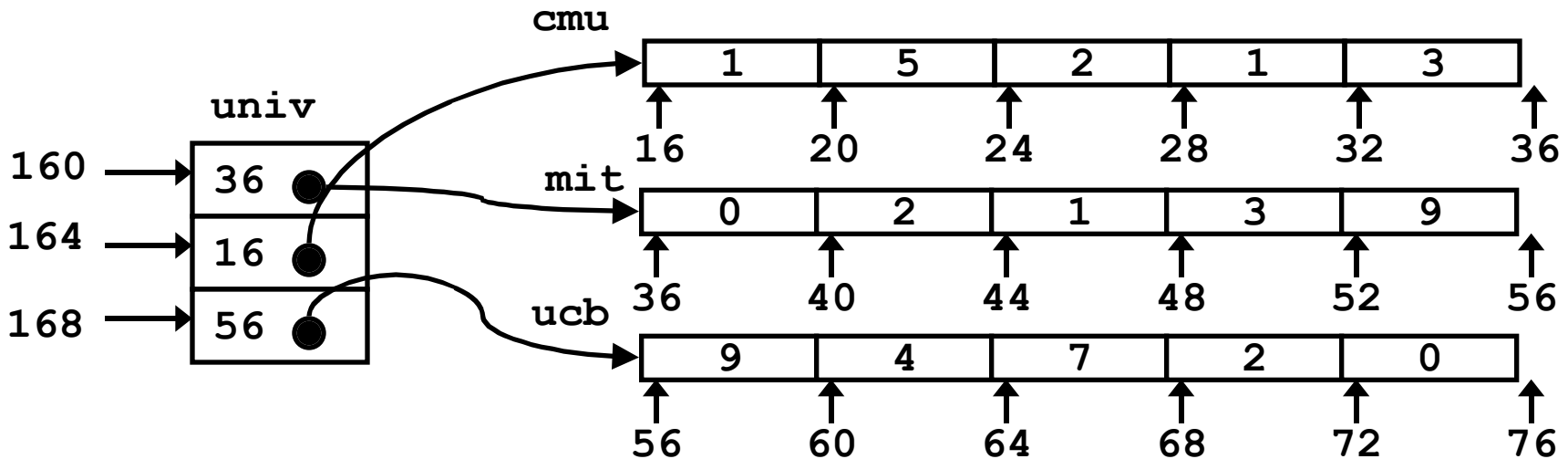
## Computation

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

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

```
# %ecx = index
# %eax = dig
leal 0(,%ecx,4),%edx      # 4*index
movl univ(%edx),%edx      # Mem[univ+4*index]
movl (%edx,%eax,4),%eax   # Mem[...+4*dig]
```

# Strange Referencing Examples



Reference	Address	Value	Guaranteed?
<code>univ[2][3]</code>	$56+4*3 = 68$	2	Yes
<code>univ[1][5]</code>	$16+4*5 = 36$	0	No
<code>univ[2][-1]</code>	$56+4*-1 = 52$	9	No
<code>univ[3][-1]</code>	??	??	No
<code>univ[1][12]</code>	$16+4*12 = 64$	7	No

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

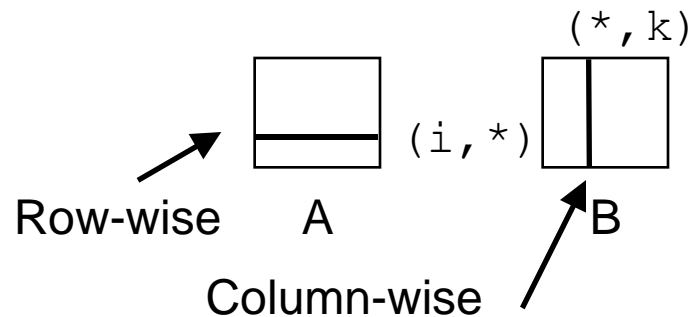
# Using Nested Arrays

## Strengths

- C compiler handles doubly subscripted arrays
- Generates very efficient code
  - Avoids multiply in index computation

## Limitation

- Only works if have fixed array size



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

```
/* Compute element i,k of
   fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

# Dynamic Nested Arrays

## Strength

- Can create matrix of arbitrary size

## Programming

- Must do index computation explicitly

## Performance

- Accessing single element costly
- Must do multiplication

```
int * new_var_matrix(int n)
{
    return (int *)
        calloc(sizeof(int), n*n);
}
```

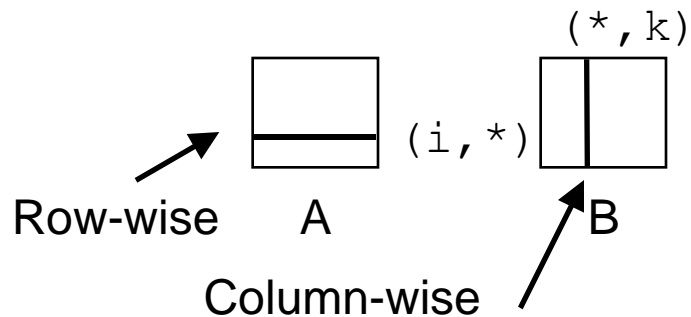
```
int var_ele
(int *a, int i,
 int j, int n)
{
    return a[i*n+j];
}
```

```
movl 12(%ebp),%eax    # i
movl 8(%ebp),%edx     # a
imull 20(%ebp),%eax   # n*i
addl 16(%ebp),%eax    # n*i+j
movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]
```

# Dynamic Array Multiplication

## Without Optimizations

- **Multiplies**
  - 2 for subscripts
  - 1 for data
- **Adds**
  - 4 for array indexing
  - 1 for loop index
  - 1 for data



```
/* Compute element i,k of
   variable matrix product */
int var_prod_ele
(int *a, int *b,
 int i, int k, int n)
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}
```

# Optimizing Dynamic Array Multiplication

## Optimizations

- Performed when set optimization level to -O2

## Code Motion

- Expression  $i*n$  can be computed outside loop

## Strength Reduction

- Incrementing  $j$  has effect of incrementing  $j*n+k$  by  $n$

## Performance

- Compiler can optimize regular access patterns

```
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}
```

```
{
    int j;
    int result = 0;
    int iTn = i*n;
    int jTnPk = k;
    for (j = 0; j < n; j++) {
        result +=
            a[iTn+j] * b[jTnPk];
        jTnPk += n;
    }
    return result;
}
```

# Dynamic Array Multiplication

```
{
  int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result += a[iTn+j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}
```

%ecx	result
%edx	j
%esi	n
%ebx	jTnPk
Mem[-4(%ebp)]	iTn

```
.L44:          # loop
  movl -4(%ebp),%eax    # iTn
  movl 8(%ebp),%edi     # a
  addl %edx,%eax       # iTn+j
  movl (%edi,%eax,4),%eax # a[..]
  movl 12(%ebp),%edi   # b
  incl %edx            # j++
  imull (%edi,%ebx,4),%eax # b[..]*a[..]
  addl %eax,%ecx       # result += ..
  addl %esi,%ebx       # jTnPk += j
  cmpl %esi,%edx       # j : n
  jl .L44              # if < goto loop
```

**Inner  
Loop**

# Summary

## Arrays in C

- **Contiguous allocation of memory**
- **Pointer to first element**
- **No bounds checking**

## Compiler Optimizations

- **Compiler often turns array code into pointer code**  
`zd2int`
- **Uses addressing modes to scale array indices**
- **Lots of tricks to improve array indexing in loops**
  - code motion
  - reduction in strength