Machine-Level Programming IV: Data

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional
    - Flat
    - Multi-level

- **Why?**
  - Allows you to write correct C code (array access calculation)
  - Performance implications
  - Opens the door to advanced applications
    - Efficient storage of arrays/data
    - Useful for data exchange (python <-> C)

Today

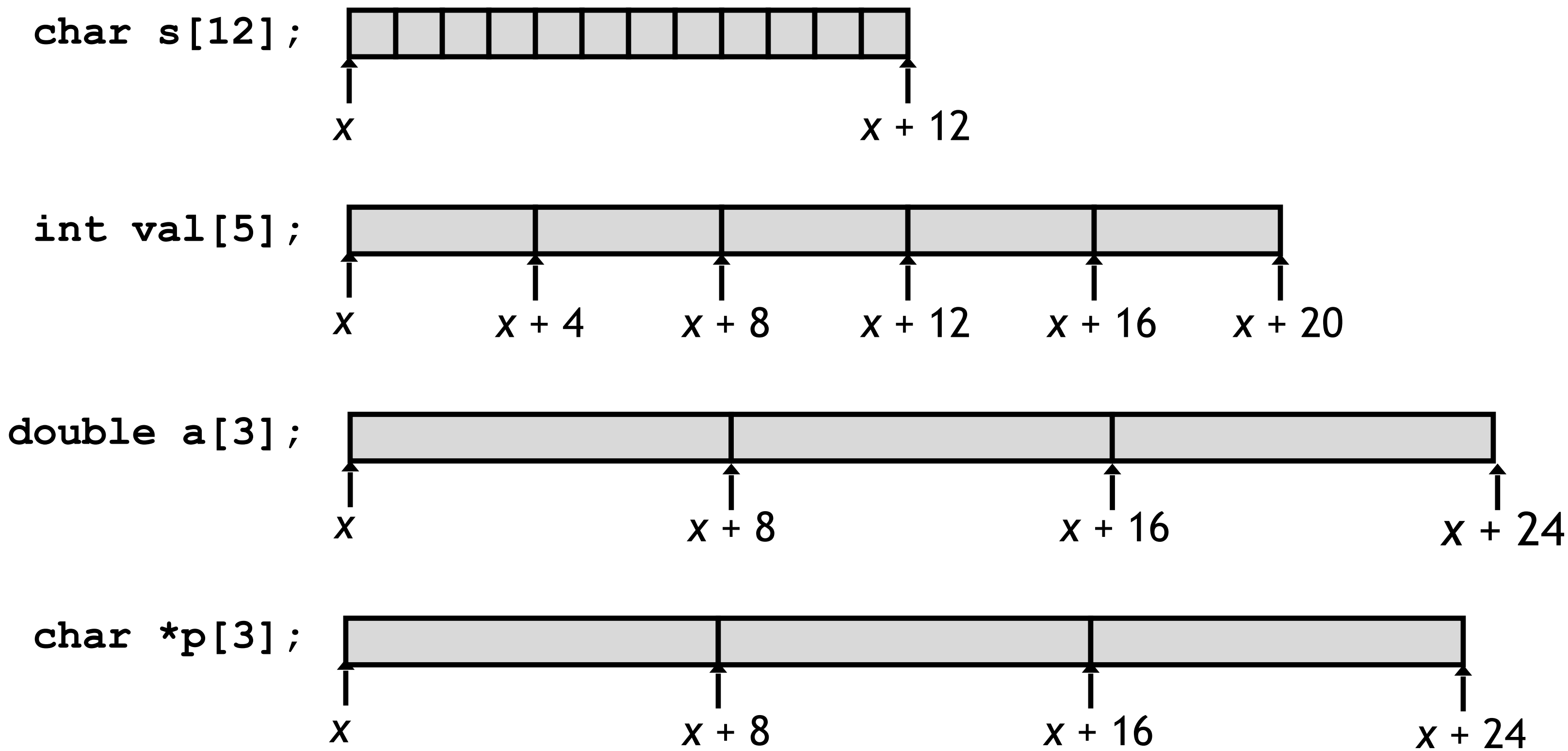
- **Arrays**
  - One-dimensional
  - Multi-dimensional
    - Flat
    - Multi-level

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



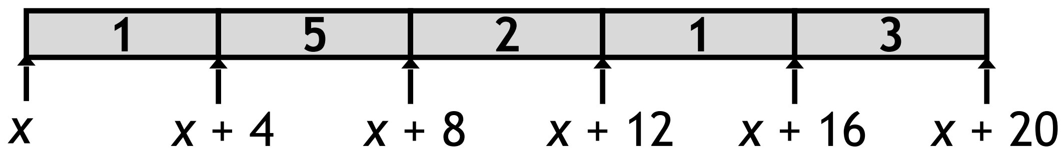
Array Access

Basic Principle

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

- Identifier **A** can be used as a pointer to array element 0

`int val[5];`

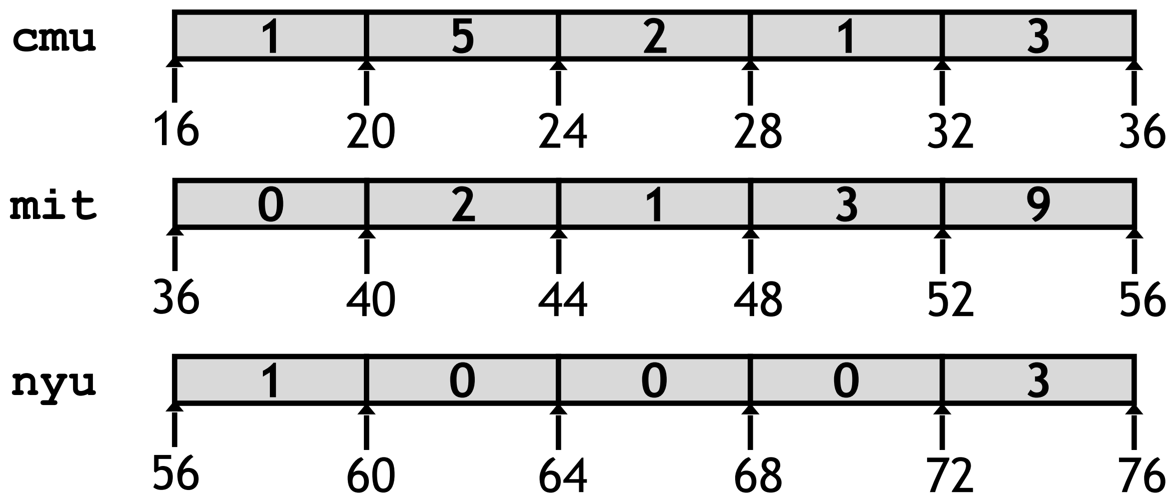


Reference	Type	Value
`val[4]`	`int`	3

Pointer arithmetic

Array Example

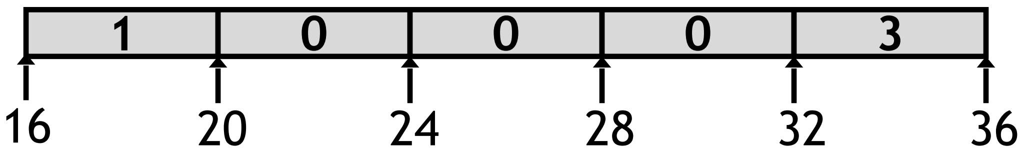
```
#define ZLEN 5

int cmu[ZLEN] = { 1, 5, 2, 1, 3 };
int mit[ZLEN] = { 0, 2, 1, 3, 9 };
int nyu[ZLEN] = { 1, 0, 0, 0, 3 };
```



- **Example arrays were allocated in successive 20 byte blocks**
  - Not guaranteed to happen in general

Array Accessing Example

```
int nyu[ZLEN];
```




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

IA32

```
# %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)`

Array Loop Example

```
void zincr(int z[ZLEN]) {
  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
```

Today

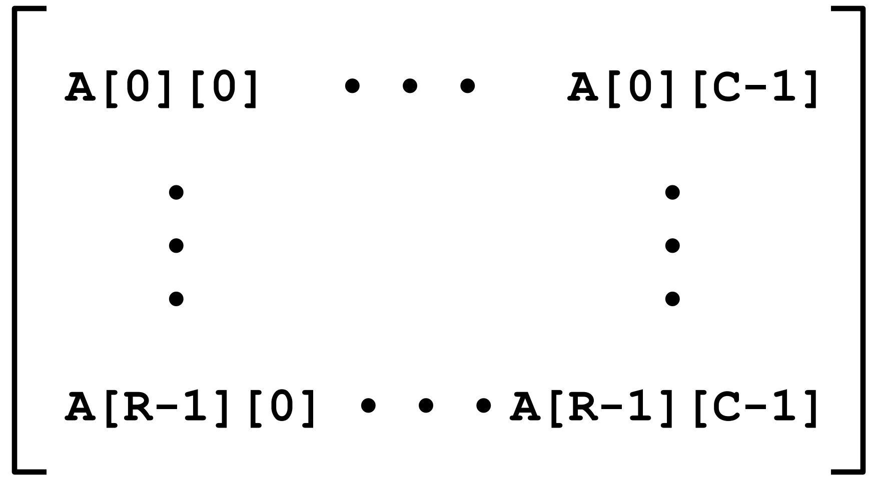
- **Arrays**
  - One-dimensional
  - Multi-dimensional
    - Flat
    - Multi-level

Multi-dimensional Arrays

- **Declaration**

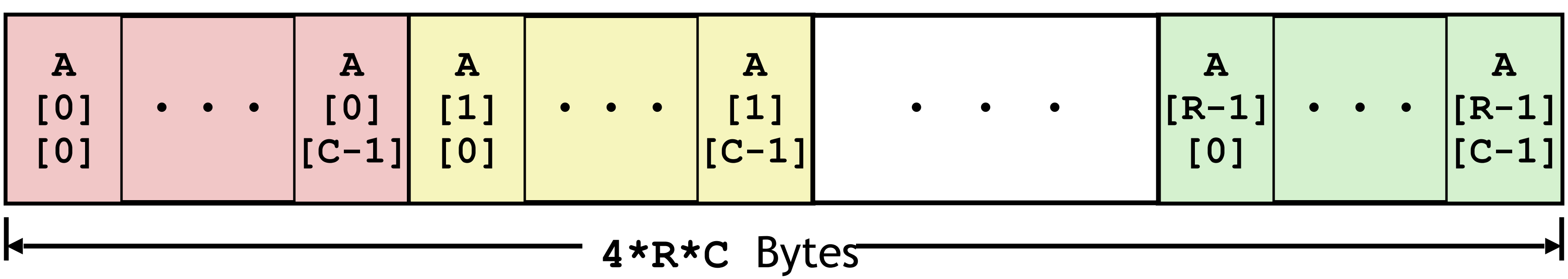
  T **A**[*R*][*C*];

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



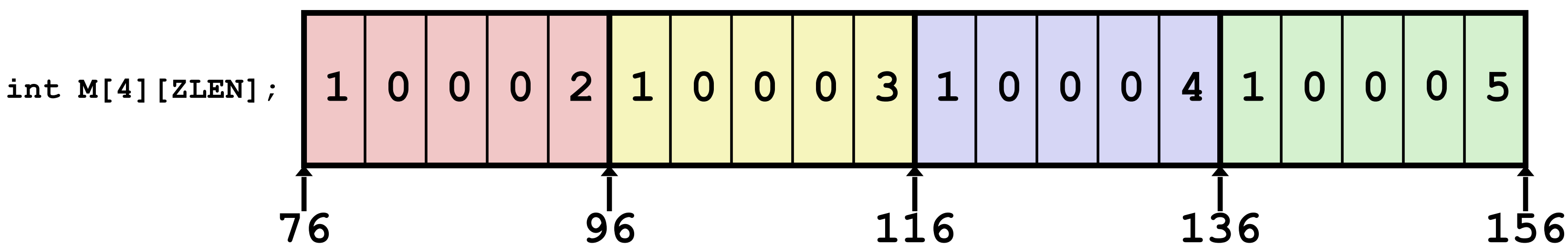
- **Array Size**
  - *R* * *C* * *K* bytes
- **Arrangement**
  - Row-Major Ordering

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



Multi-dimensional Array Example

```
int M[4][ZLEN] =
  {{1, 0, 0, 0, 2},
   {1, 0, 0, 0, 3 },
   {1, 0, 0, 0, 4 },
   {1, 0, 0, 0, 5 }};
```

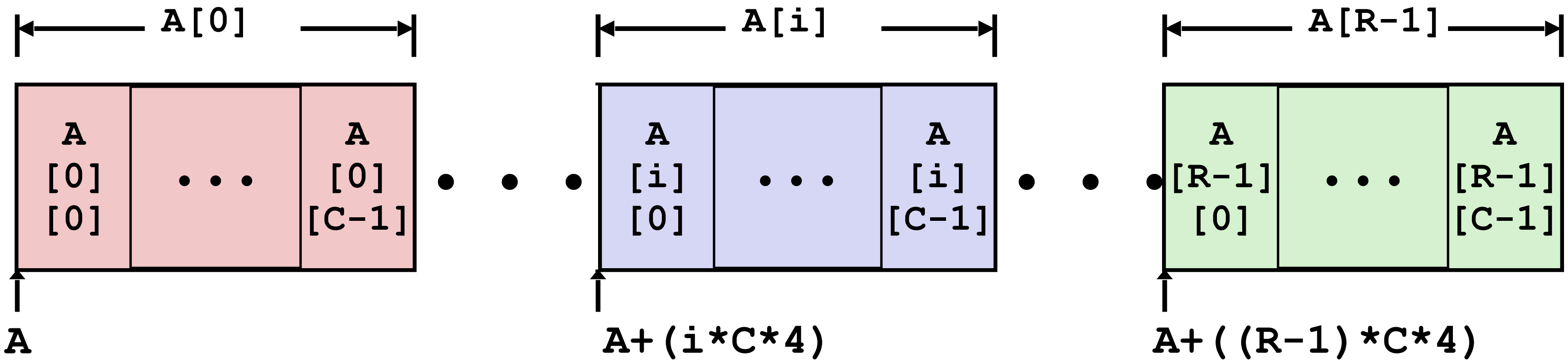


- **`int M[4][ZLEN]`**
  - Variable **`M`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- **"Row-Major" ordering of all elements in memory**

Multi-dimensional 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];
```



Row Access Code

1	0	0	0	2	1	0	0	0	3	1	0	0	0	4	1	0	0	0	5

M

```
int *get_zip(int index)
{
  return M[index];
}
```

Assembly code?

Address: $M + index * C * K = M + index * 5 * 4$

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

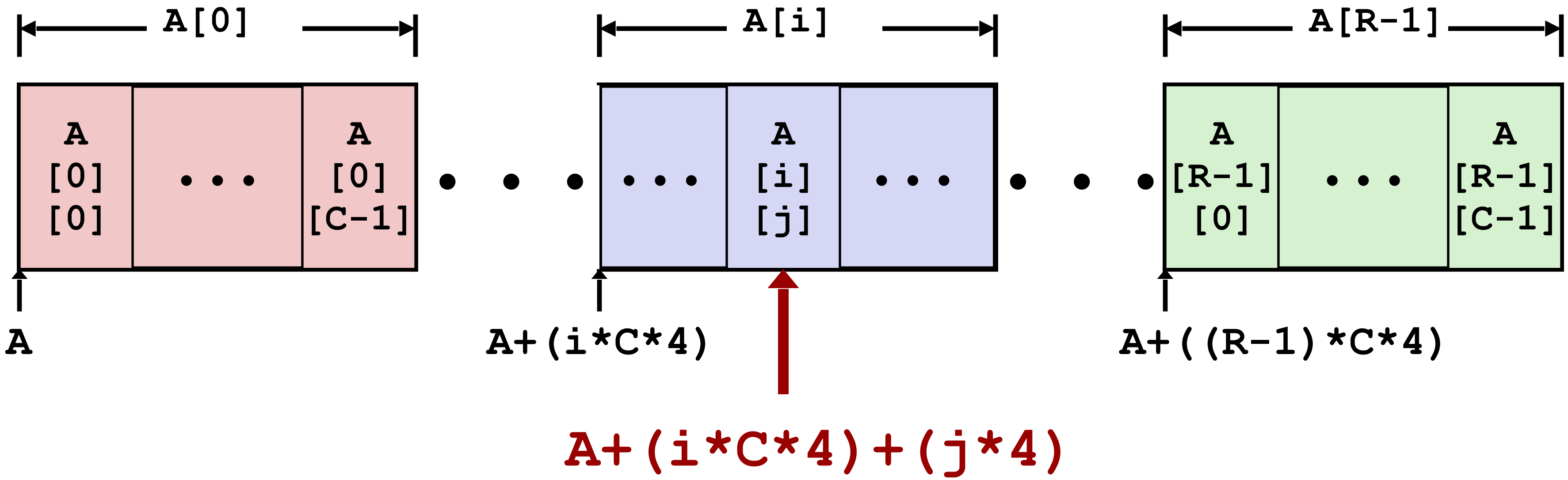
Row Vector

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

Multi-dimensional 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];
```



Element Access Code

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

M

```
int get_digit(int index, int dig)
{
  return M[index][dig];
}
```

Assembly code?

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   M(,%rsi,4), %eax       # M + 4*(5*index+dig)
```

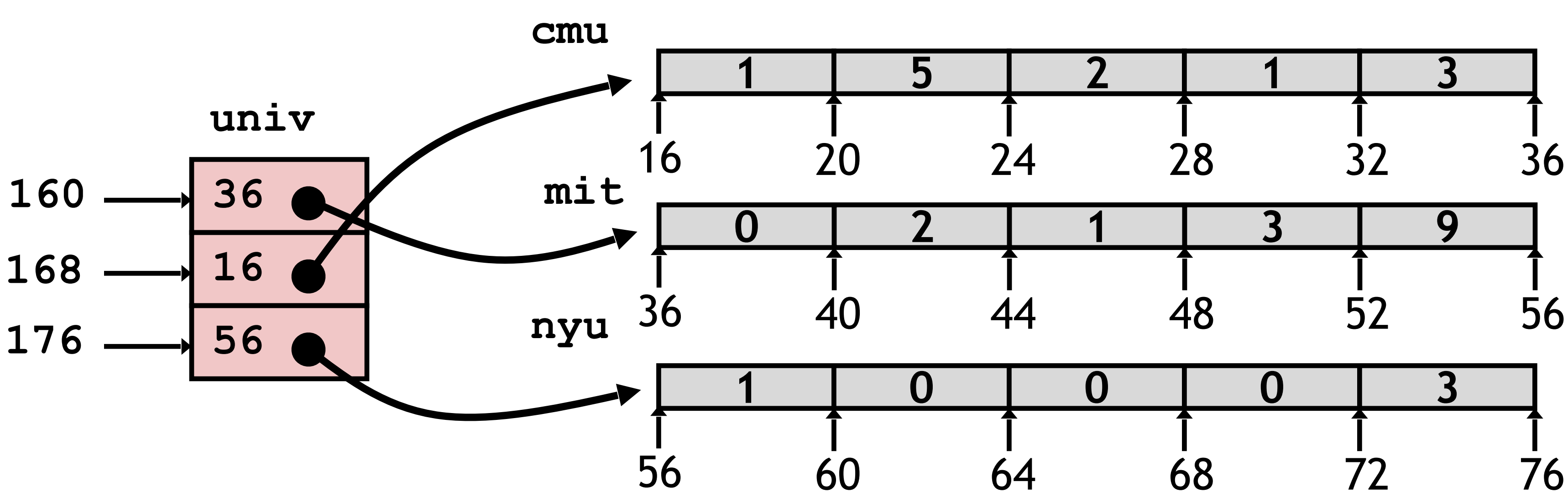
- **Array Elements**
  - `M[index][dig]` is `int`
  - Address: `M + 20*index + 4*dig`
    - = `M + 4*(5*index + dig)`
    - `M[index][dig] = M + K*(C*index + dig)`
      `where K is 4 and C is 5`

Multi-Level Array Example

```
int cmu[ZLEN] = { 1, 5, 2, 1, 3 };
int mit[ZLEN] = { 0, 2, 1, 3, 9 };
int nyu[ZLEN] = { 1, 0, 0, 0, 3 };
```

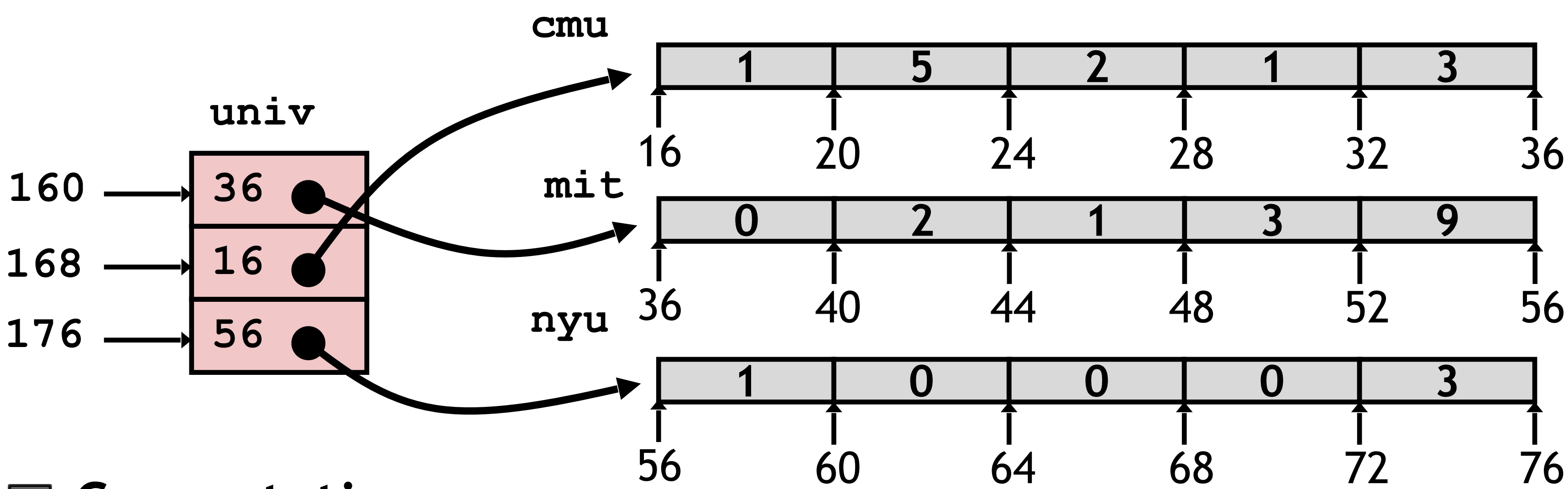
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, nyu};
```

- 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_digit(size_t index, size_t digit)
{
  return univ[index][digit];
}
```



Computation

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

Element Access in Multi-Level Array

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

- **Computation**
  - Element access **`Mem[Mem[univ+8*index]+4*digit]`**

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

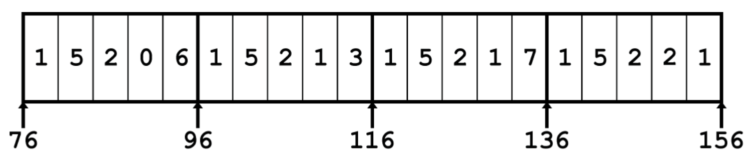
Array Element Accesses

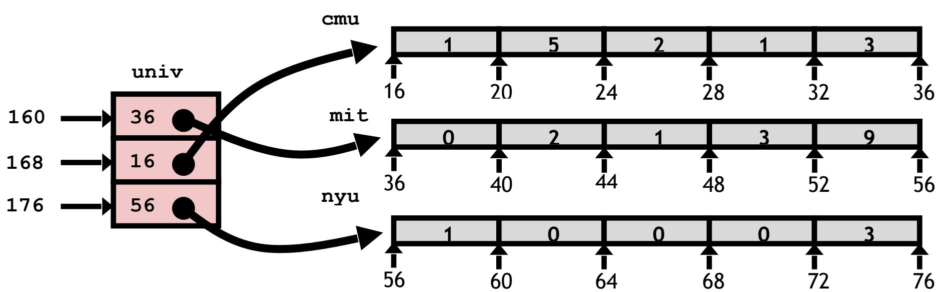
Flat

```
int get_digit
  (size_t index, size_t digit)
{
  return M[index][digit];
}
```

Multi-level array

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





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

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

Summary

- **Arrays**
  - We have seen
    - One-dimensional arrays
    - Multi-dimensional arrays
  - For each one, we have seen
    - How to declare it in C?
    - How is it allocated in memory?
      - Contiguous region of memory
      - Array of arrays (multi-level arrays)
    - How to calculate the address of an individual element?
      - Use index arithmetic to locate individual elements
      - We have seen how to do this in C and in Assembly