# Machine-Level Programming: Arrays and Structures

CMPU 224 – Computer Organization
Jason Waterman

# Arrays in C

- Declaring arrays
  - `type array_name [ array_size ];`
    - Example: `int count [5];`
  - All elements of the array have the same type
- Declaring and initializing
  - `int count[] = {4, 2, 9, 3, 5};`

First Element ↓

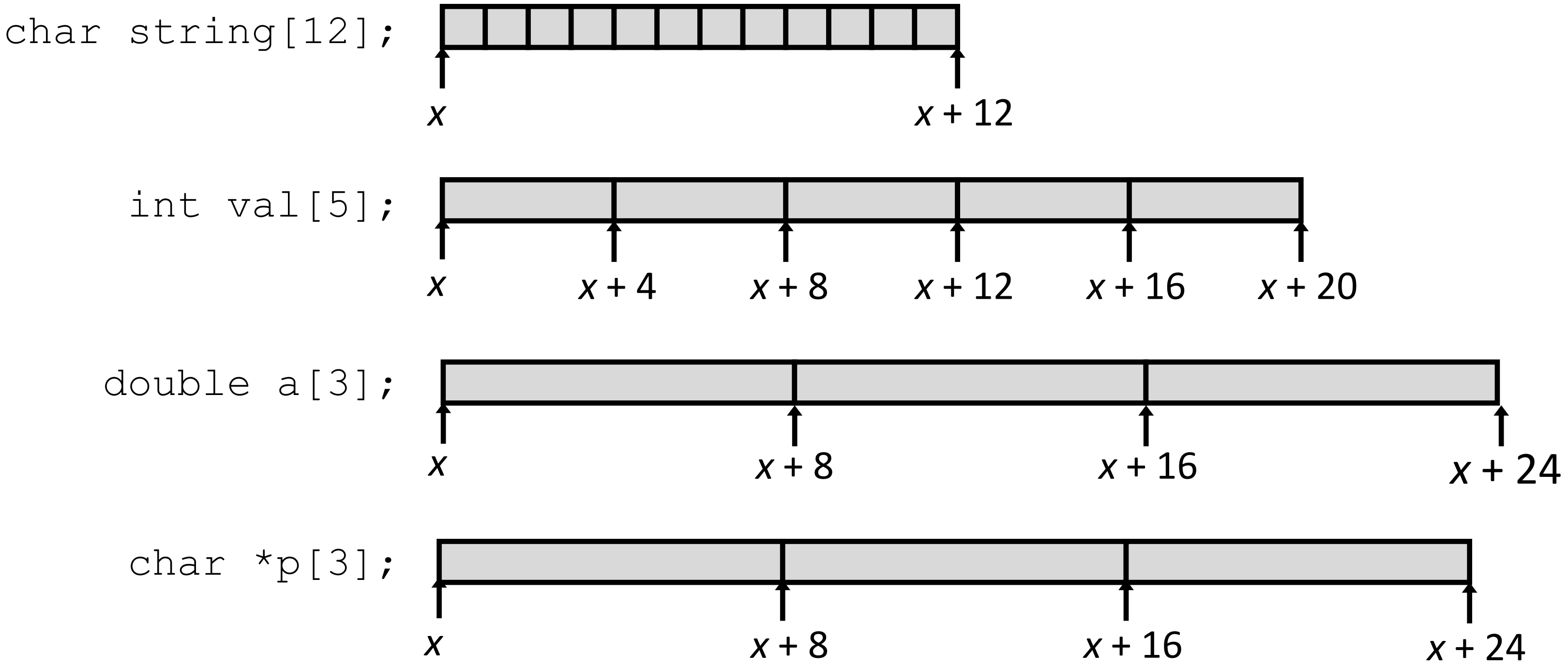
| count[0] | count[1] | count[2] | count[3] | count[4] |
|---|---|---|---|---|
| 4 | 2 | 9 | 3 | 5 |

# 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

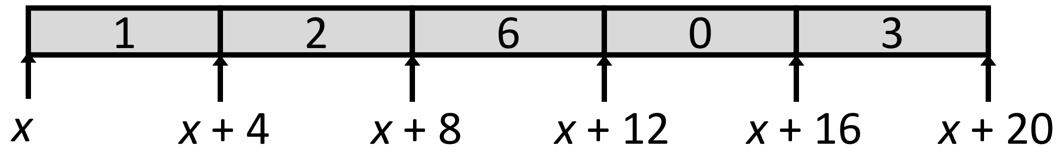


# Array Access

- Basic Principle

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

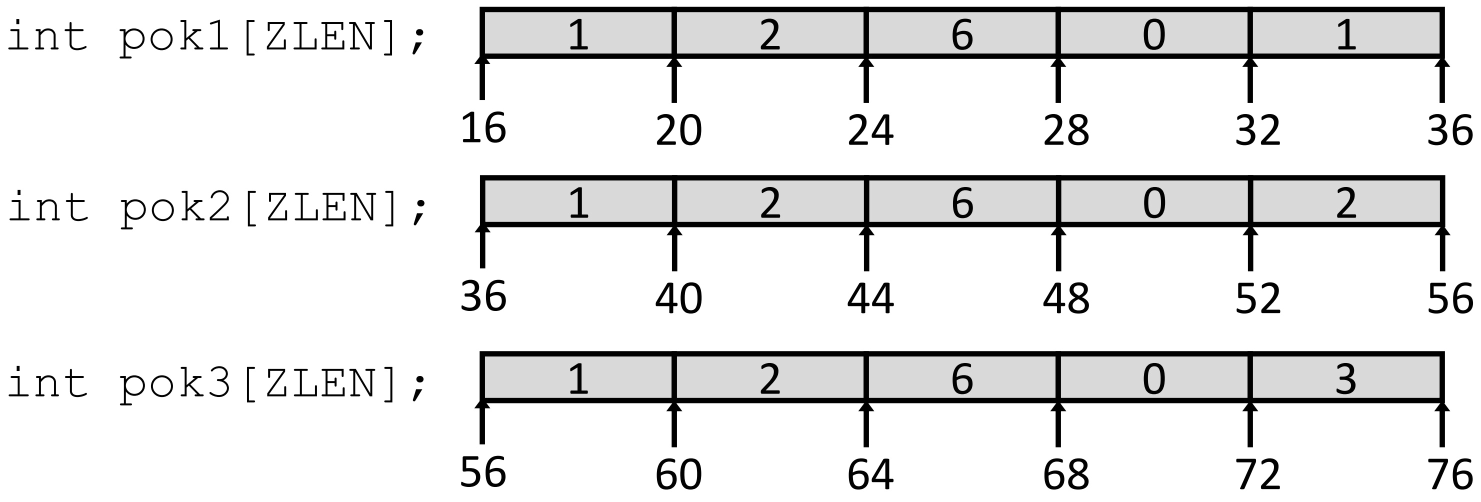
| Reference | Type | Value |
|---|---|---|
| **val** | **int *** | *x* |
| **val[4]** | **int** | 3 |
| **val+1** | **int *** | *x* + 4 |
| **&val[2]** | **int *** | *x* + 8 |
| **val[5]** | **int** | ?? |
| ***(val+1)** | **int** | 2 |
| **val + *i*** | **int *** | *x* + 4 *i* |

# Array Example

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

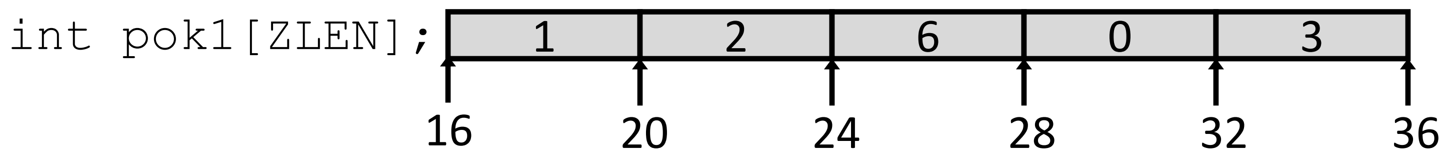
```
#define ZLEN 5

int pok1[ZLEN]  =   { 1, 2, 6, 0, 1 };
int pok2[ZLEN]  =   { 1, 2, 6, 0, 2 };
int pok3[ZLEN]  =   { 1, 2, 6, 0, 3 };
```



# Array Accessing Example

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



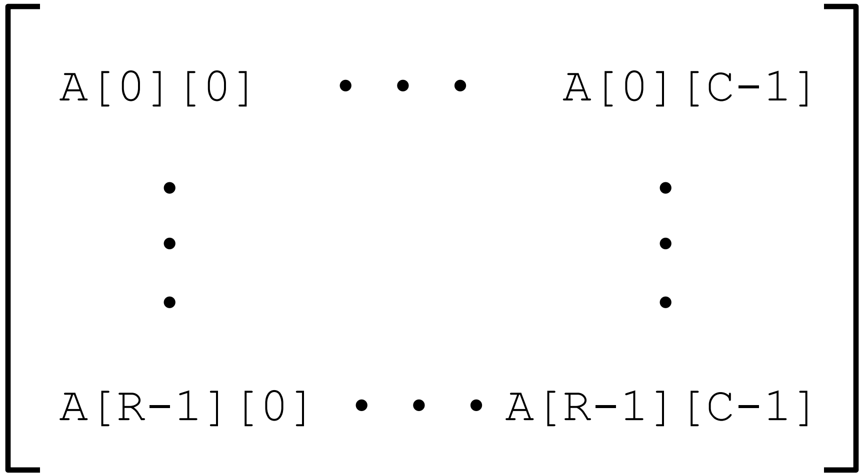

```
int get_digit(int z[], int digit){
  return z[digit];
}
```

x86-64

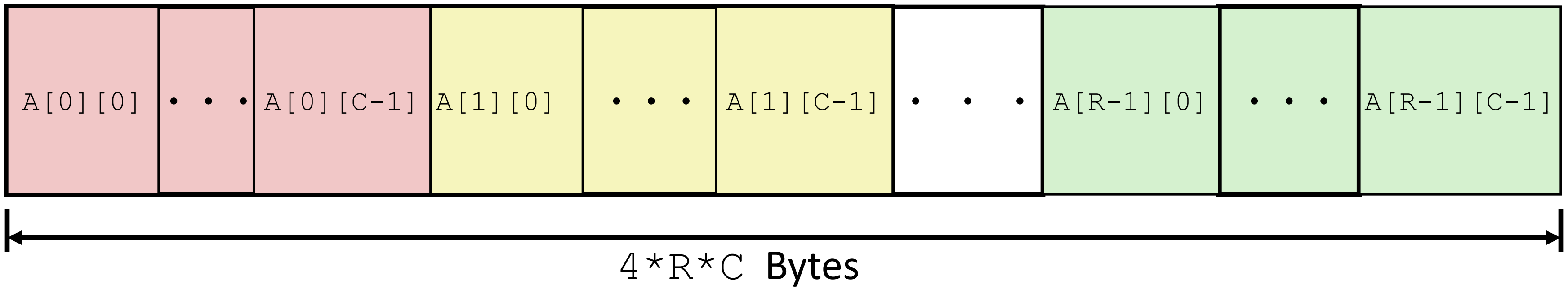
```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax   # z[digit]
```

# Multidimensional (Nested) Arrays

- Declaration
  *T* **A**[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- Arrangement
  - **Row-Major Ordering**
- Array Size
  - *R* * *C* * *K* bytes



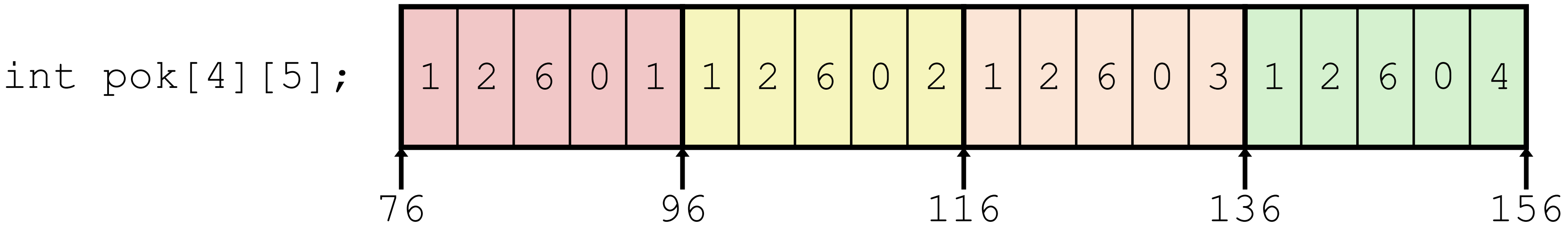

```
int A[R][C];
```



# Nested Array Example

- `int pok[4][5];`
  - Variable **`pok`**: 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

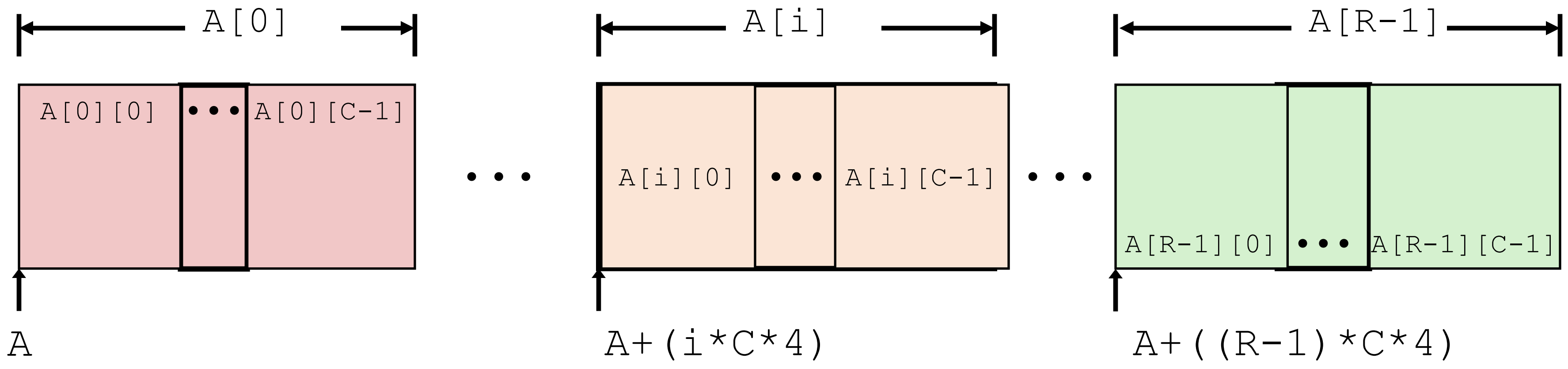
```
int pok[4][5] =
  {{1, 2, 6, 0, 1},
   {1, 2, 6, 0, 2},
   {1, 2, 6, 0, 3},
   {1, 2, 6, 0, 4}};
```



# Nested Array Row Access

- Row Vectors
  - `int A[R][C];`
  - **`A[i]`** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address **`A +`** $i * (C * K)$

SIZE OF ONE ROW

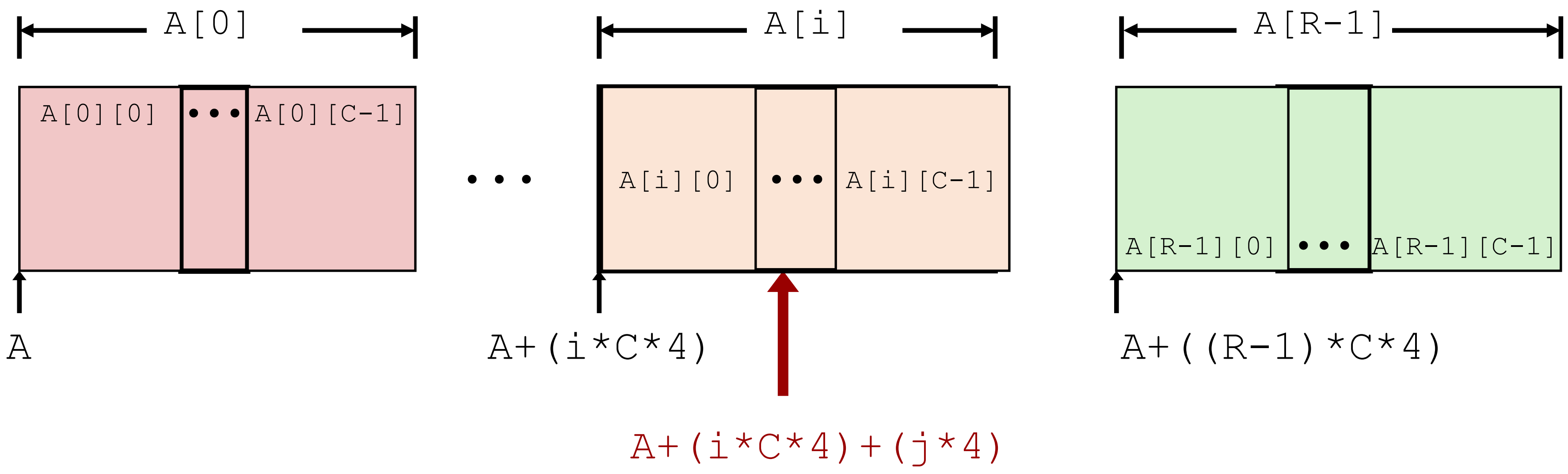


# Nested Array Element Access

- Array Elements
  - **A[i][j]** is element of type *T*, which requires *K* bytes and has *R* rows and *C* cols
  - Address **A** + $i * (C * K) + j * K = \boldsymbol{A + (i * C + j) * K}$

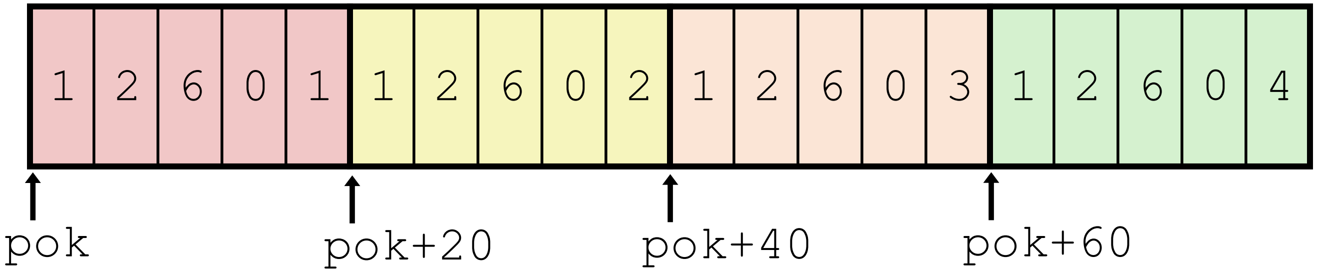
SIZE OF ROW

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



# Nested Array Row Access Code

- Row Vector
  - **pok[index]** is array of 5 **int**'s
  - Row vector: starting address **pok + 20*index**
- Machine Code
  - Computes and returns address
  - Compute as **pok + 4*(index + 4*index)**

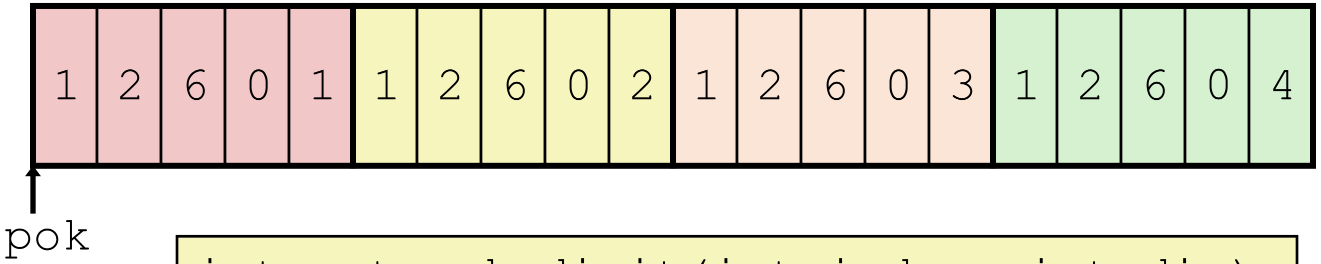



```
int *get_pok_zip(int index){
  return pok[index];
}
```

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

# Nested Array Element Access Code

- Array Elements
  - **`pok[index][dig]`** is type **`int`**
  - Address: **`pok + 20*index + 4*dig = pok + 4*(5*index + dig)`**



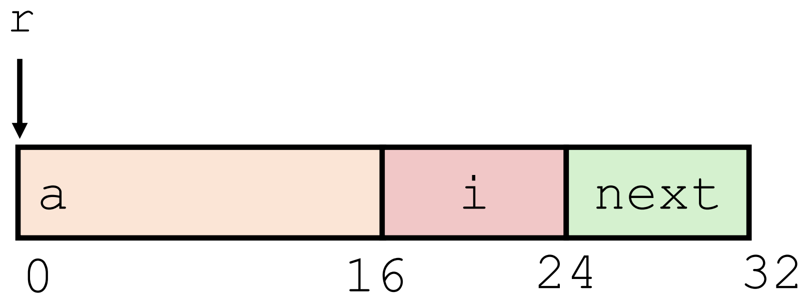

```
int get_pok_digit(int index, int dig)
{
  return pok[index][dig];
}
```

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

# Structure Representation

- 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**

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

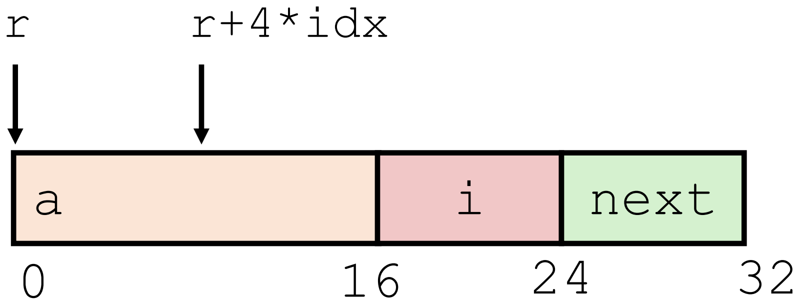


# Generating Pointer to Structure Member

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Accessing an element in array a: compute as **`r + 4*idx`**

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

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

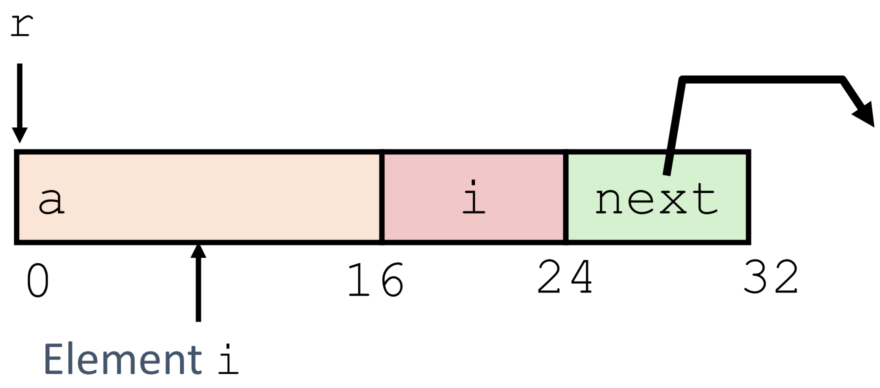



```
# r in %rdi, idx in %rsi
movl   (%rdi,%rsi,4), %eax
ret
```

# Next Linked List

- Return address of next node in the linked list

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



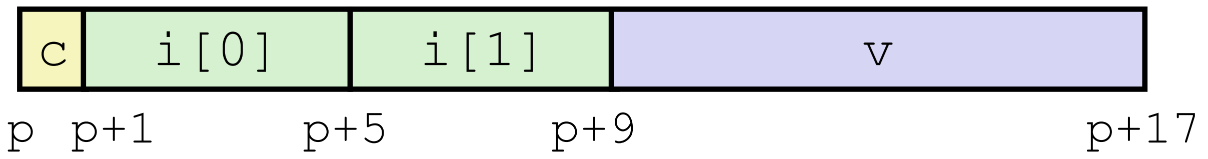

```
struct rec* get_next(struct rec *r) {
    return r->next;
}
```

| Register | Value |
|---|---|
| %rdi | r |

```
movq 24(%rdi), %rax
ret
```

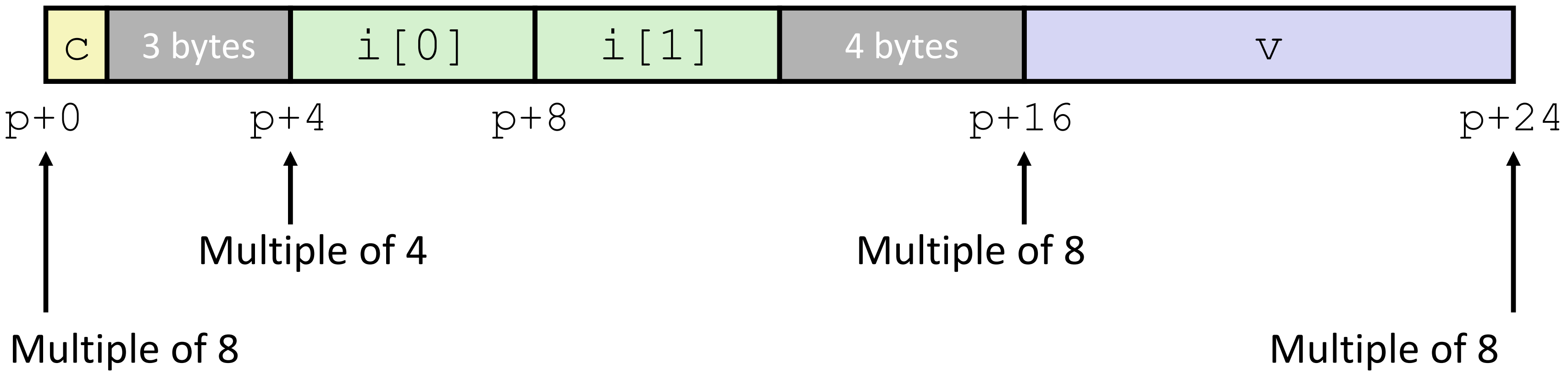
# Structures & Alignment

- Unaligned Data




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

- Aligned Data
  - A primitive data type of K bytes must have an address that is 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

- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store data that spans quad word boundaries

- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

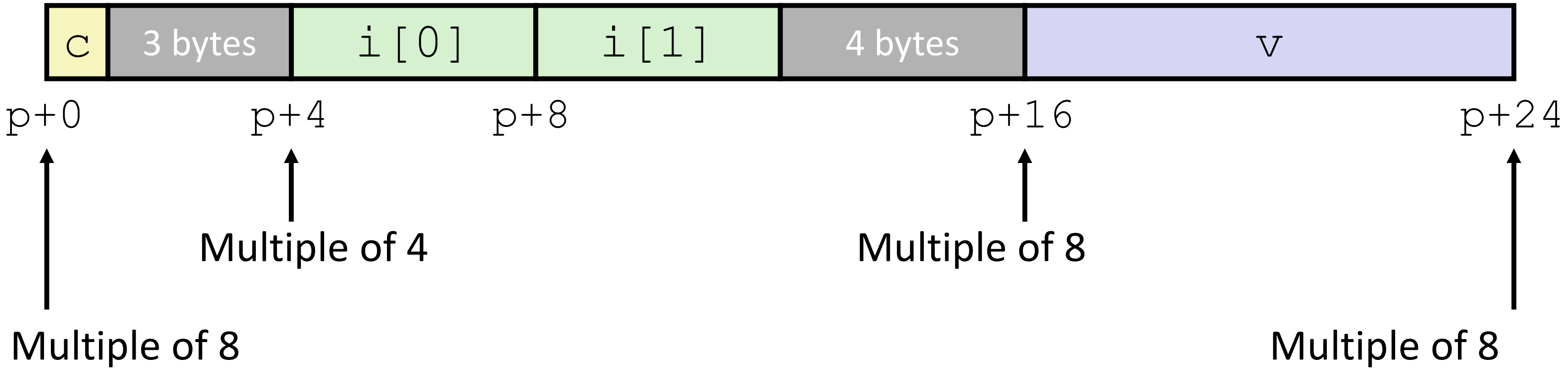
# Specific Cases of Alignment (x86-64)

- 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 $000_2$

# Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement K
    - $K_{struct}$ = Largest alignment of any element in struct
  - Initial address & structure length must be multiples of $K_{struct}$
- Example:
  - $K_{struct}$ = 8, due to **double** element

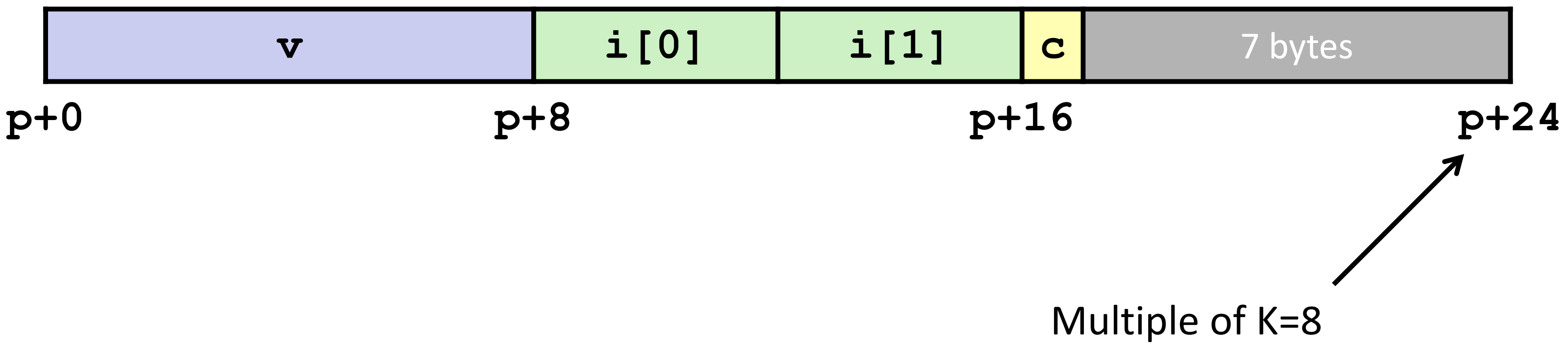
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



# Meeting Overall Alignment Requirement

- Largest alignment requirement $K_{struct}$
- Overall structure must be multiple of $K_{stuct}$

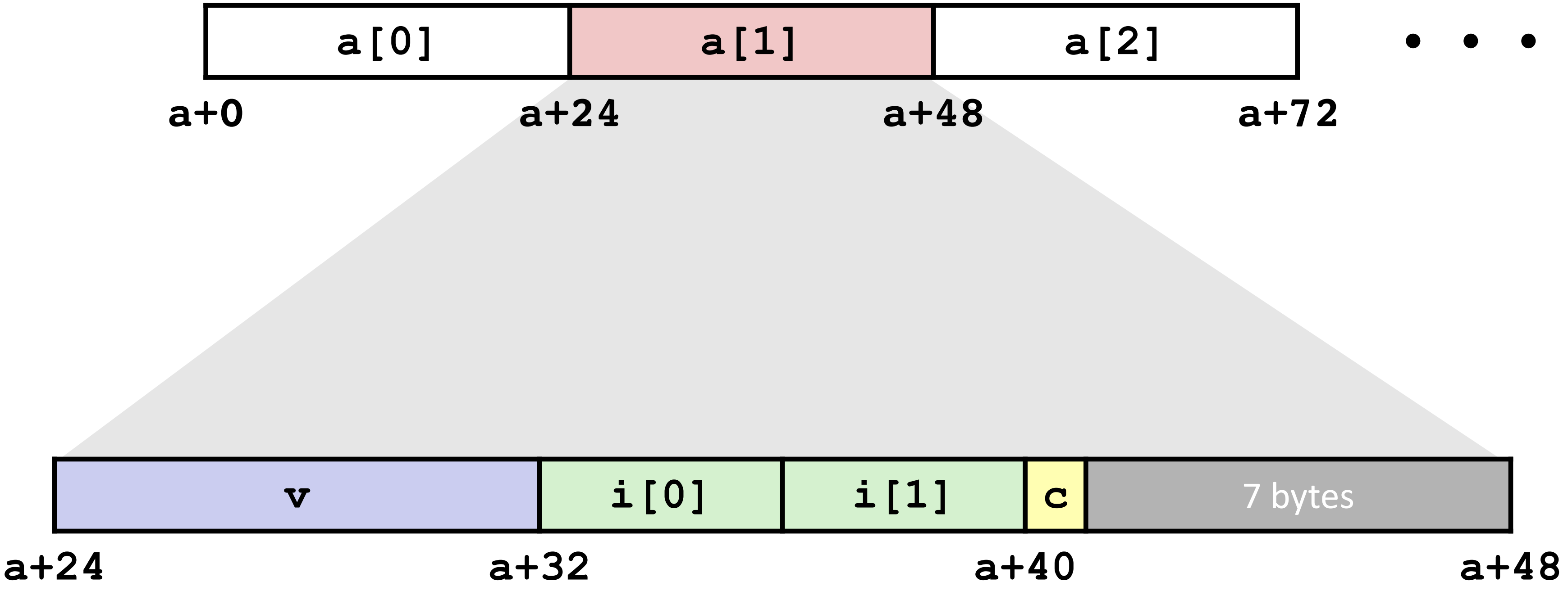
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



# Arrays of Structures

- Overall structure length multiple of $K_{struct}$
- Satisfy alignment requirement for every element

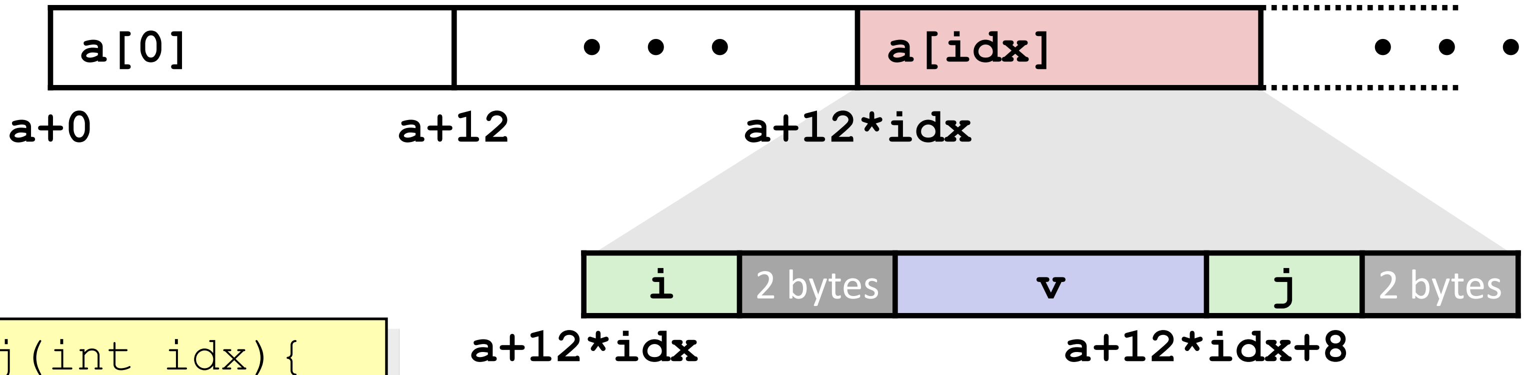
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



# Accessing Array Elements

- Compute array offset as `sizeof(s3)*idx`
  - **`sizeof(S3)`** = 12, including alignment spacers
- Element **`j`** is at offset 8 within structure
- Assembler gives offset **`a+8`**

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




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

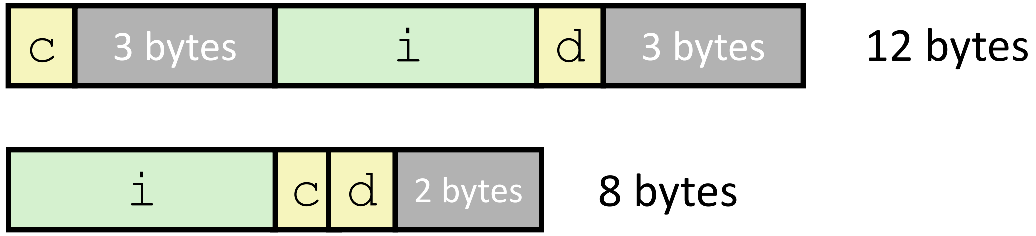
```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

# Saving Space

- Put large data types first

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

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



# 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