Section 5: Arrays & Other Data Structures

- Array allocation and access in memory
- Multi-dimensional or nested arrays
- Multi-level arrays
- Other structures in memory
- Data structures and alignment

Nested Array Example

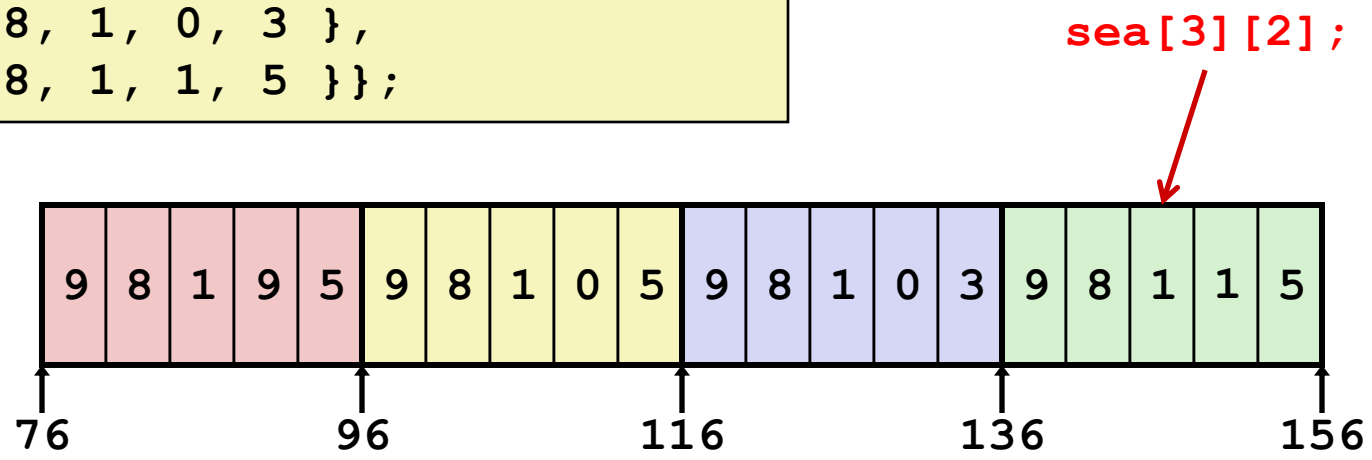
```
#define PCOUNT 4
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

Remember, **`T A[N]`** is an array with **`N`** elements of type **`T`**

Nested Array Example

```
#define PCOUNT 4
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

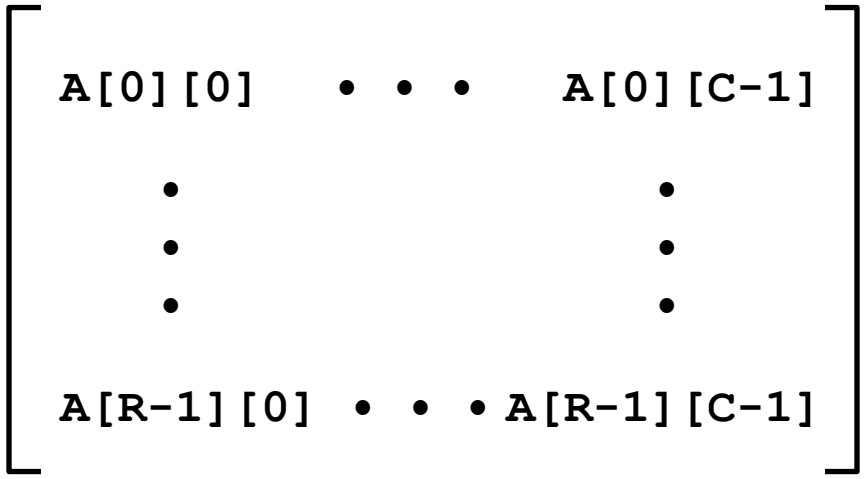
Remember, `T A[N]` is an array with `N` elements of type `T`



- "Row-major" ordering of all elements
- This is guaranteed

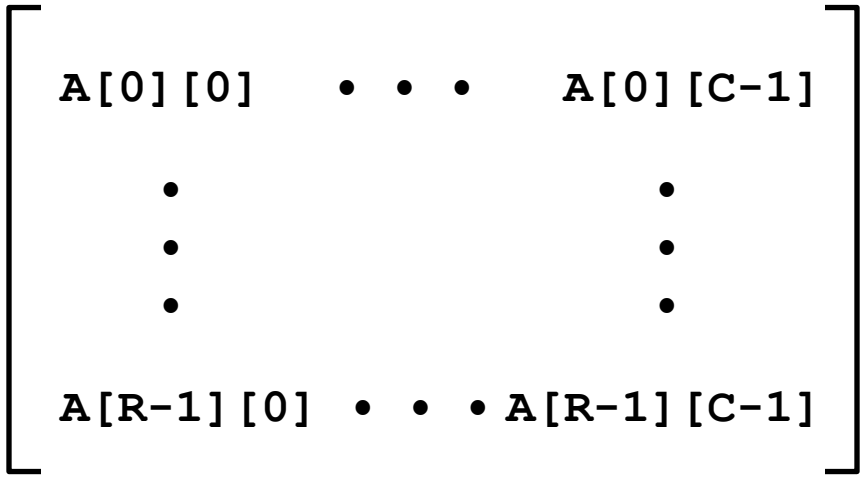
Multidimensional (Nested) Arrays

- **Declaration**
  - T A[R][C];
  - 2D array of data type T
  - R rows, C columns
  - Type T element requires K bytes
- **Array size?**

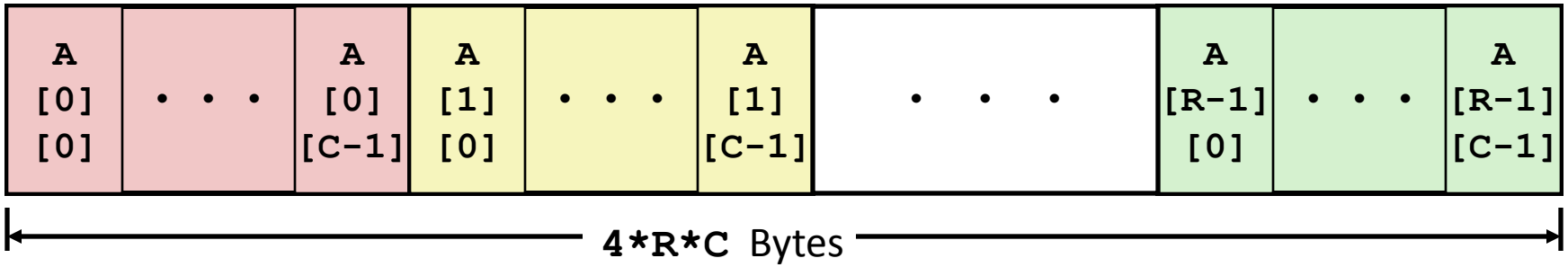


Multidimensional (Nested) Arrays

- **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



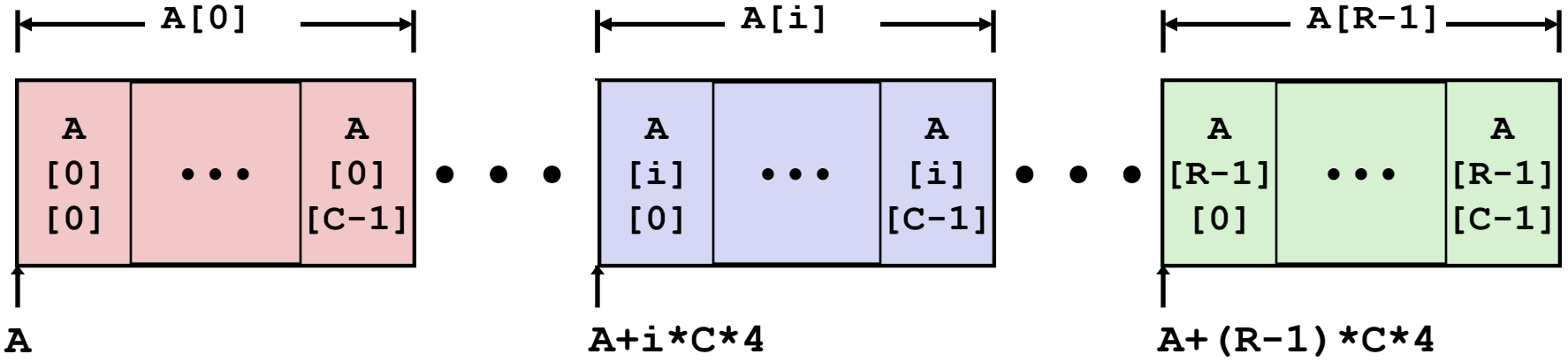
`int A[R][C];`



Nested Array Row Access

- **Row vectors**
  - T A[R][C]: 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_sea_zip(int index)
{
  return sea[index];
}
```

```
#define PCOUNT 4
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

Nested Array Row Access Code

```
int *get_sea_zip(int index)
{
  return sea[index];
}
```

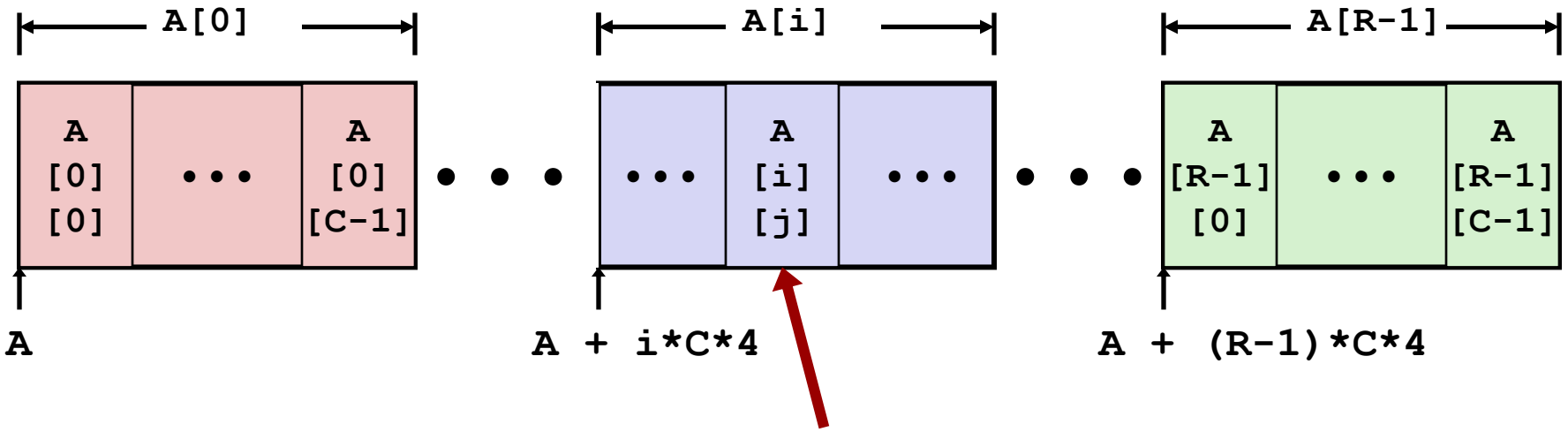
```
#define PCOUNT 4
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

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

- **Row Vector**
  - **`sea[index]`** is array of 5 **`int`**s (a **`zip_dig`** data type)
  - Starting address **`sea+20*index`**
- **IA32 Code**
  - Computes and returns address
  - Compute as **`sea+4*(index+4*index)=sea+20*index`**

Nested Array Row Access

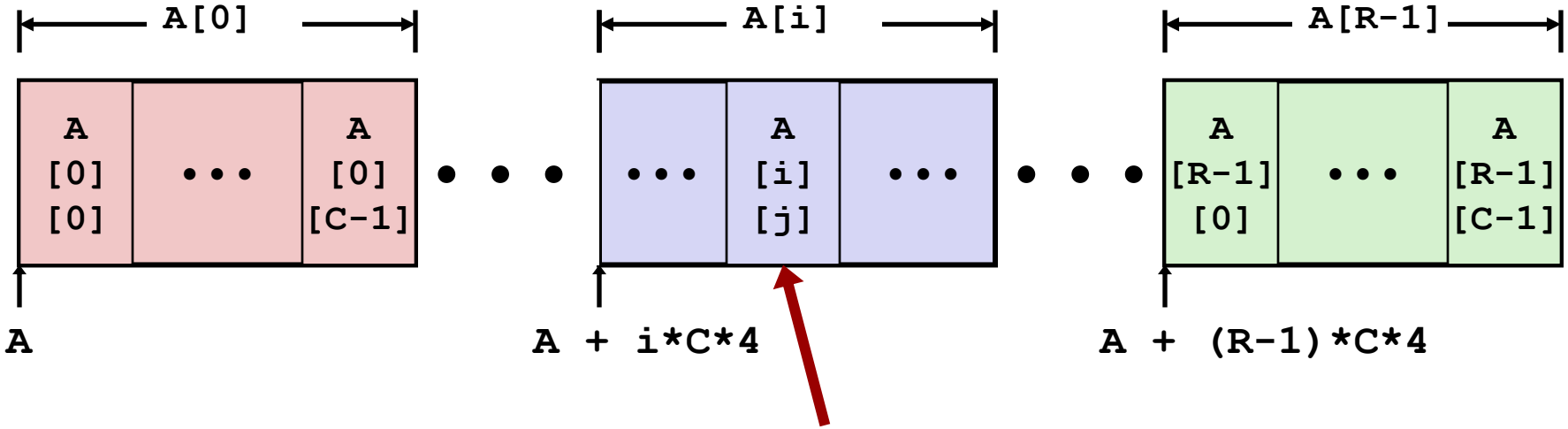
```
int A[R][C];
```



Nested Array Row 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

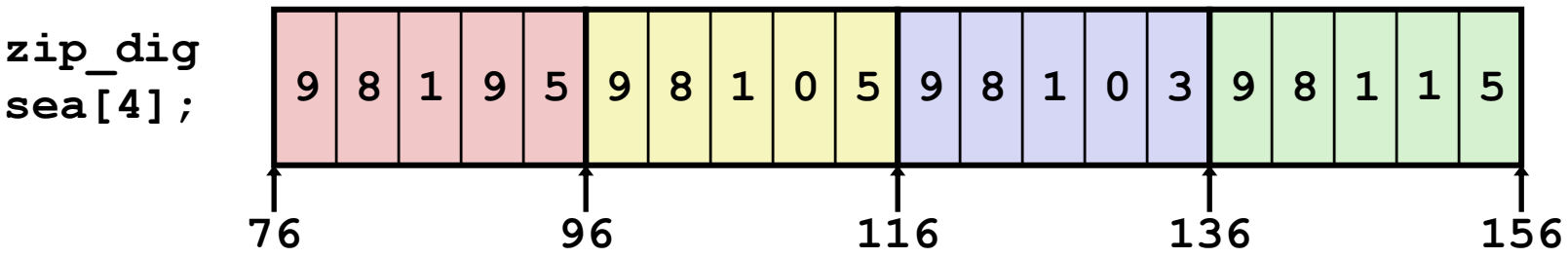
```
int get_sea_digit
  (int index, int dig)
{
  return sea[index][dig];
}
```

```
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

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

- **Array Elements**
  - sea[index][dig] is int
  - Address: sea + 20*index + 4*dig
- **IA32 Code**
  - Computes address sea + 4*dig + 4*(index+4*index)
  - movl performs memory reference

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`sea[3][3]`	`76+20*3+4*3 = 148`	`1`	Yes
`sea[2][5]`	`76+20*2+4*5 = 136`	`9`	Yes
`sea[2][-1]`	`76+20*2+4*-1 = 112`	`5`	Yes
`sea[4][-1]`	`76+20*4+4*-1 = 152`	`5`	Yes
`sea[0][19]`	`76+20*0+4*19 = 152`	`5`	Yes
`sea[0][-1]`	`76+20*0+4*-1 = 72`	`??`	No

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