Arrays and Structs

CSE 351 Summer 2018

Instructor:

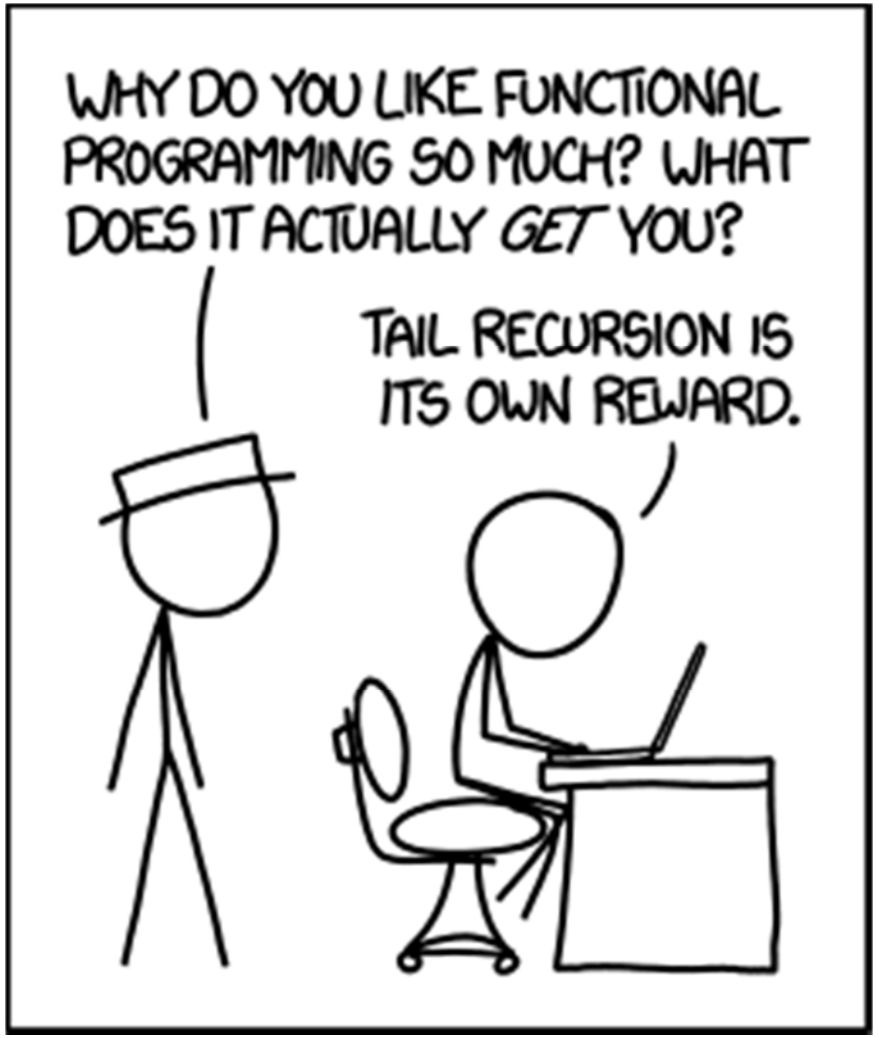
Justin Hsia

Teaching Assistants:

Josie Lee

Natalie Andreeva

Teagan Horkan



http://xkcd.com/1270/

Administrivia

- Lab 2 due tonight
- Homework 3 due next Monday (7/23)

- **Midterm** (Wednesday in lecture)
  - 60-minute exam
  - Midterm details Piazza post: @58
  - **Review session:** 5:00-6:30pm tonight in EEB 045
    - Take a look at midterm review packet

- Some lecture material covered in Section on Thursday
- Lab 3 released on Thursday (7/19)

Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - **Multi-dimensional (nested)**
  - Multi-level
- Structs
  - Alignment
- ~~Unions~~

```
typedef int zip_dig[5];
```

Nested Array Example

```
zip_dig sea[4] =
  {{ 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 elements of type **T**, with length **N**

same as:

```
int sea[4][5];
```

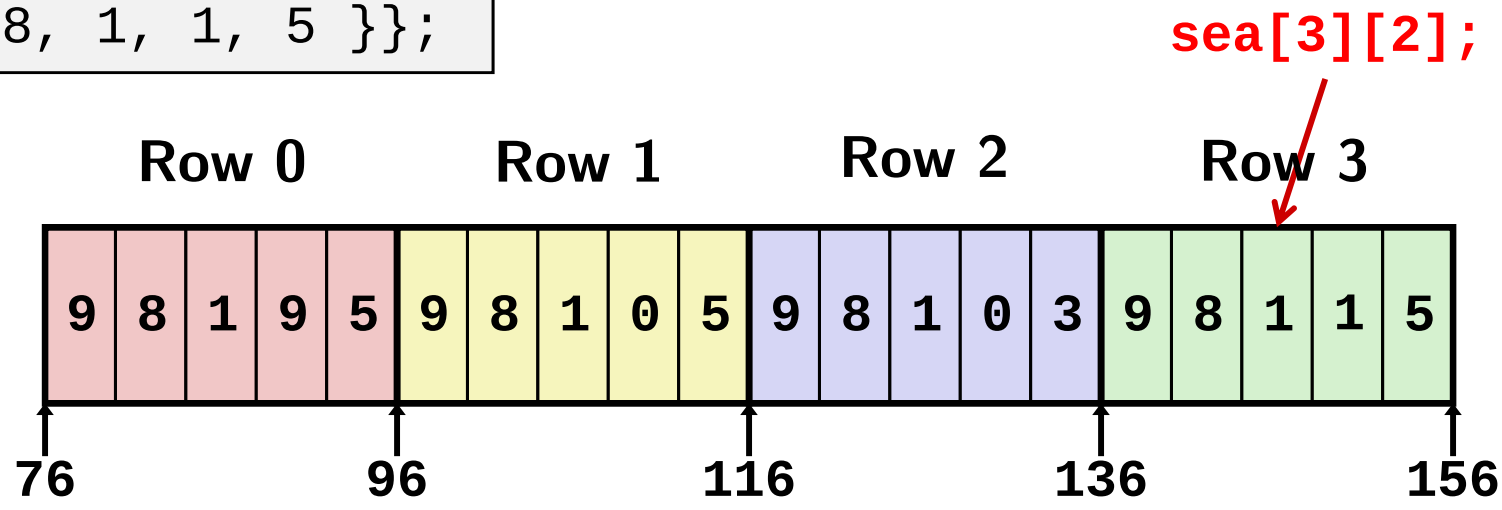
What is the layout in memory?

Nested Array Example

```
typedef int zip_dig[5];
```

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

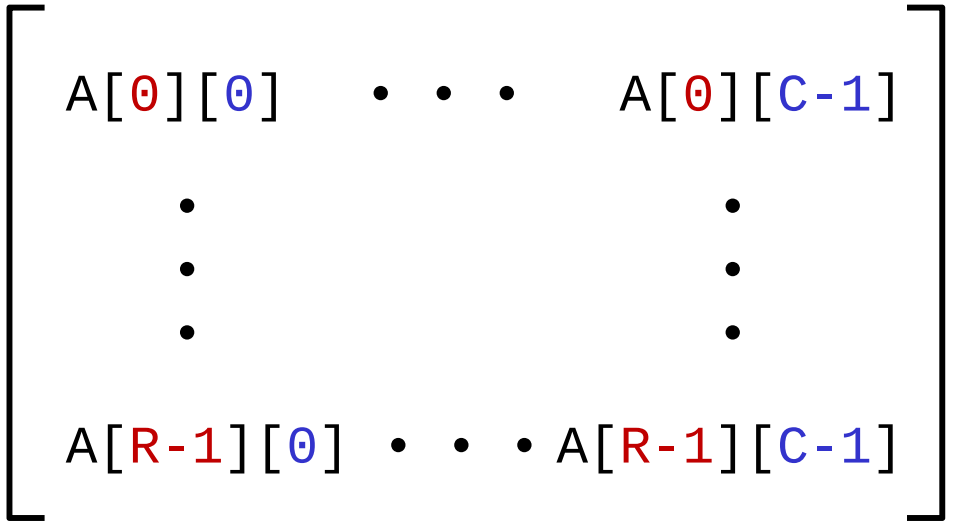
Remember, **T A[N]** is an array with elements of type **T**, with length **N**



- "Row-major" ordering of all elements
- Elements in the same row are contiguous
- Guaranteed (in C)

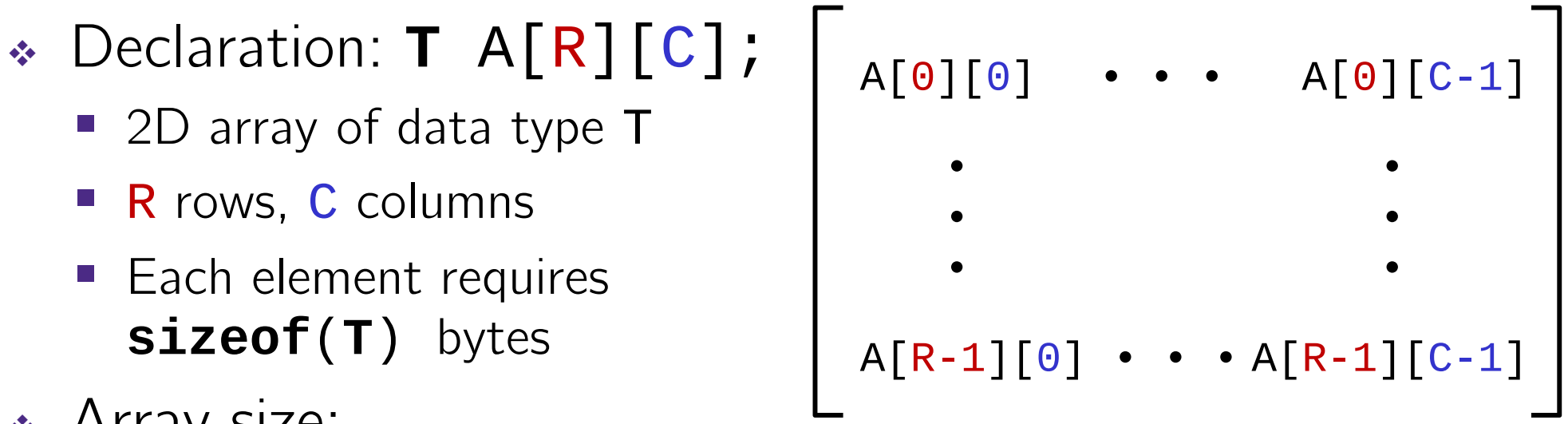
Two-Dimensional (Nested) Arrays

- Declaration: `T A[R][C];`
  - 2D array of data type `T`
  - `R` rows, `C` columns
  - Each element requires **`sizeof(T)`** bytes
- Array size?

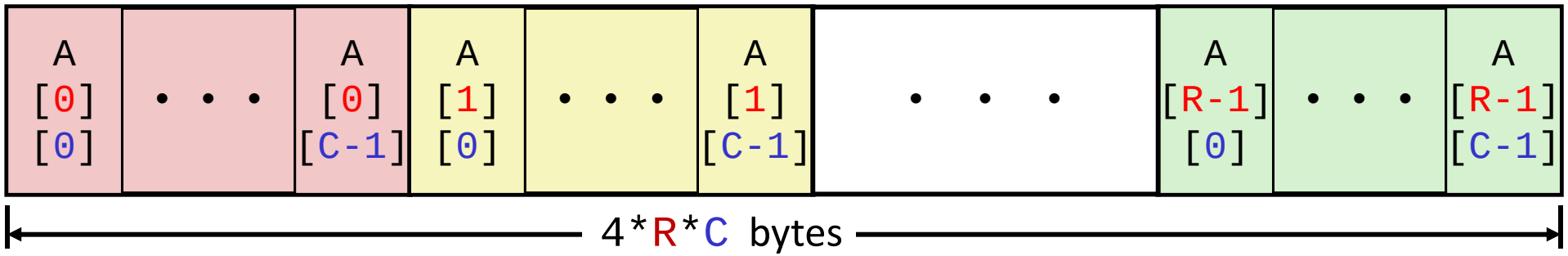


Two-Dimensional (Nested) Arrays

- Declaration: **`T A[R][C];`**
  - 2D array of data type T
  - R rows, C columns
  - Each element requires **`sizeof(T)`** bytes
- Array size:
  - **`R*C*sizeof(T)`** bytes
- Arrangement: **row-major** ordering

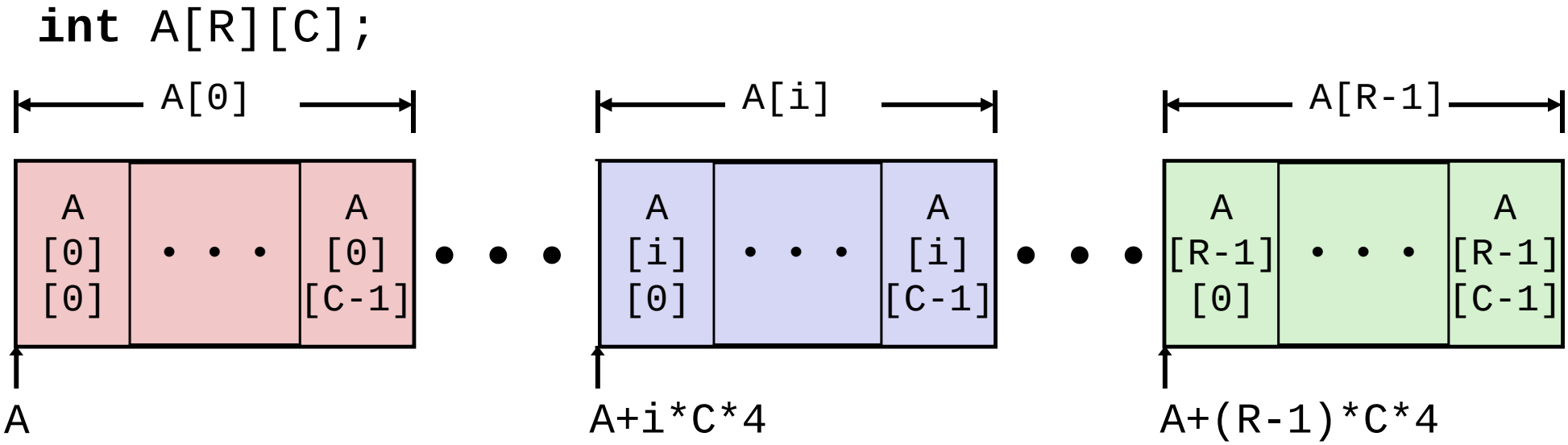


`int A[R][C];`



Nested Array Row Access

- Row vectors
  - Given `T A[R][C]`,
    - `A[i]` is an array of `C` elements ("row `i`")
    - `A` is address of array
    - Starting address of row `i = A + i*(C * sizeof(T))`



Nested Array Row Access Code

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

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
get_sea_zip(int):
    movslq  %edi, %rdi
    leaq    (%rdi,%rdi,4), %rax
    leaq    sea(,%rax,4), %rax
    ret

sea:
    .long   9
    .long   8
    .long   1
    .long   9
    .long   5
    .long   9
    .long   8
...
```

Nested Array Row Access Code

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

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

- What data type is `sea[index]`?
- What is its value?

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax
leaq sea(,%rax,4),%rax
```

Translation?

Nested Array Row Access Code

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

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

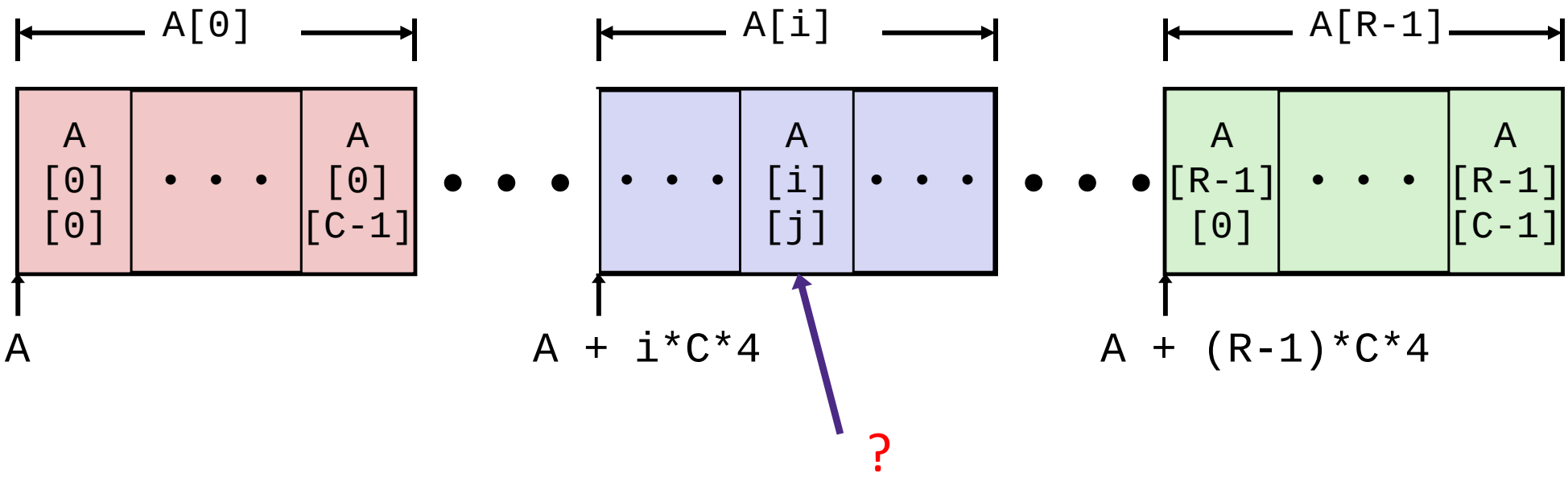
```
  # %rdi = index
  leaq (%rdi,%rdi,4),%rax  # 5 * index
  leaq sea(,%rax,4),%rax   # sea + (20 * index)
```

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

Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type `T`, which requires *K* bytes
  - Address of `A[i][j]` is

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

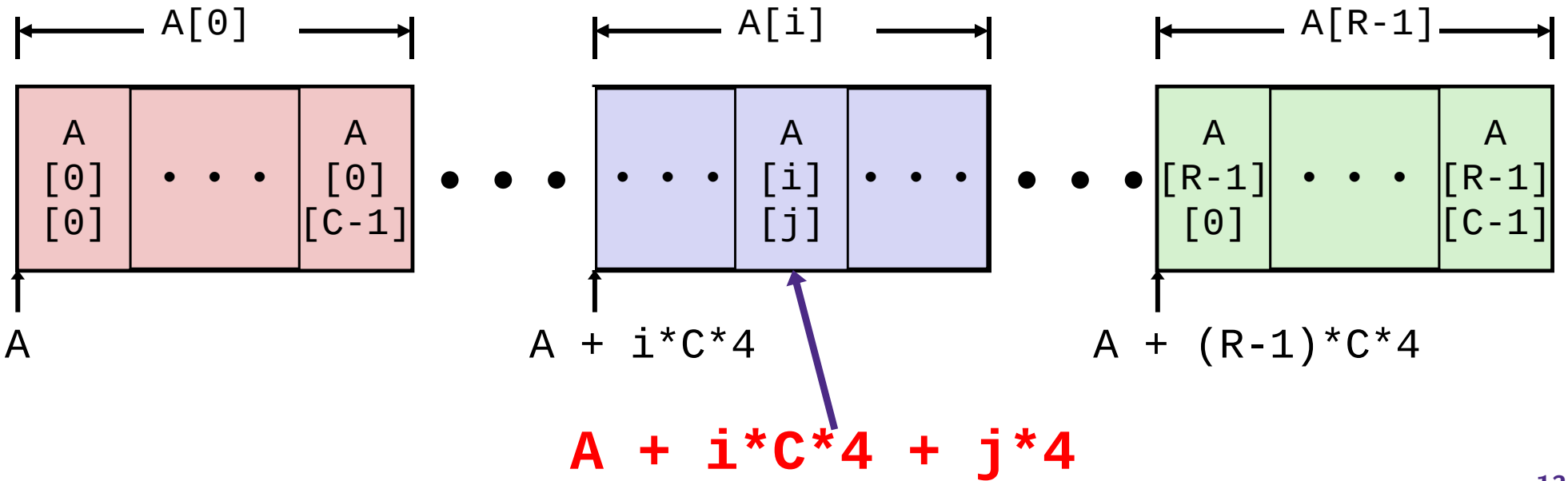


Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type `T`, which requires *K* bytes
  - Address of `A[i][j]` is

    `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 digit)
{
  return sea[index][digit];
}
```

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

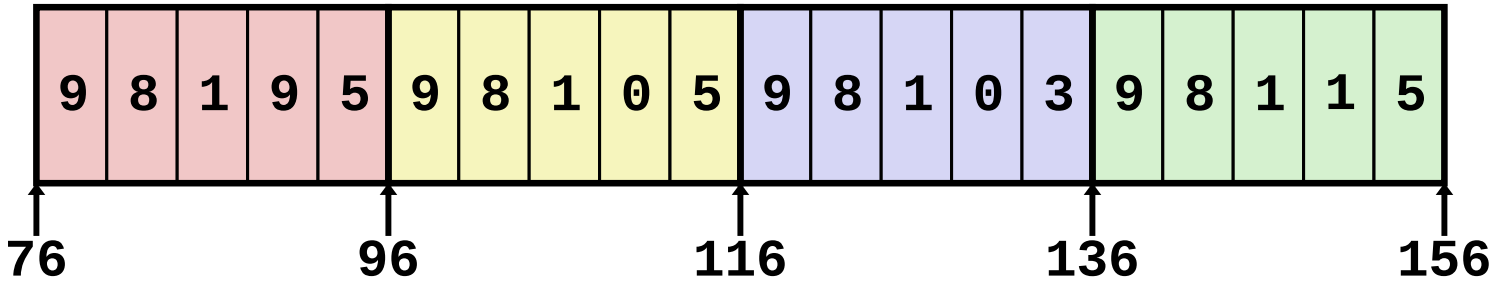
```
leaq   (%rdi,%rdi,4), %rax   # 5*index
addl   %rax, %rsi            # 5*index+digit
movl   sea(,%rsi,4),  %eax   # *(sea + 4*(5*index+digit))
```

- Array Elements
  - `sea[index][digit]` is an **`int`** (**`sizeof(int)`**=4)
  - Address = `sea + 5*4*index + 4*digit`
- Assembly Code
  - Computes address as: `sea + ((index+4*index) + digit)*4`
  - `movl` performs memory reference

```
typedef int zip_dig[5];
```

Strange Referencing Examples

```
zip_dig sea[4];
```



Reference	Address	Value	Guaranteed?
`sea[3][3]`			
`sea[2][5]`			
`sea[2][-1]`			
`sea[4][-1]`			
`sea[0][19]`			
`sea[0][-1]`			

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

Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - **Multi-level**
- Structs
  - Alignment
- ~~Unions~~

Multi-Level Array Example

Multi-Level Array Declaration(s):

```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {uw, cmu, ucb};
```

2D Array Declaration:

```
zip_dig univ2D[3] = {
  { 9, 8, 1, 9, 5 },
  { 1, 5, 2, 1, 3 },
  { 9, 4, 7, 2, 0 }
};
```

Is a multi-level array the same thing as a 2D array? **NO**

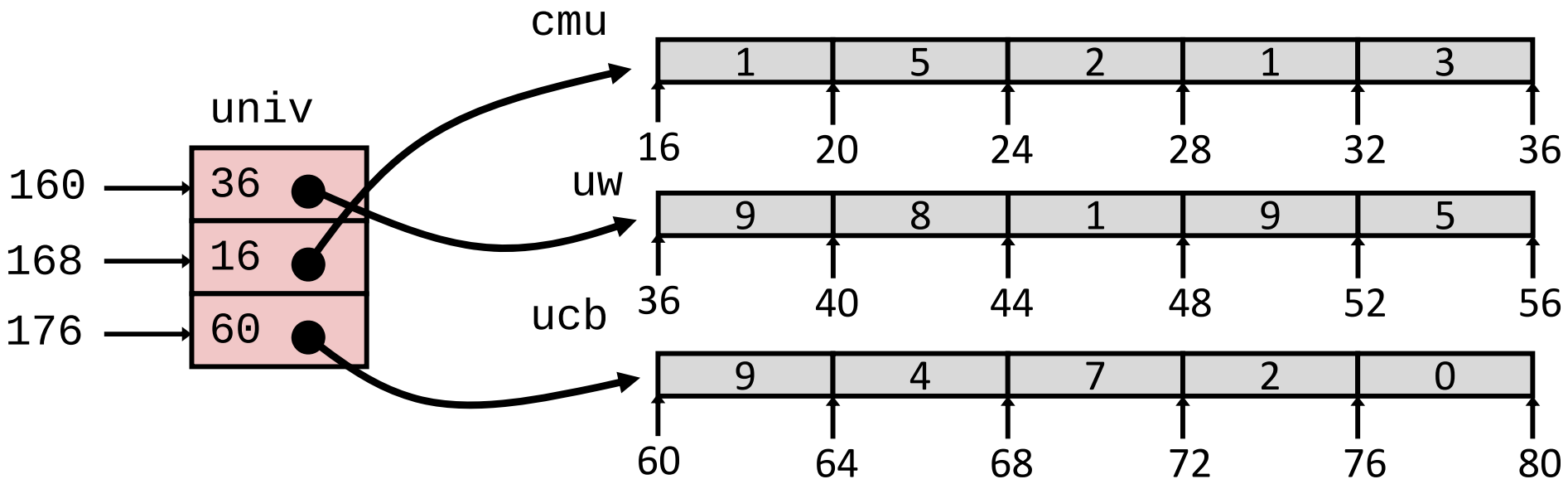
One array declaration = one contiguous block of memory

Multi-Level Array Example

```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {uw, cmu, ucb};
```

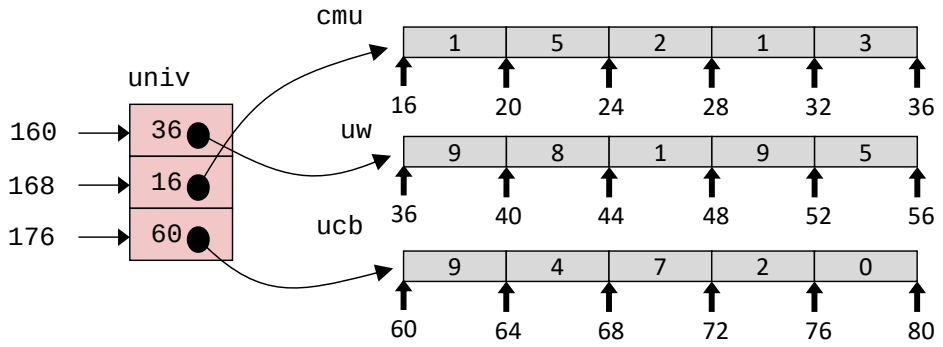
- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes each
- Each pointer points to array of `int`s



Note: this is how Java represents multi-dimensional arrays

Element Access in Multi-Level Array

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



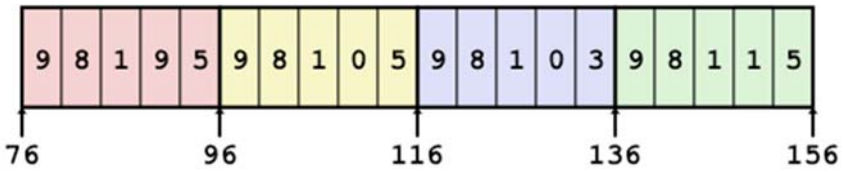

```
salq    $2, %rsi                # rsi = 4*digit
addq    univ(,%rdi,8), %rsi     # p = univ[index] + 4*digit
movl    (%rsi), %eax            # return *p
ret
```

- Computation
  - Element access Mem[Mem[`univ+8*index`]+`4*digit`]
  - Must do **two memory reads**
    - First get pointer to row array
    - Then access element within array
  - But allows inner arrays to be different lengths (not seen here)

Array Element Accesses

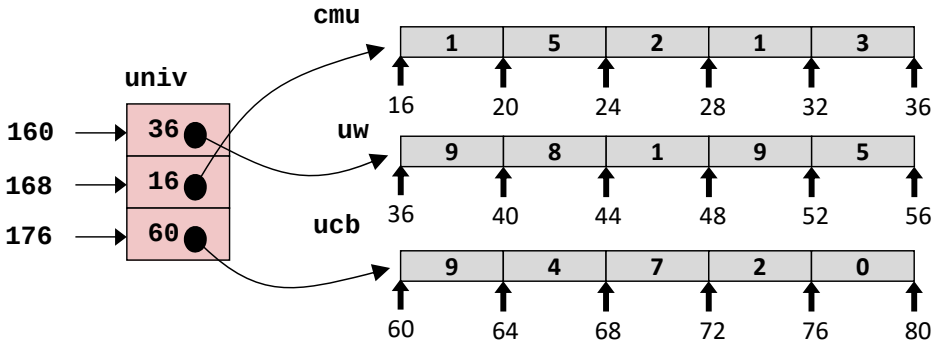
Nested array

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



Multi-level array

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

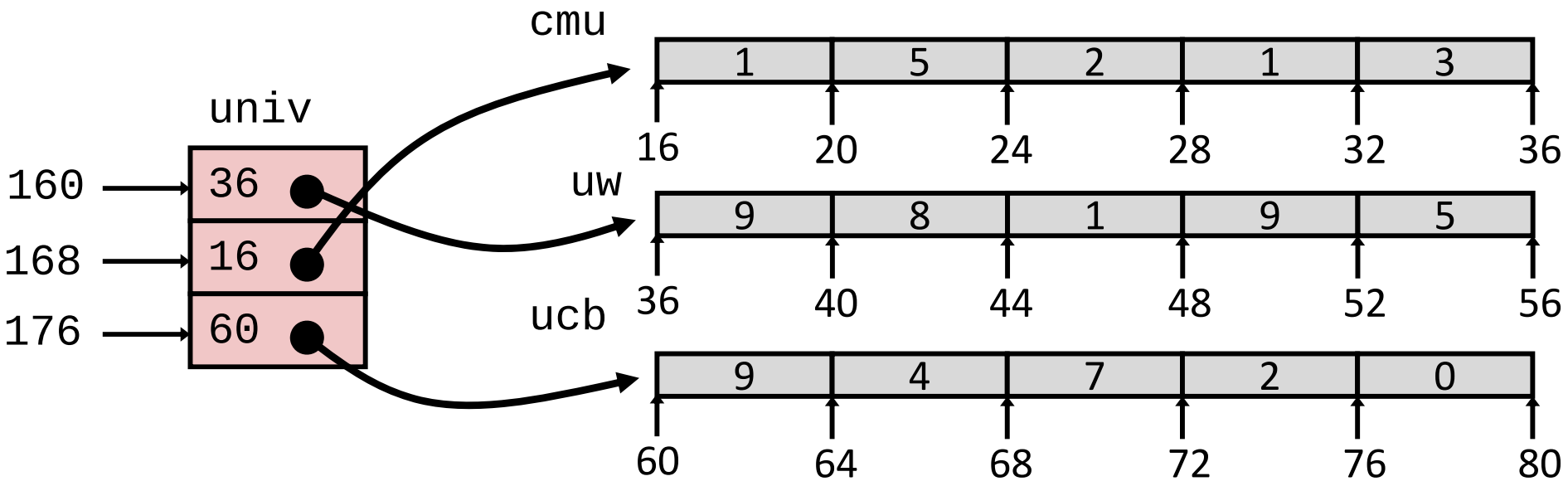


Access *looks* the same, but it isn't:

Mem[sea+20*index+4*digit]

Mem[Mem[univ+8*index]+4*digit]

Strange Referencing Examples



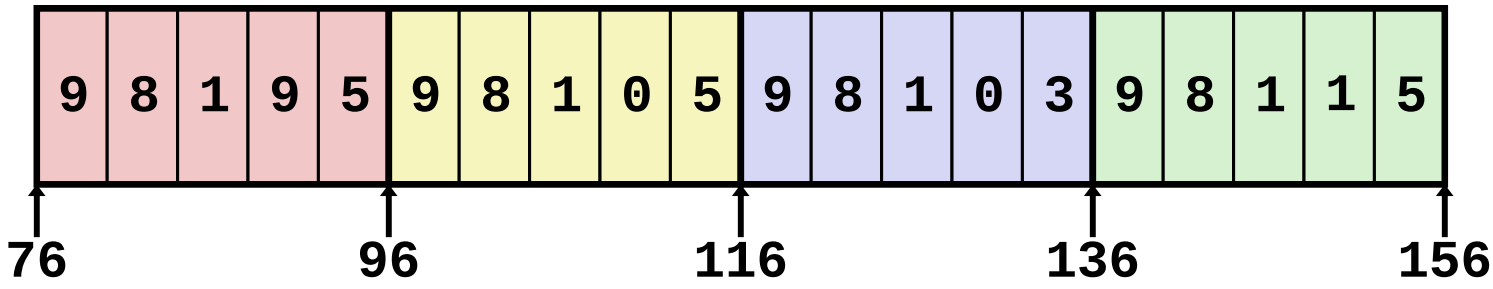
Reference	Address	Value	Guaranteed?
`univ[2][3]`			
`univ[1][5]`			
`univ[2][-2]`			
`univ[3][-1]`			
`univ[1][12]`			

- C code does not do any bounds checking
- Location of each lower-level array in memory is *not* guaranteed

Peer Instruction Question

- Which of the following statements is FALSE?
  - Vote at http://PollEv.com/justinh

```
int sea[4][5];
```



A. `sea[4][-2]` is a *valid* array reference

B. `sea[1][1]` makes *two* memory accesses

C. `sea[2][1]` will *always* be a higher address than `sea[1][2]`

D. `sea[2]` is calculated using *only* `lea`

E. We're lost…

Data Structures in Assembly

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structs**
  - **Alignment**
- ~~Unions~~

Structs in C

- Way of defining compound data types
- A structured group of variables, possibly including other structs

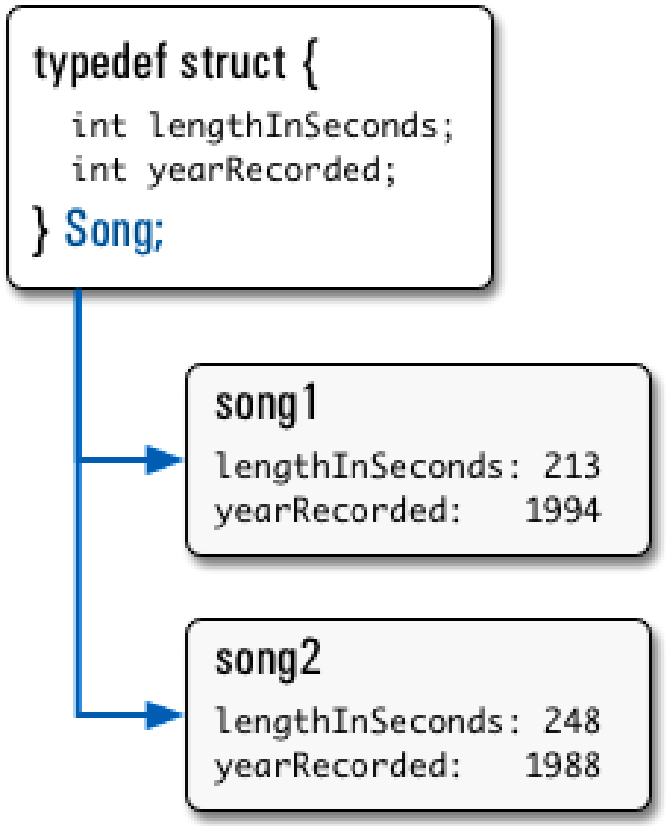
```
typedef struct {
  int lengthInSeconds;
  int yearRecorded;
} Song;

Song song1;

song1.lengthInSeconds =  213;
song1.yearRecorded    = 1994;

Song song2;

song2.lengthInSeconds =  248;
song2.yearRecorded    = 1988;
```



Struct Definitions

- Structure definition:
  - Does NOT declare a variable
  - Variable type is "**`struct name`**"

```
struct name {
   /* fields */
};
```

Easy to forget semicolon!

pointer

```
struct name name1, *pn, name_ar[3];
```

array

- Joint struct definition and typedef
  - Don't need to give struct a name in this case

```
struct nm {
   /* fields */
};
typedef struct nm name;
name n1;
```

```
typedef struct {
    /* fields */
} name;
name n1;
```

Scope of Struct Definition

- Why is placement of struct definition important?
  - What actually happens when you declare a variable?
    - Creating space for it somewhere!
  - Without definition, program doesn't know how much space

```
struct data {
   int ar[4];
   long d;
};
```

← Size = _____ bytes

Size = _____ bytes →

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

- Almost always define structs in global scope near the top of your C file
  - Struct definitions follow normal rules of scope

Accessing Structure Members

- Given a struct instance, access member using the . operator:

```
struct rec r1;
r1.i = val;
```

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

- Given a *pointer* to a struct:

```
struct rec *r;
r = &r1;  // or malloc space for r to point to
```

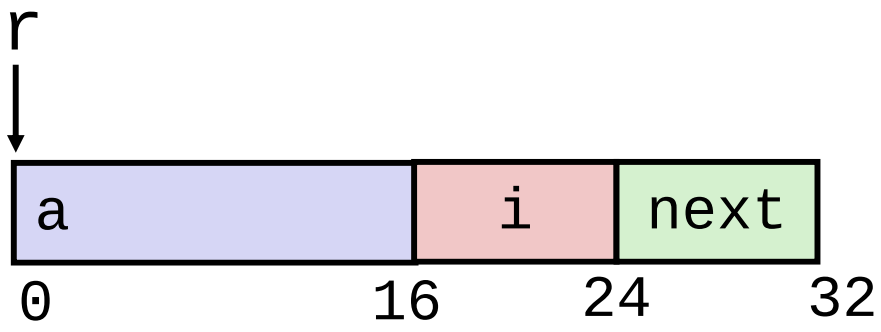
We have two options:

- Use * and . operators: `(*r).i = val;`
- Use -> operator for short: `r->i = val;`

- **In assembly:** register holds address of the first byte
  - Access members with offsets

Structure Representation

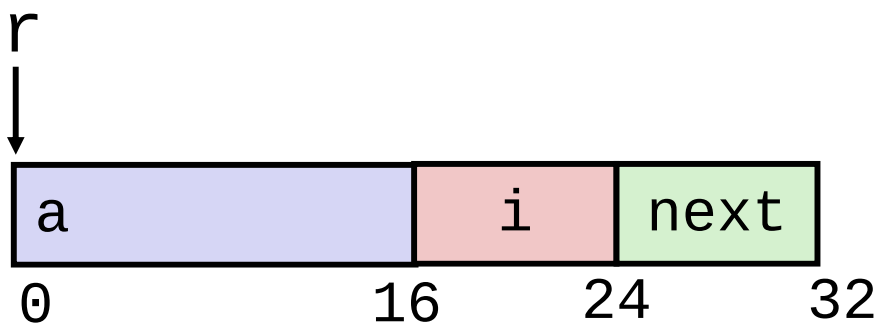
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



- Characteristics
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

Structure Representation

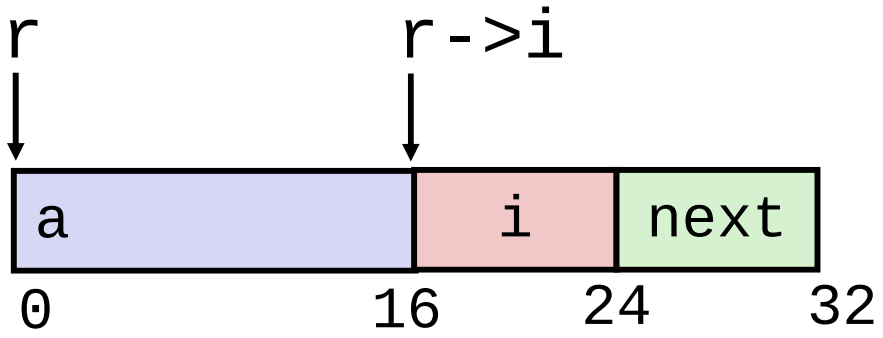
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering would be more compact
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

Accessing a Structure Member

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



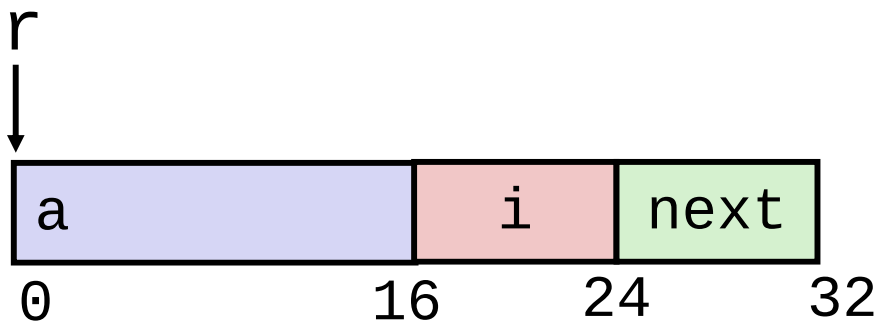
- Compiler knows the *offset* of each member within a struct
  - Compute as `*(r+offset)`
    - Referring to absolute offset, so no pointer arithmetic

```
long get_i(struct rec *r)
{
  return r->i;
}
```

```
  # r in %rdi, index in %rsi
  movq  16(%rdi), %rax
  ret
```

Pointer to Structure Member

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




```
long* addr_of_i(struct rec *r)
{
  return &(r->i);
}
```

```
  # r in %rdi
  _____  __16___,%rax
  ret
```

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

```
  # r in %rdi
  _____  __24___,%rax
  ret
```

Review: Memory Alignment in x86-64

- For good memory system performance, Intel recommends data be aligned
  - However the x86-64 hardware will work correctly regardless of alignment of data
- *Aligned* means that any primitive object of K bytes must have an address that is a multiple of K
- Aligned addresses for data types:

K	Type	Addresses
1	`char`	No restrictions
2	`short`	Lowest bit must be zero: $...0_2$
4	`int, float`	Lowest 2 bits zero: $...00_2$
8	`long, double, *`	Lowest 3 bits zero: $...000_2$
16	`long double`	Lowest 4 bits zero: $...0000_2$

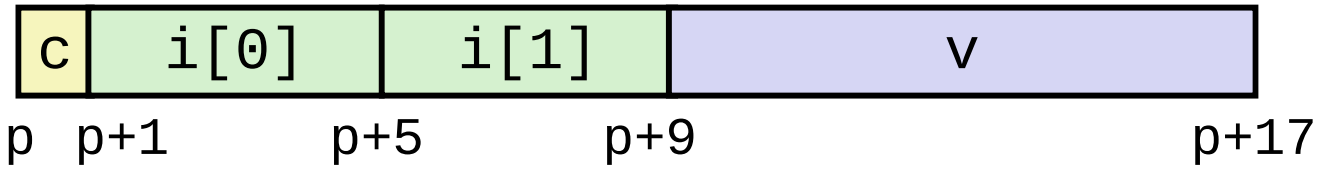
Alignment Principles

- 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 value that spans quad word boundaries
    - Virtual memory trickier when value spans 2 pages (more on this later)

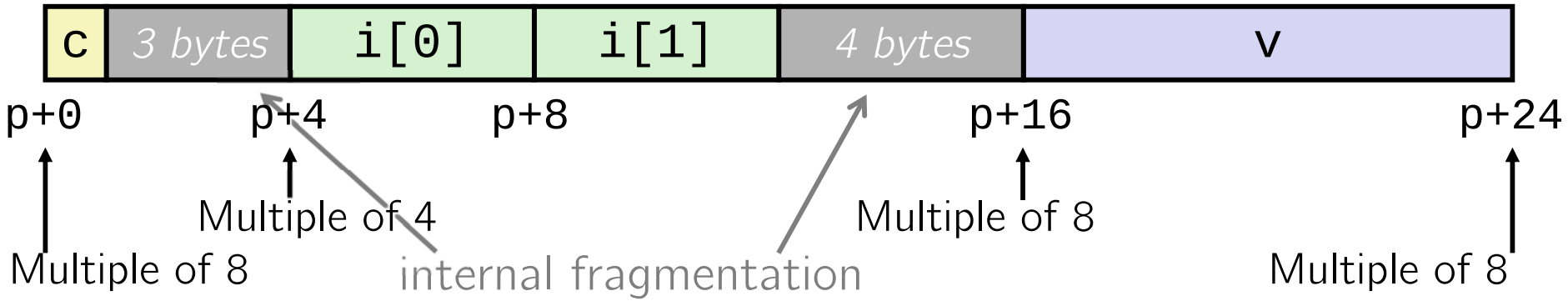
Structures & Alignment

- Unaligned Data




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

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K

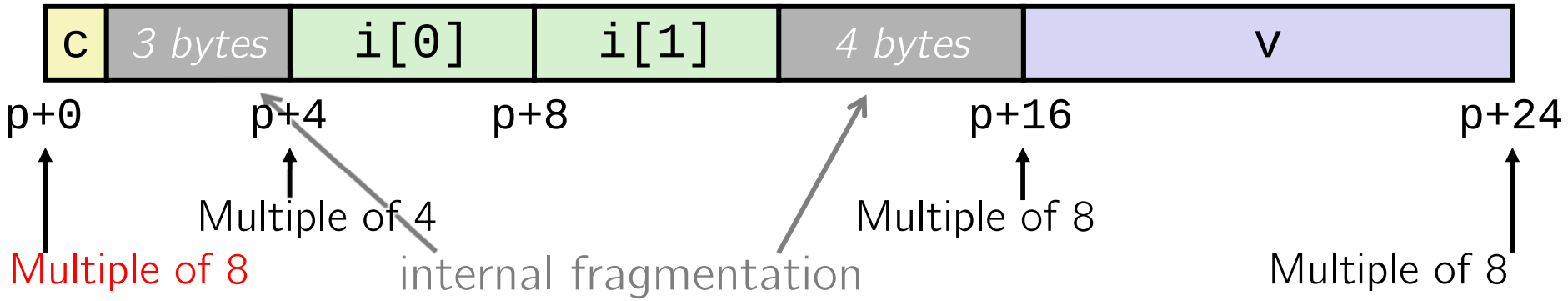


Satisfying Struct Alignment (1)

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement $K_{\max}$
    - $K_{\max}$ = Largest alignment of any element
    - Counts array elements individually as elements

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

- Example:
  - $K_{\max}$ = 8, due to `double` element

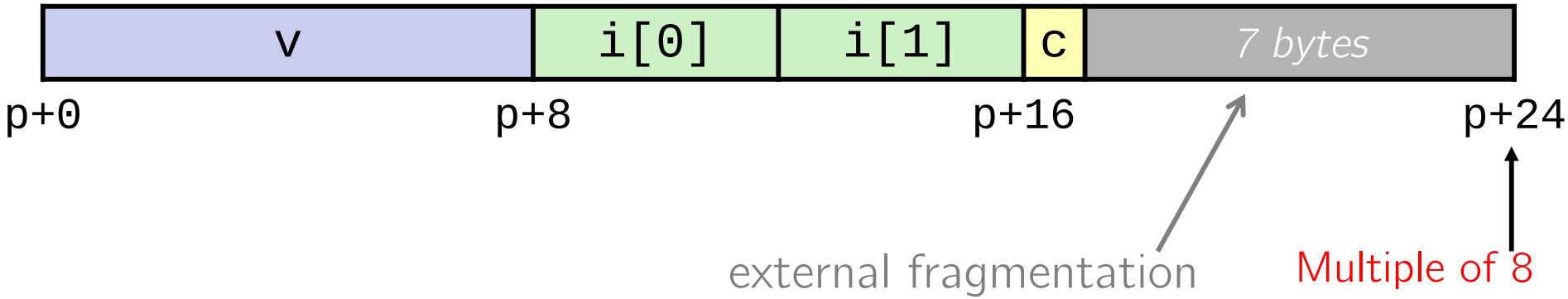


Satisfying Struct Alignment (2)

- Can find offset of individual fields using `offsetof()`
  - Need to `#include <stddef.h>`
  - Example: `offsetof(struct S2,c)` returns 16

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

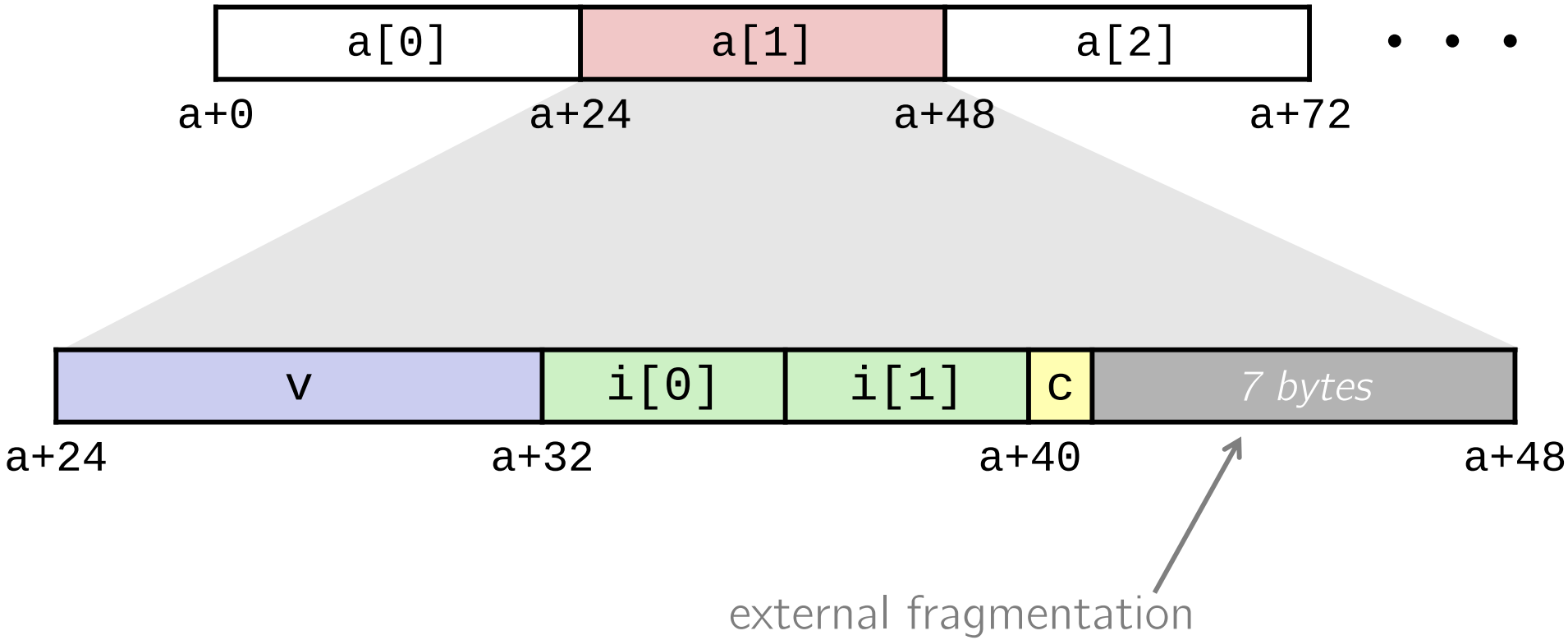
- For largest alignment requirement $K_{\max}$, overall structure size must be multiple of $K_{\max}$
  - Compiler will add padding at end of structure to meet overall structure alignment requirement



Arrays of Structures

- Overall structure length multiple of K_{max}
- Satisfy alignment requirement for every element in array

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



Alignment of Structs

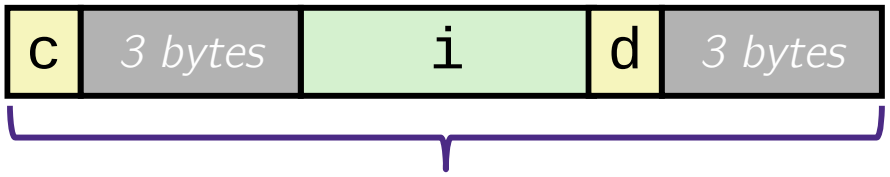
- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each ***field*** must be aligned *within* the struct
    (may insert padding)
    - `offsetof` can be used to get actual field offset
  - Overall struct must be ***aligned*** according to largest field
  - Total struct ***size*** must be multiple of its alignment
    (may insert padding)
    - `sizeof` should be used to get true size of structs

How the Programmer Can Save Space

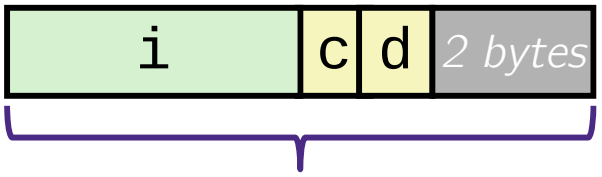
- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

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

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



12 bytes



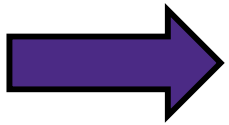
8 bytes

Peer Instruction Question

Vote on sizeof(**struct old**): http://PollEv.com/justinh

- Minimize the size of the struct by re-ordering the vars

```
struct old {
  int i;
  short s[3];
  char *c;
  float f;
};
```



```
struct new {
  int    i;
  _____ _____;
  _____ _____;
  _____ _____;
};
```

- What are the old and new sizes of the struct?

sizeof(struct old) = _____ sizeof(struct new) = _____

A. 16 bytes

B. 22 bytes

C. 28 bytes

D. 32 bytes

E. We're lost...

Summary

- Arrays are contiguous allocations of memory
  - No bounds checking (and no default initialization)
- `int a[4][5];` → array of arrays
  - all levels in one contiguous block of memory
- `int* b[4];` → array of pointers to arrays
  - First level in one contiguous block of memory
  - Each element in the first level points to another "sub" array
  - Parts anywhere in memory
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment