Structs & Alignment

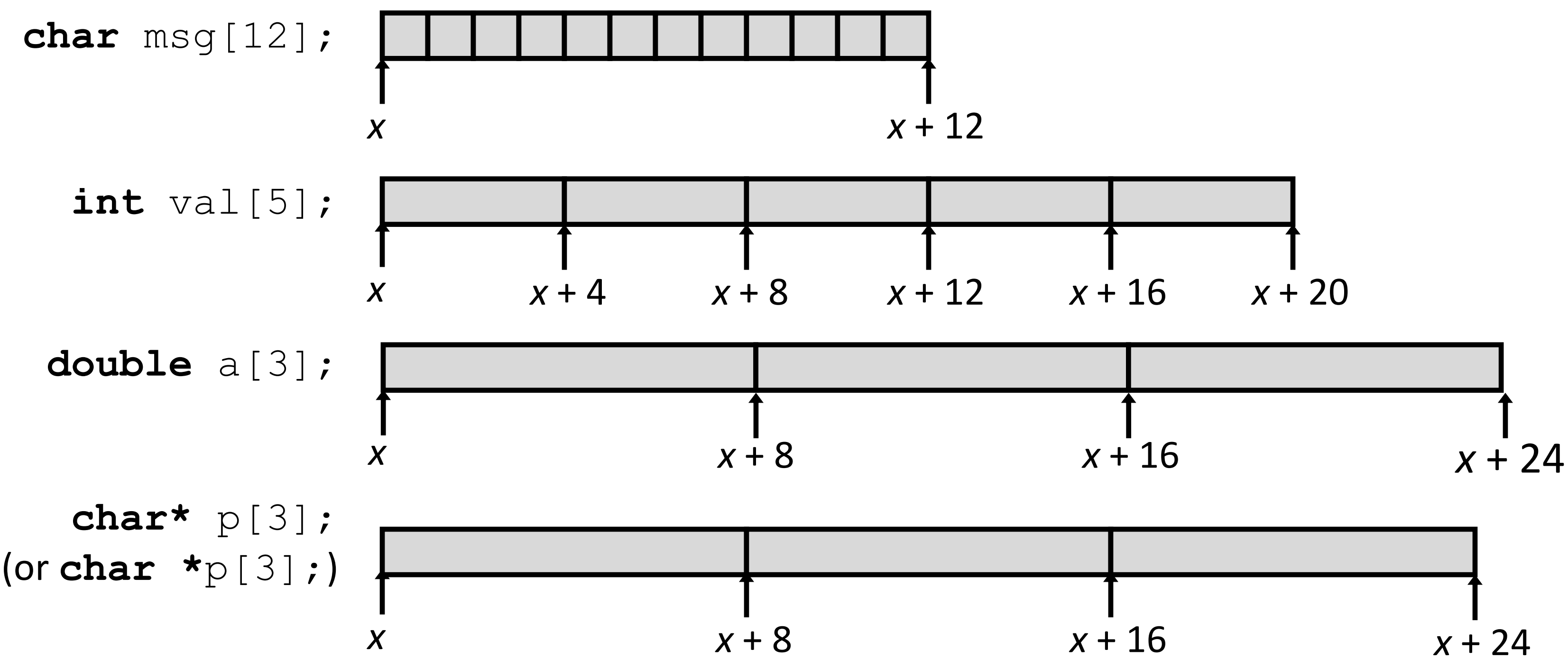
Acknowledgments: These slides have been modified by Arrvindh Shriraman, Justin Tsia

Data Structures in Assembly

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

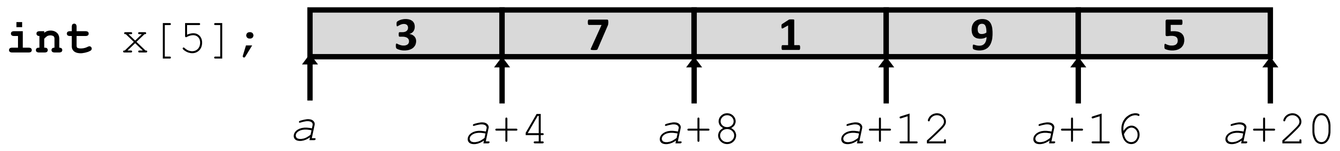
Array Allocation

- Basic Principle
  - `T A[N];` → array of data type `T` and length `N`
  - *Contiguously* allocated region of `N*sizeof(T)` bytes
  - Identifier `A` returns address of array (type `T*`)



Array Access

- Basic Principle
  - **T** A[N]; → array of data type **T** and length N
  - Identifier A returns address of array (type **T***)



Reference	Type	Value
x[4]	**int**	5
x	**int***	a
x+1	**int***	a + 4
&x[2]	**int***	a + 8
x[5]	**int**	?? (whatever's in memory at addr x+20)
*(x+1)	**int**	7
x+i	**int***	a + 4*i

Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig sfu  = { 9, 8, 1, 9, 5
};
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

initialization

- `typedef`: Declaration "**`zip_dig`** `sfu`" equivalent to "**`int`** `sfu[5]`"

C Details: Arrays and Pointers

- Arrays are (almost) identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - Differ in subtle ways: initialization, `sizeof()`, etc.
- An array name looks like a pointer to the first (0^{th}) element
  - `ar[0]` same as `*ar`; `ar[2]` same as `*(ar+2)`
- An array name is read-only (no assignment)
  - Cannot use "`ar = <anything>`"

C Details: Arrays and Functions

- Declared arrays only allocated while the scope is valid:

```
char* foo() {
    char string[32]; ...;
    return string;
}
```

BAD!

- An array is passed to a function as a pointer:
  - Array size gets lost!

Really `int *ar`

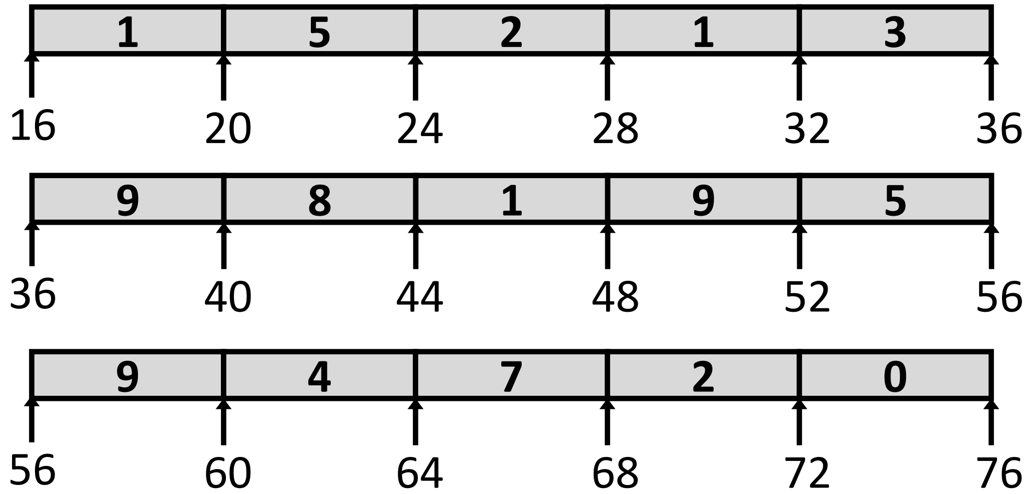
```
int foo(int ar[], unsigned int size) {
    ... ar[size-1] ...
}
```

Must explicitly pass the size!

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

Referencing Examples

```
zip_dig cmu;
zip_dig sfu;
zip_dig ucb;
```



Reference	Address	Value	Guaranteed?
`sfu[3]`	`36 + 4* 3 = 48`	`9`	Yes
`sfu[6]`	`36 + 4* 6 = 60`	`4`	No
`sfu[-1]`	`36 + 4*-1 = 32`	`3`	No
`cmu[15]`	`16 + 4*15 = 76`	`??`	No

- No bounds checking
- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

```
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];
```

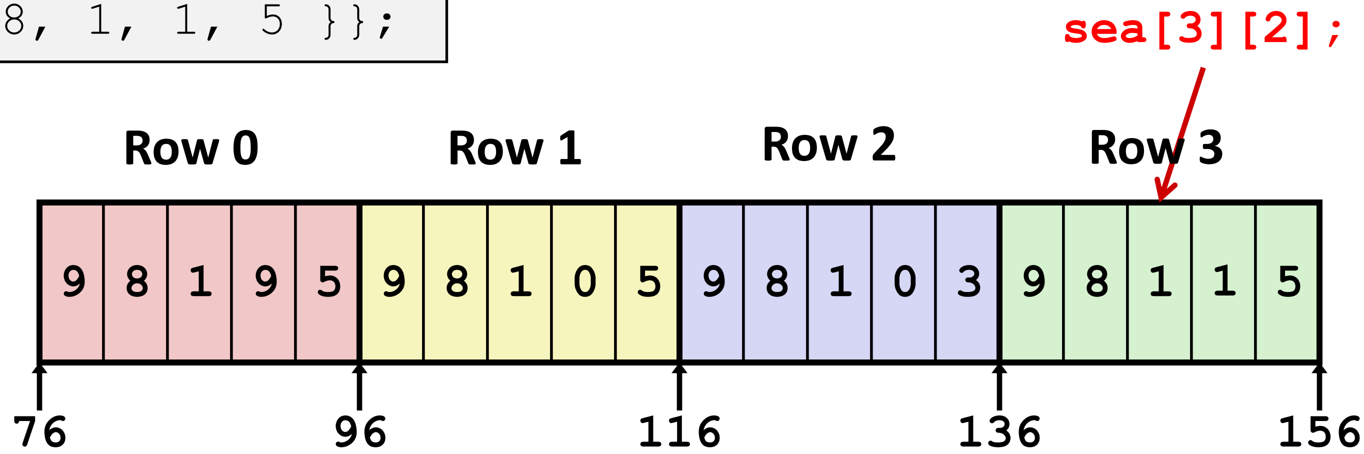
What is the layout in memory?

```
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`



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

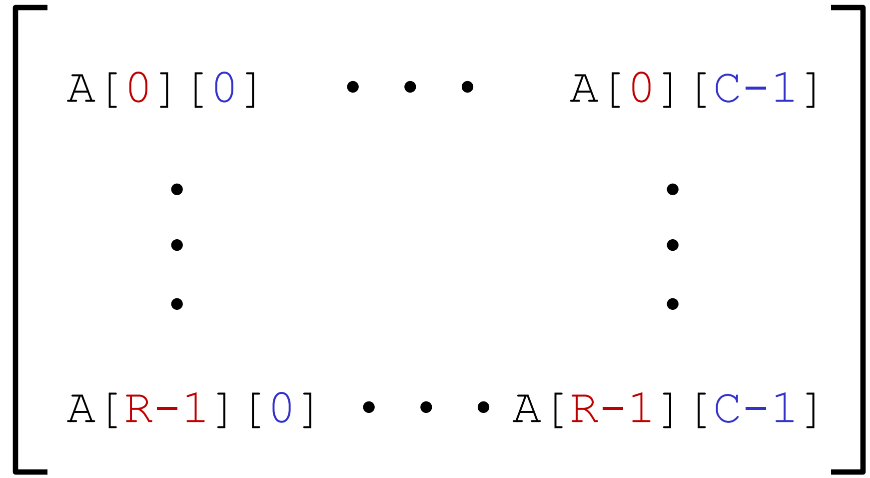
Two-Dimensional (Nested) Arrays

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

`A[0][0]`	• • •	`A[0][C-1]`
⋮		⋮
`A[R-1][0]`	• • •	`A[R-1][C-1]`

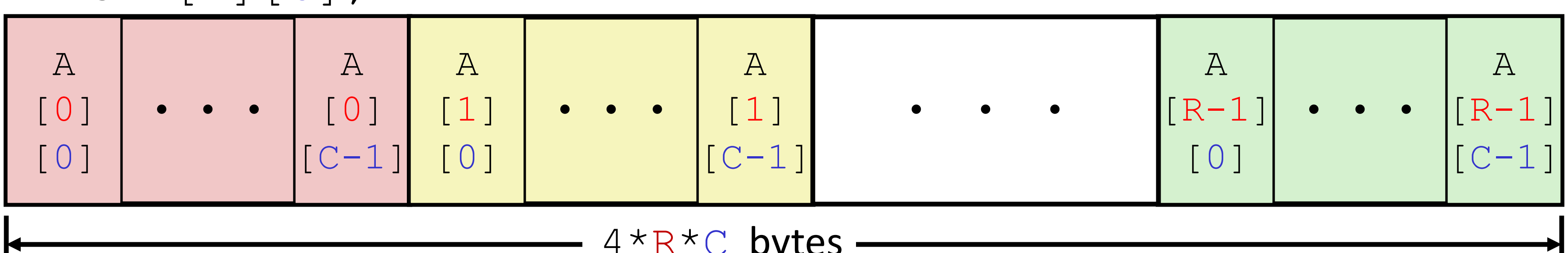
Two-Dimensional (Nested) Arrays

- 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];`



Multi-Level Array Example

Multi-Level Array Declaration(s):

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

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

Is a multi-level array the same thing as a 2D array?

NO

2D Array Declaration:

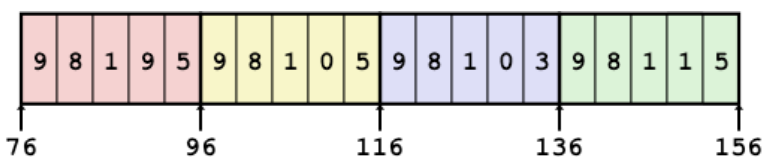
```
zip_dig univ2D[3] = {
  { 9, 8, 1, 9, 5 },
  { 1, 5, 2, 1, 3 },
  { 9, 4, 7, 2, 0 }
};
```

One array declaration = one contiguous block of memory

Array Element Accesses

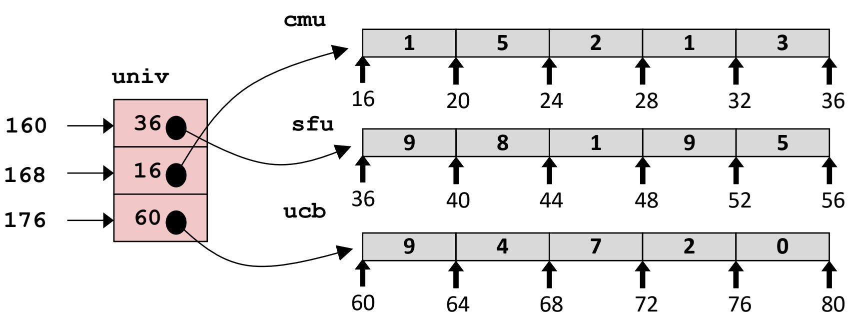
Nested array

```
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];
}
```

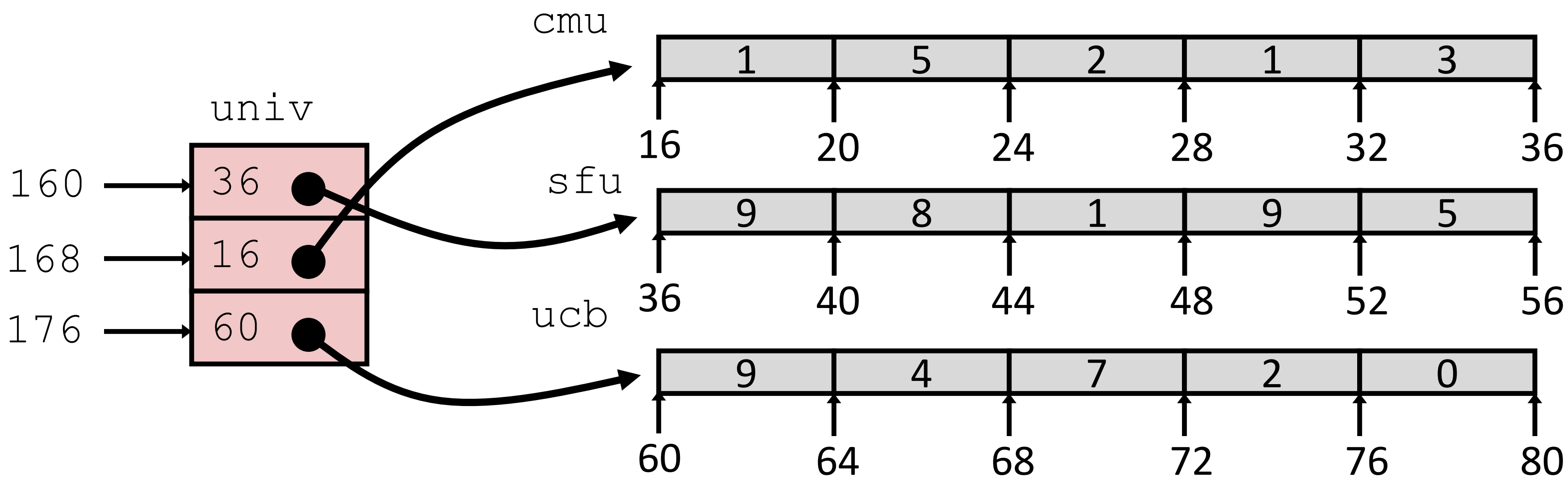


Access *looks* the same, but it isn't:

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

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

Multi-Level 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

Summary

- Contiguous allocations of memory
- No bounds checking (and no default initialization)
- Can usually be treated like a pointer to first element
- `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

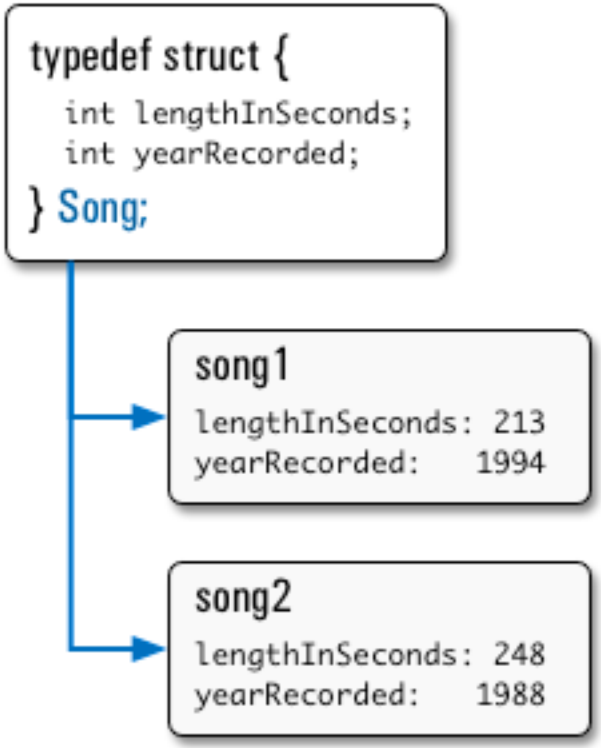
Data Structures in Assembly

- 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;
```



Accessing Structure Members

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

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

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

- Given a *pointer* to a struct:

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

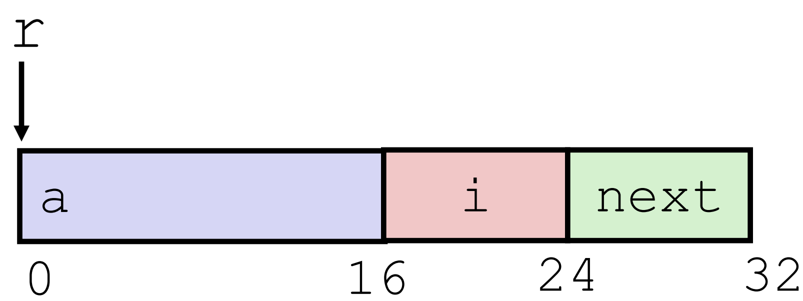
We have two options:

- 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

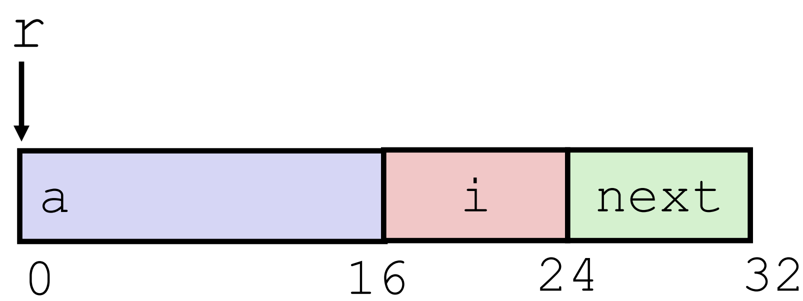
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};

struct rec *r;
```



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

Structure Representation

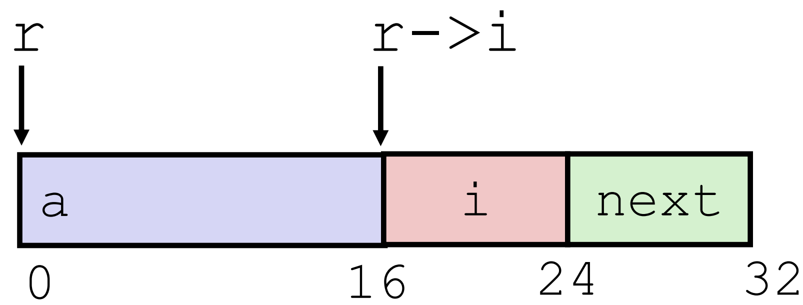
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};

struct rec *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;
};

struct rec *r;
```



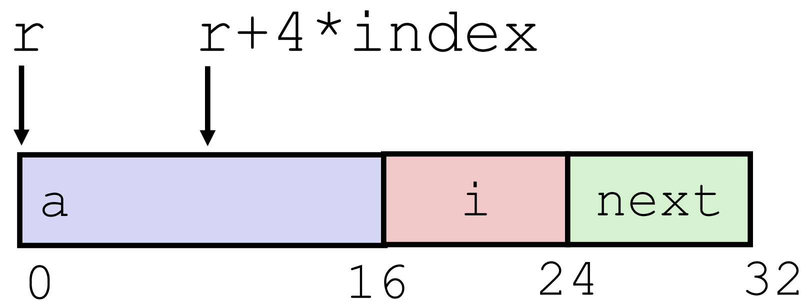

```
long get_i(struct rec *r)
{
  return r->i;
}
```

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

```
add  a0, a1, 16 # Coming up in Week 3
  ret
```

Generating Pointer to Array Element

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

struct rec *r;
```




```
int* find_addr_of_array_elem
   (struct rec *r, long index)
{
   return &r->a[index];
}
```

&(r->a[index])

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as: `r+4*index`

Struct Definitions

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

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

Easy to forget semicolon!

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

pointer (*pn), array (name_ar[3])

- 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

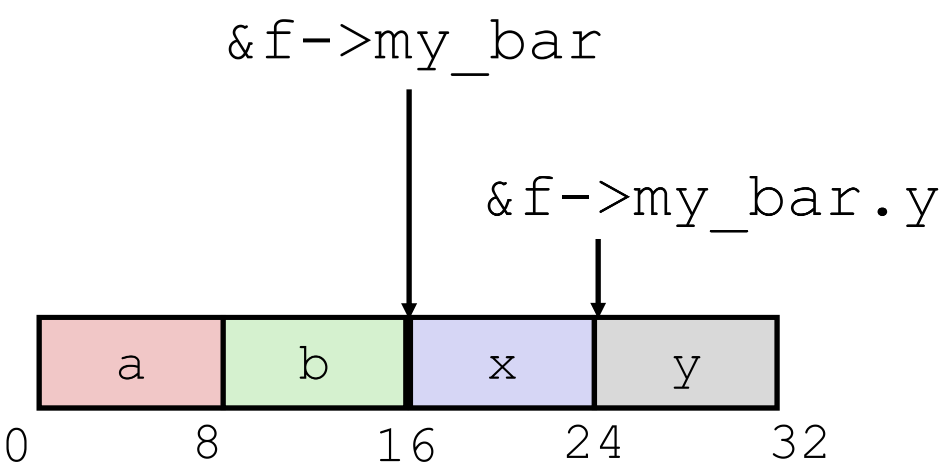
Size = _____ bytes →

```
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

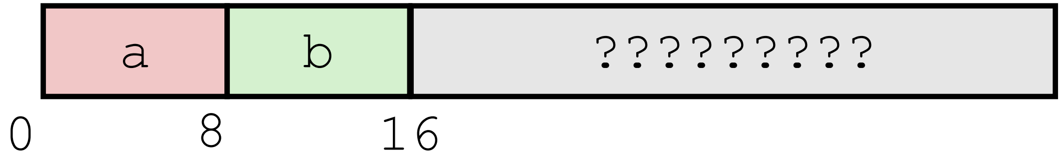
Nested Struct

```
struct foo {
    long a;
    long b;
    struct bar my_bar;
};

struct bar {
    long x;
    long y;
};

struct foo *f;
```



Nested Struct

```
struct foo {
    long a;
    long b;
    struct foo my_foo;
};
```



Review: Memory Alignment

- *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

Alignment Principles

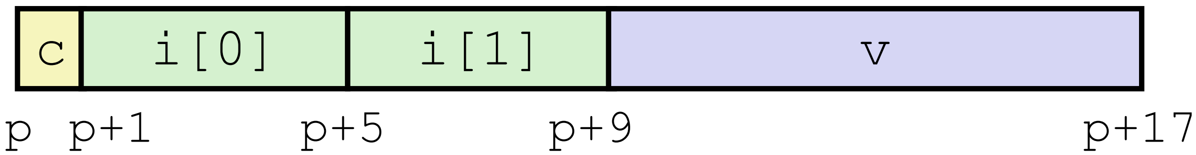
- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K

- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of bytes (width is 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)

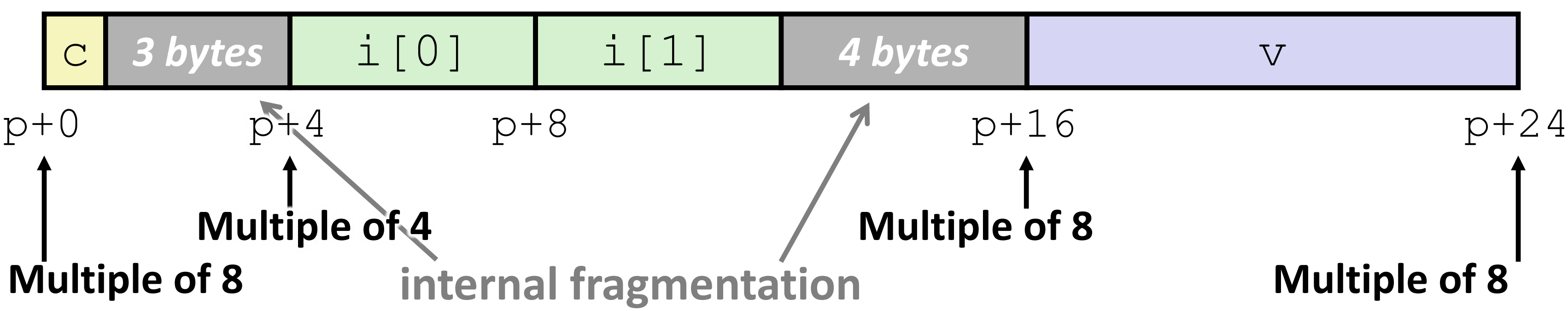
Structures & Alignment

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

- Unaligned Data



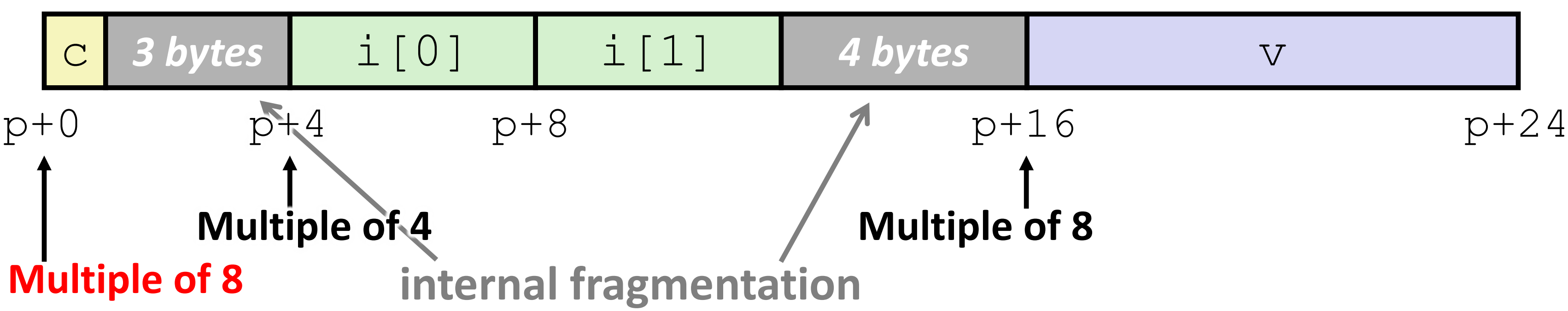
- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



Satisfying Alignment with Structures (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
    - Inner structs are aligned to *their* largest alignment
- Example:
  - K_{max} = 8, due to `double` element

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

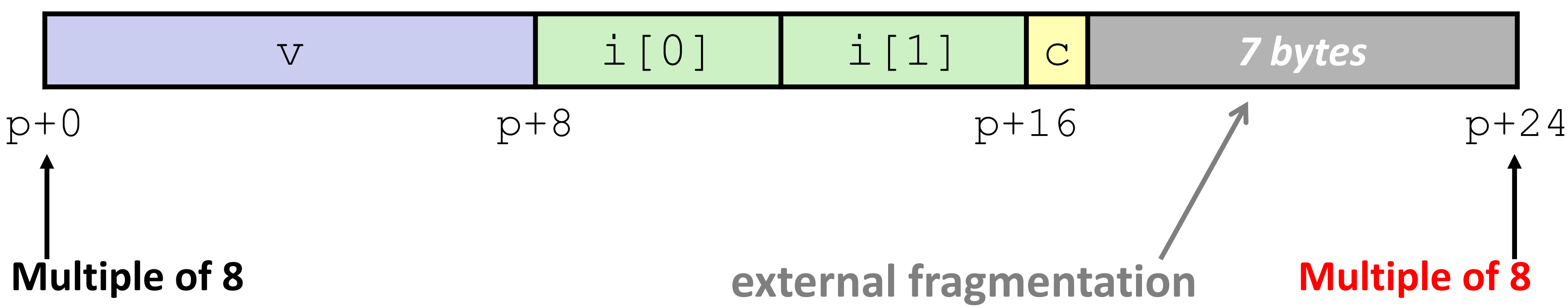


Satisfying Alignment with Structures (2)

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

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

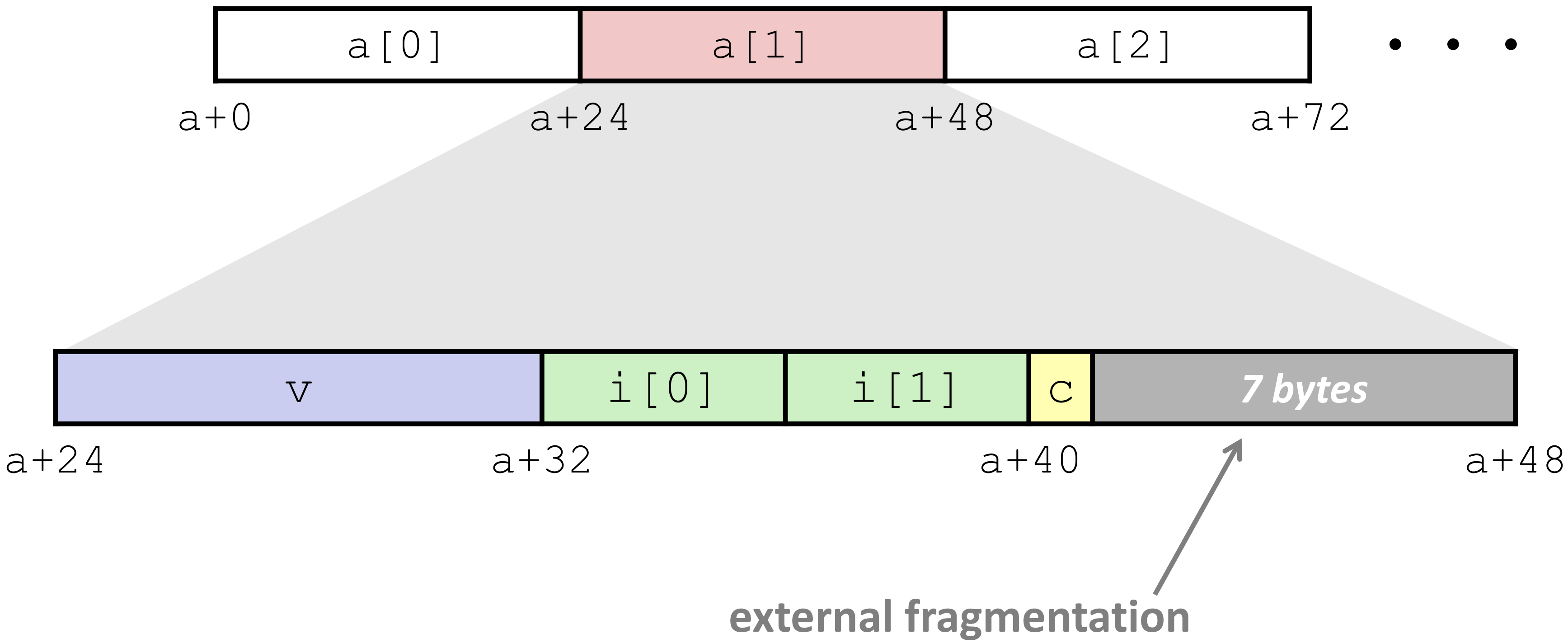
- For largest alignment requirement K_{max}, overall structure size must be multiple of K_{max}
  - Compiler will add padding at end of structure to meet overall structure alignment requirement



Arrays of Structures

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

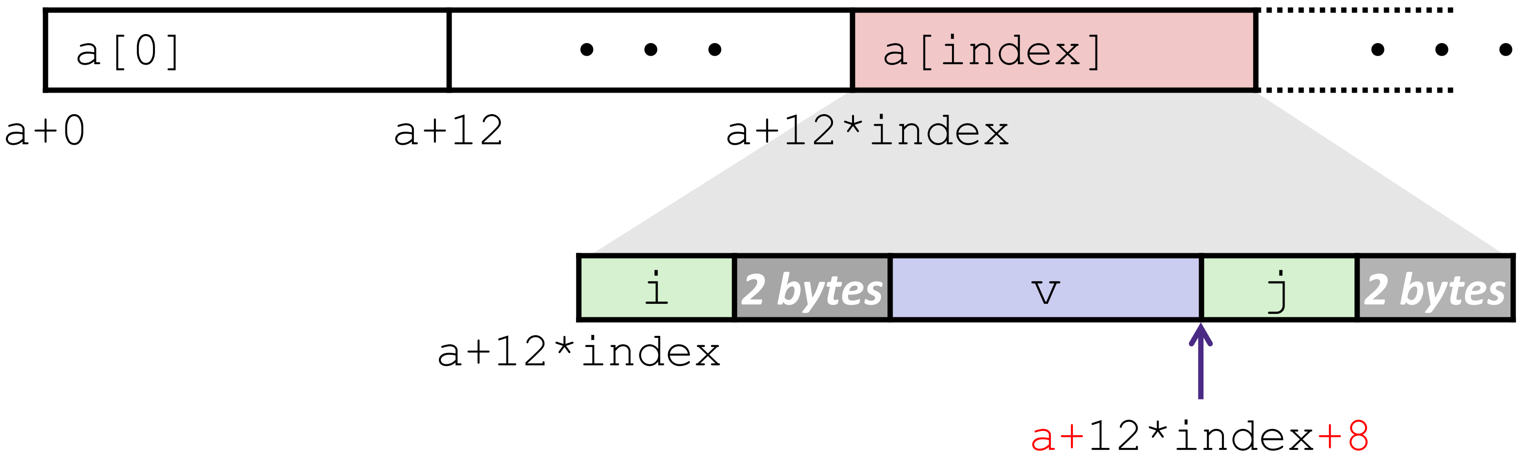
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

- Compute start of array element as: `12*index`
  - `sizeof(S3) = 12`, including alignment padding
- 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 index)
{
  return a[index].j;
}
```

Alignment of Structs

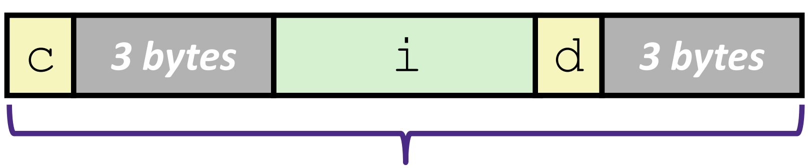
- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each ***field*** must be aligned *within* the struct *(may insert padding)*
    - `offsetof` can be used to get actual field offset
  - Overall struct must be ***aligned*** according to largest field
  - Total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

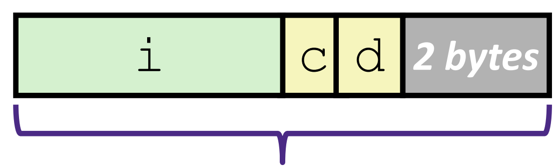
- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

```
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

- 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) = _____

Summary

- Arrays in C
  - Aligned to satisfy every element's alignment requirement
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment