Computer Systems

CSE 410 Autumn 2013

8 – Data Structures: Arrays and Structs

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

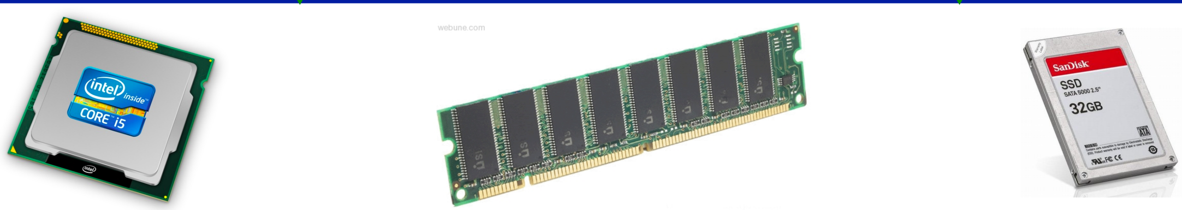
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:

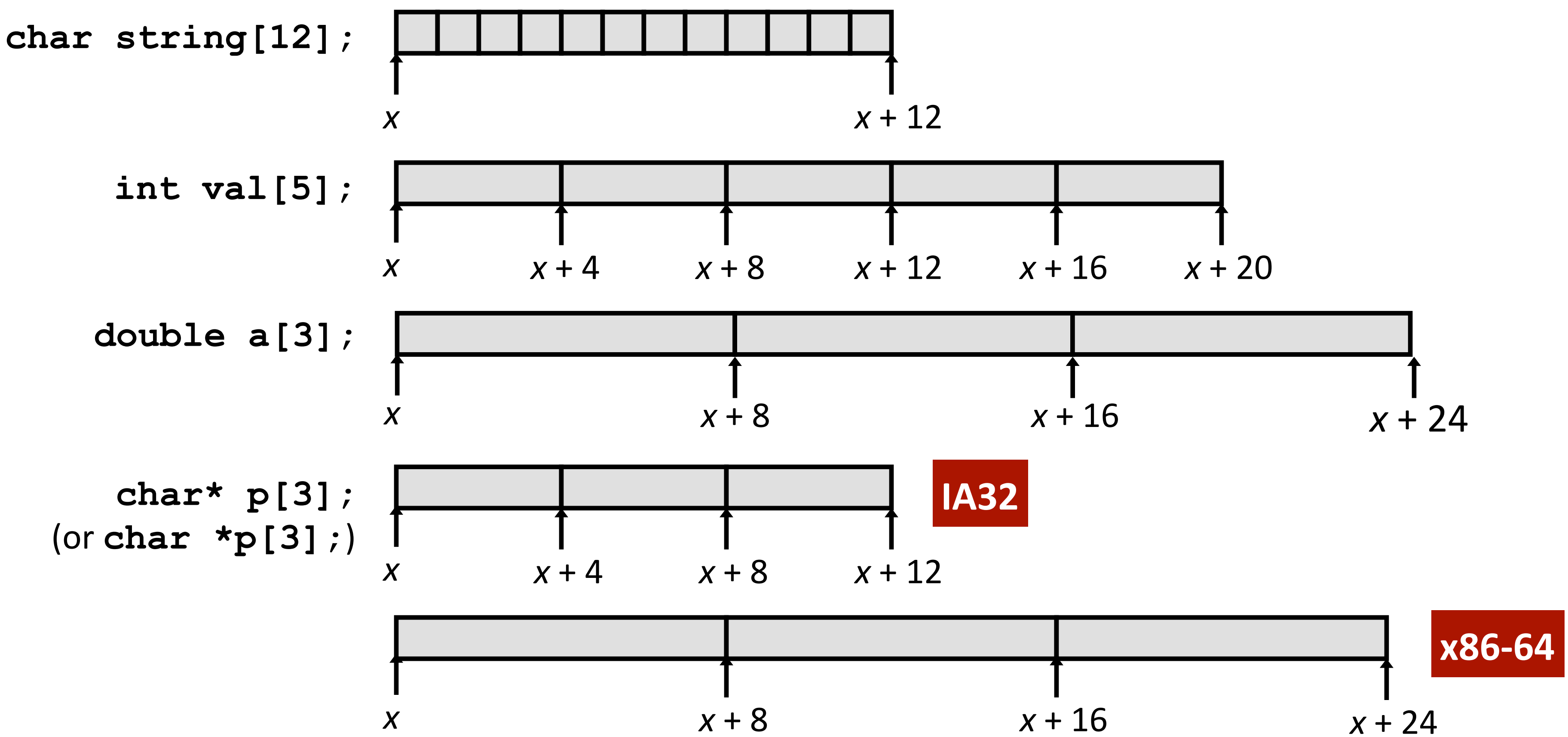


Arrays & Other Data Structures

- <u>**Array allocation and access in memory**</u>
- **Multi-dimensional or nested arrays**
- **Multi-level arrays**
- **Other structures in memory**
- **Data structures and alignment**

Array Allocation

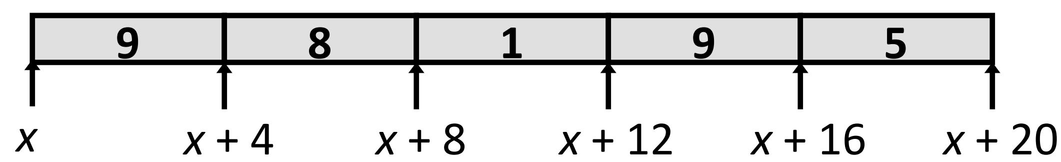
- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - *Contiguously* allocated region of N * sizeof(T) bytes



Array Access

- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - Identifier A can be used as a pointer to array element 0: Type T*

```
int val[5];
```



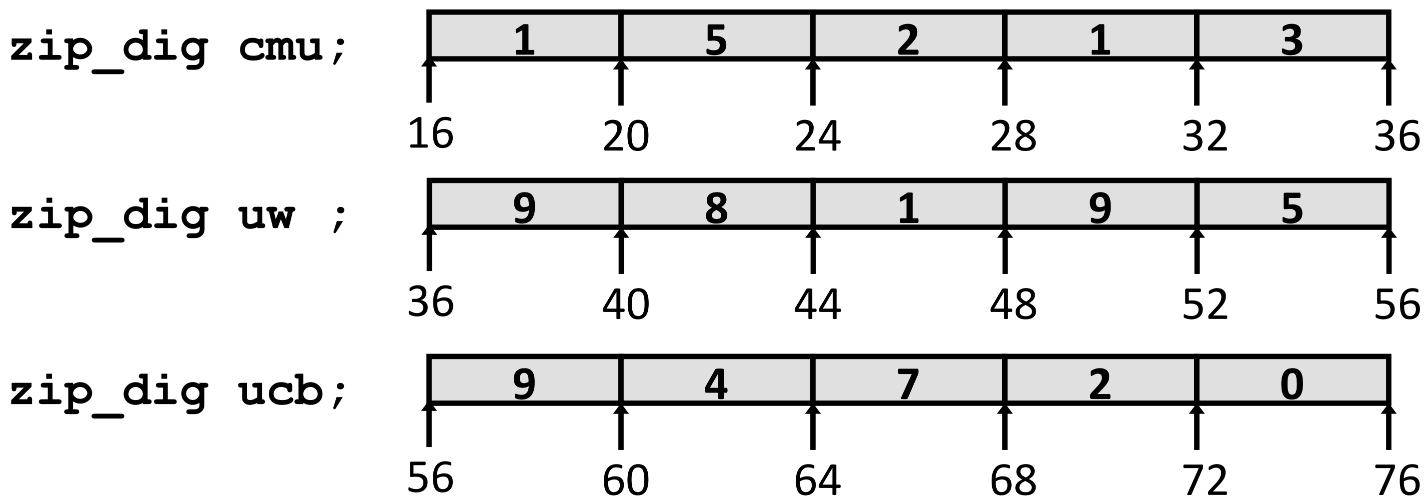
Reference	Type	Value
val[4]	int	5
val	int *	*x*
val+1	int *	*x* + 4
&val[2]	int *	*x* + 8
val[5]	int	?? (whatever is in memory at address *x* + 20)
*(val+1)	int	8
val + i	int *	*x* + 4*i

Array Example

```
typedef int zip_dig[5];

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

Array Example

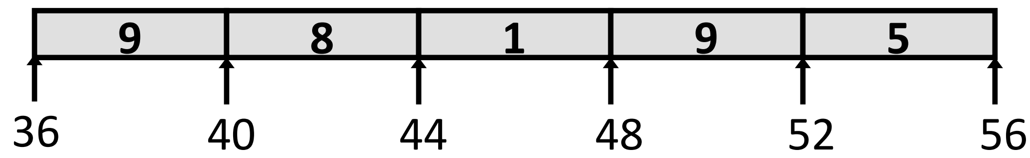
```
typedef int zip_dig[5];

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



- **Declaration "`zip_dig uw`" equivalent to "`int uw[5]`"**
- **Example arrays were allocated in successive 20 byte blocks**
  - Not guaranteed to happen in general

Array Accessing Example

```
zip_dig uw;
```



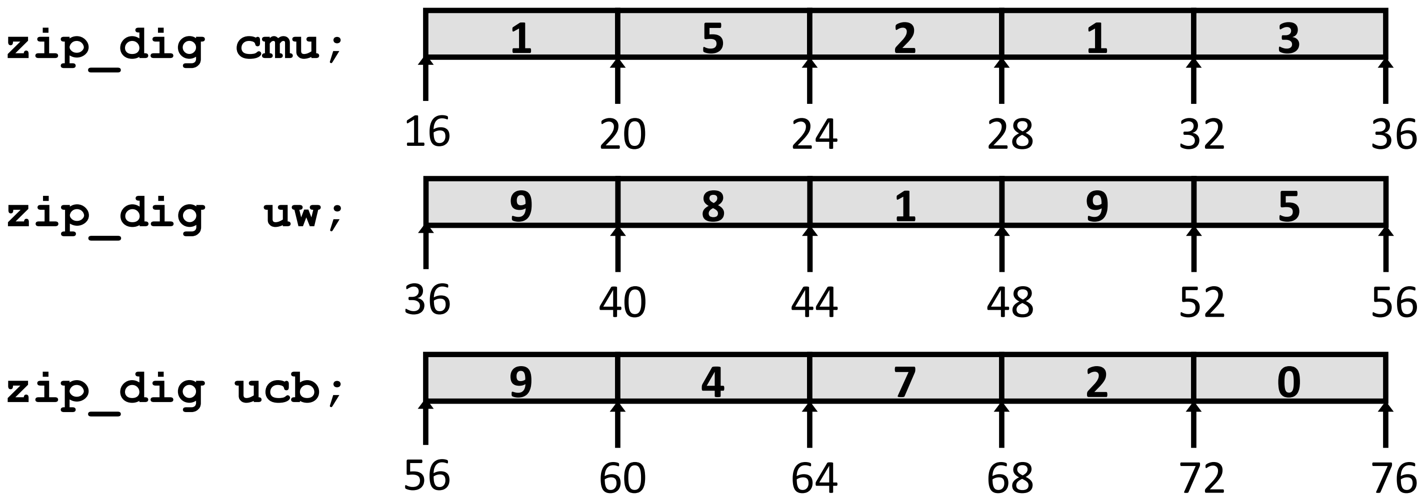

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

IA32

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax   # z[dig]
```

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

Referencing Examples



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

- No bounds checking
- Location of each separate array in memory is not guaranteed

Array Loop Example

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

Array Loop Example

- **Original**

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- **Transformed**
  - Eliminate loop variable **`i`**, use pointer **`zend`** instead
  - Convert array code to pointer code
    - Pointer arithmetic on **`z`**
  - Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z <= zend);
  return zi;
}
```

Array Loop Implementation (IA32)

- **Registers**

  `%ecx  z`
  `%eax  zi`
  `%ebx  zend`

- **Computations**
  - `10*zi + *z` implemented as `*z + 2*(5*zi)`
  - `z++` increments by 4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax              # zi = 0
  leal 16(%ecx),%ebx          # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx # zi + 4*zi = 5*zi
  movl (%ecx),%eax            # *z
  addl $4,%ecx                # z++
  leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx              # z : zend
  jle .L59                    # if <= goto loop
```

Arrays & Other Data Structures

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

Nested Array Example

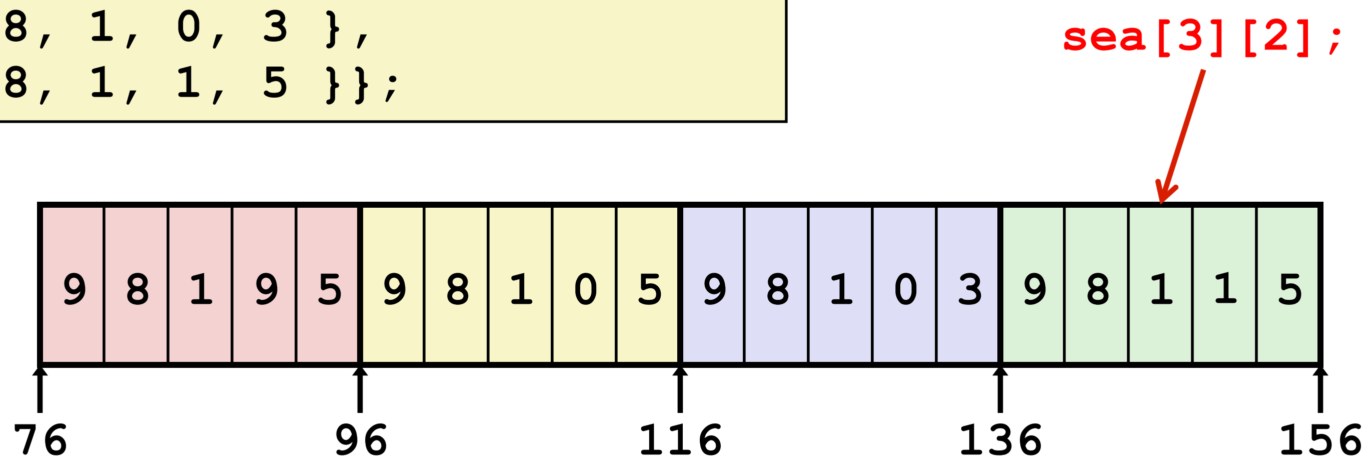
```
#define PCOUNT 4
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

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 <u>guaranteed</u>

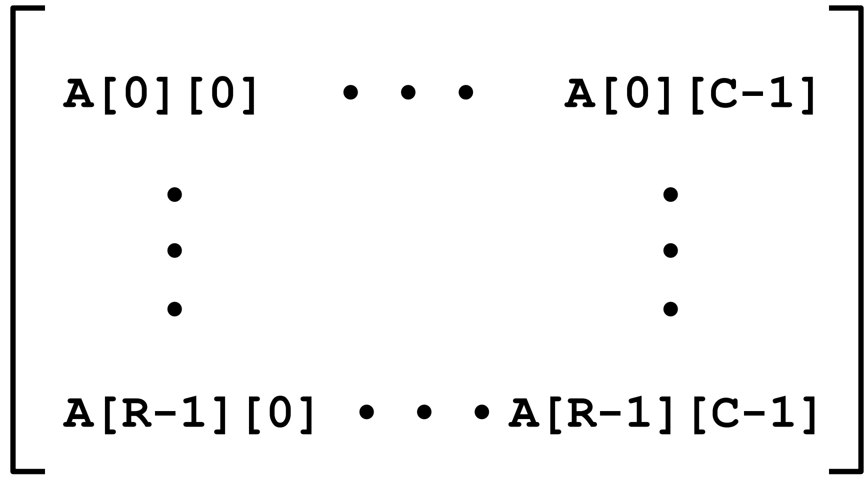
Multidimensional (Nested) Arrays

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

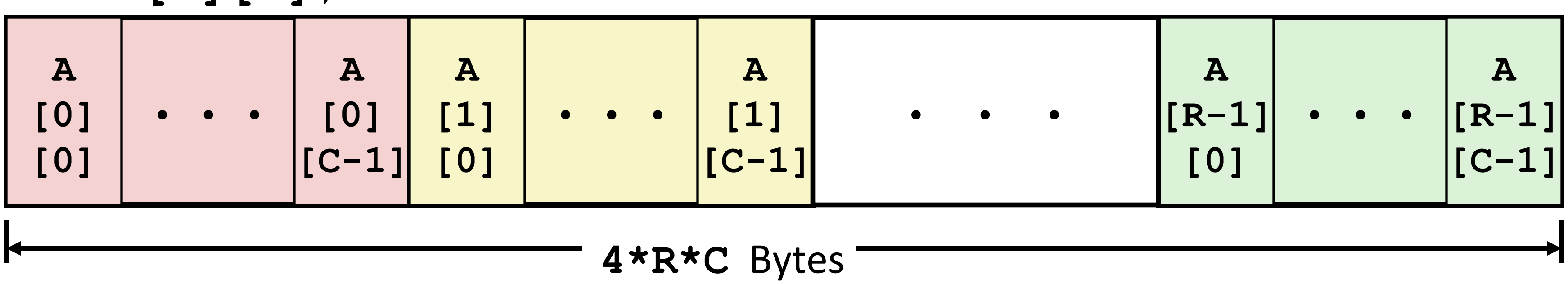
$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

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



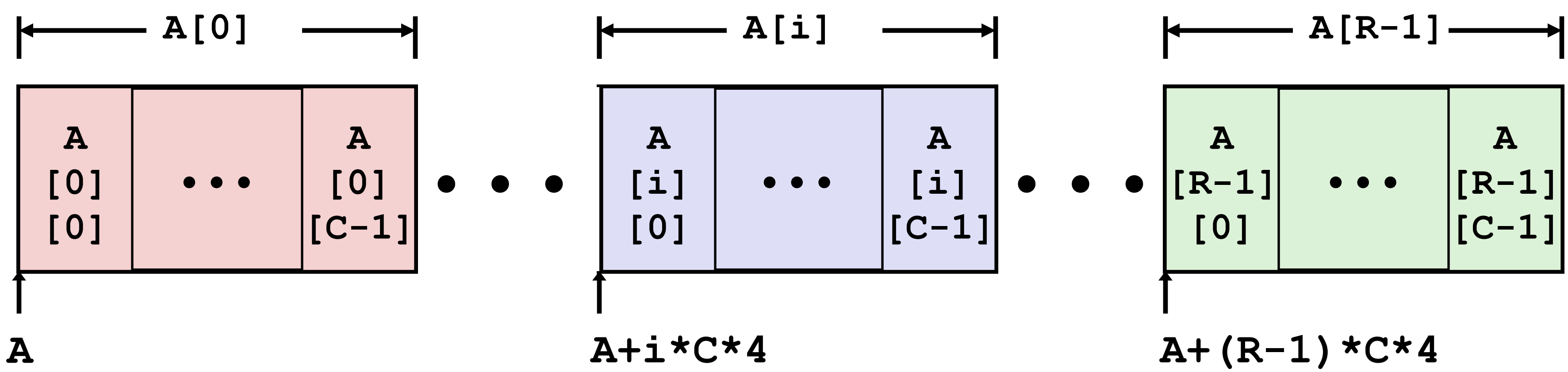

```
int A[R][C];
```



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];
}
```

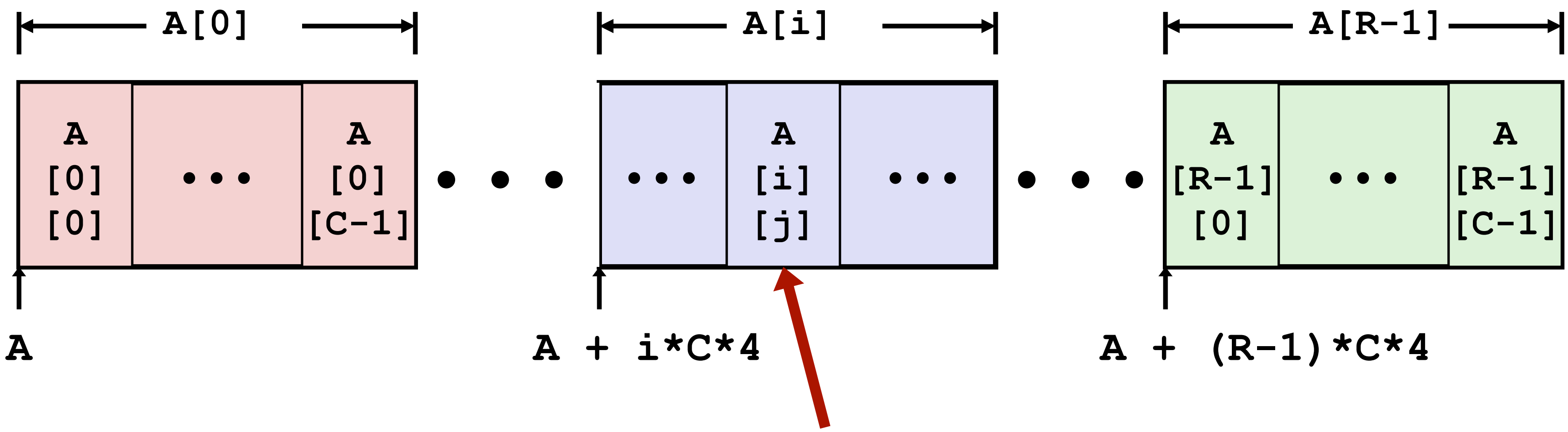
```
#define PCOUNT 4
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
  # %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

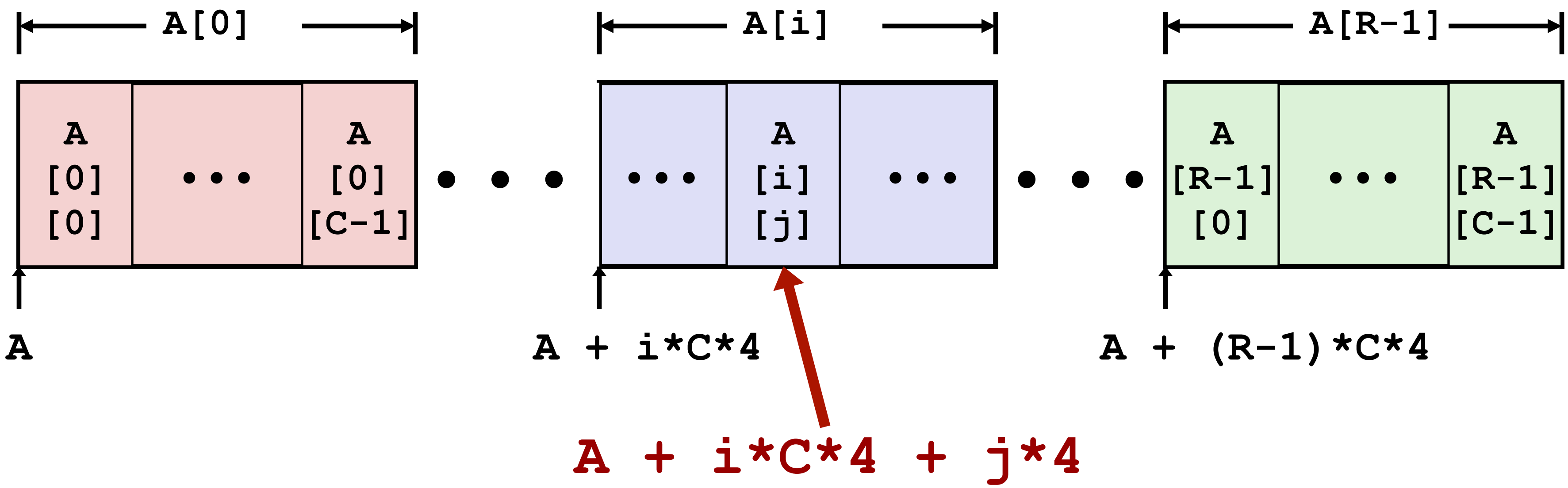
```
int A[R][C];
```



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];
}
```

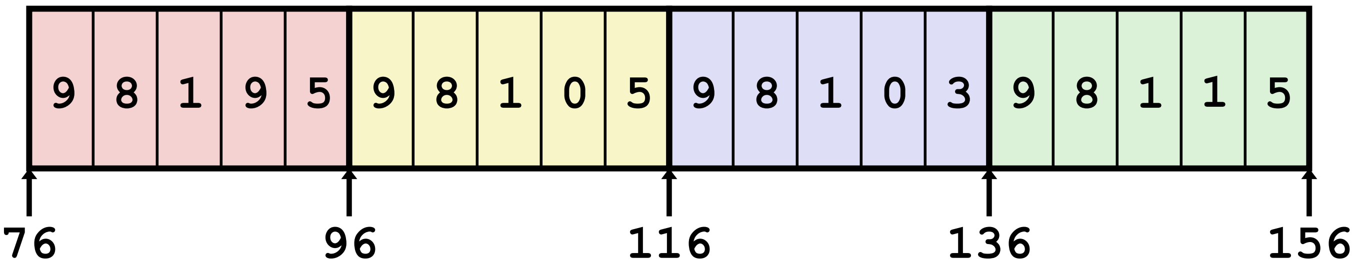
```
zip_dig sea[PCOUNT] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
# %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

`zip_dig sea[4];`



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

Arrays & Other Data Structures

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

Multi-Level Array Example

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

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

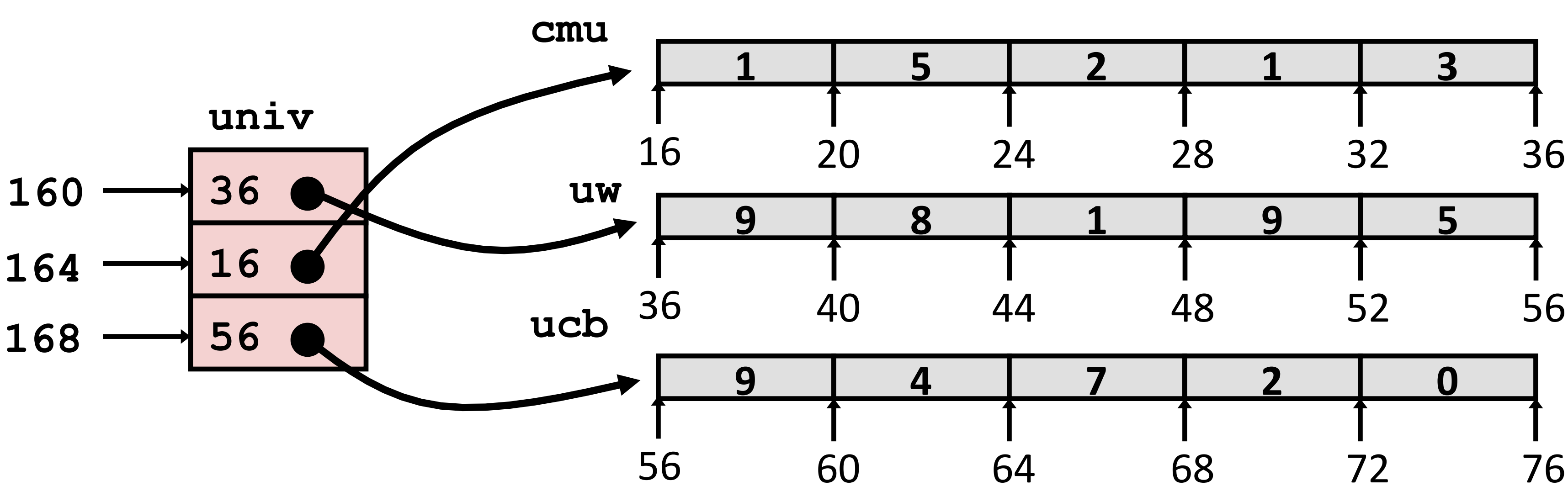
Same thing as a 2D array?

Multi-Level Array Example

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

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

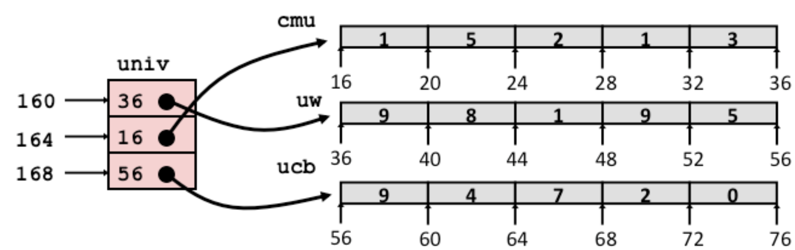
- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`s



Note: this is how Java represents multi-dimensional arrays.

Element Access in Multi-Level Array

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



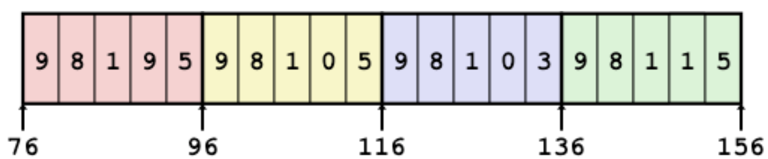

```
  # %ecx = index
  # %eax = dig
  leal 0(,%ecx,4),%edx     # 4*index
  movl univ(%edx),%edx     # Mem[univ+4*index]
  movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```

- **Computation (IA32)**
  - Element access **`Mem[Mem[univ+4*index]+4*dig]`**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

Array Element Accesses

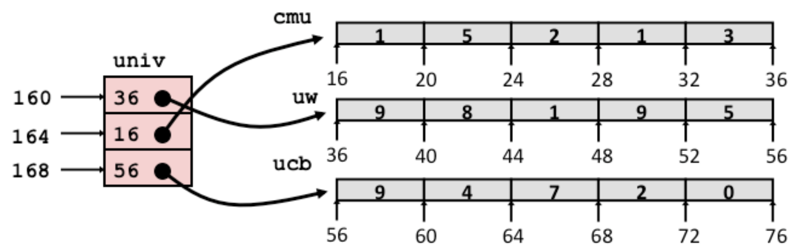
Nested array

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



Multi-level array

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

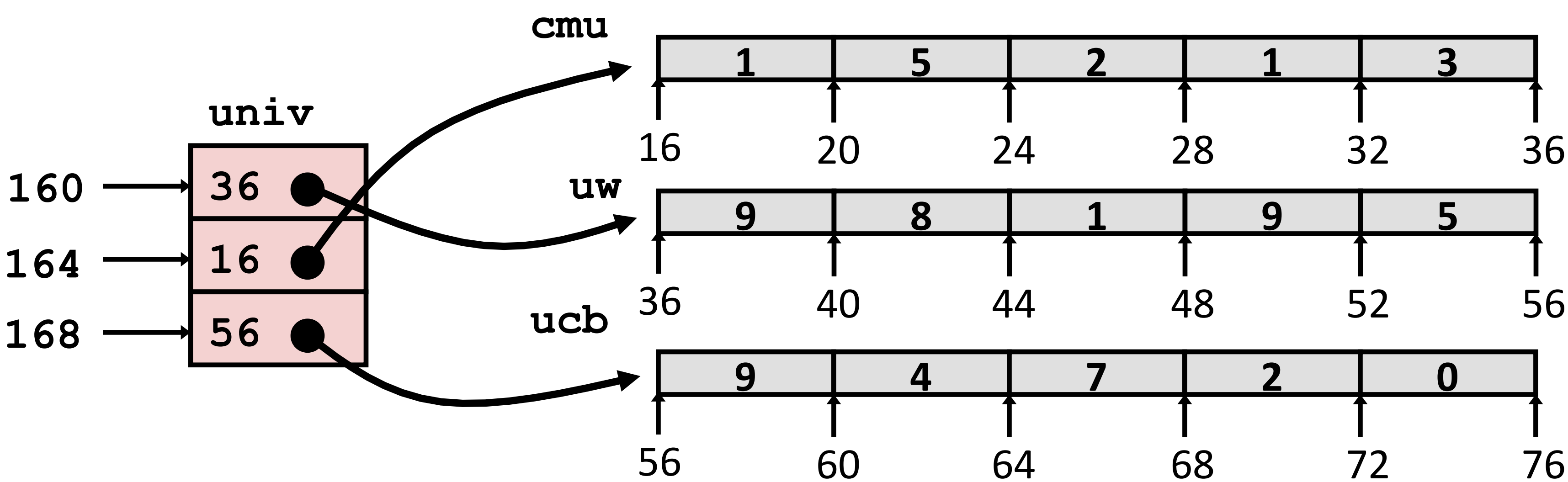


Access looks similar, but it isn't:

`Mem[sea+20*index+4*dig]`

`Mem[Mem[univ+4*index]+4*dig]`

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	`56+4*3 = 68`	2	Yes
`univ[1][5]`	`16+4*5 = 36`	9	No
`univ[2][-1]`	`56+4*-1 = 52`	5	No
`univ[3][-1]`	`??`	??	No
`univ[1][12]`	`16+4*12 = 64`	7	No

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

Arrays in C

- Contiguous allocations of memory
- No bounds checking
- Can usually be treated like a pointer to first element (elements are offset from start of array)
- Nested (multi-dimensional) arrays are contiguous in memory (row-major order)
- Multi-level arrays are not contiguous (pointers used between levels)

Arrays & Other Data Structures

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

Structures

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Structures

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout

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

Structures

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

- **Accessing Structure Member**
  - Given an instance of the struct, we can use the `.` operator, just like Java:
    - `struct rec r1;   r1.i = val;`
  - What if we have a *pointer* to a struct: `struct rec *r = &r1;`
    - Using `*` and `.` operators: `(*r).i = val;`
    - Or, use `->` operator for short: `r->i  = val;`
  - Pointer indicates first byte of structure; access members with offsets

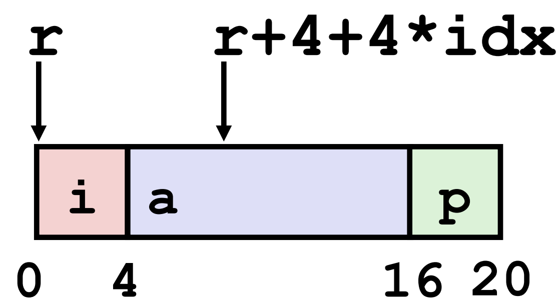
```
void
set_i(struct rec *r,
      int val)
{
     r->i = val;
}
```

IA32 Assembly

```
 # %eax = val
 # %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Generating Pointer to Structure Member

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time

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

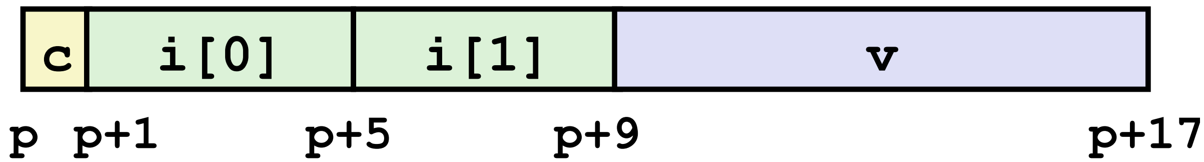
```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax   # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Arrays & Other Data Structures

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

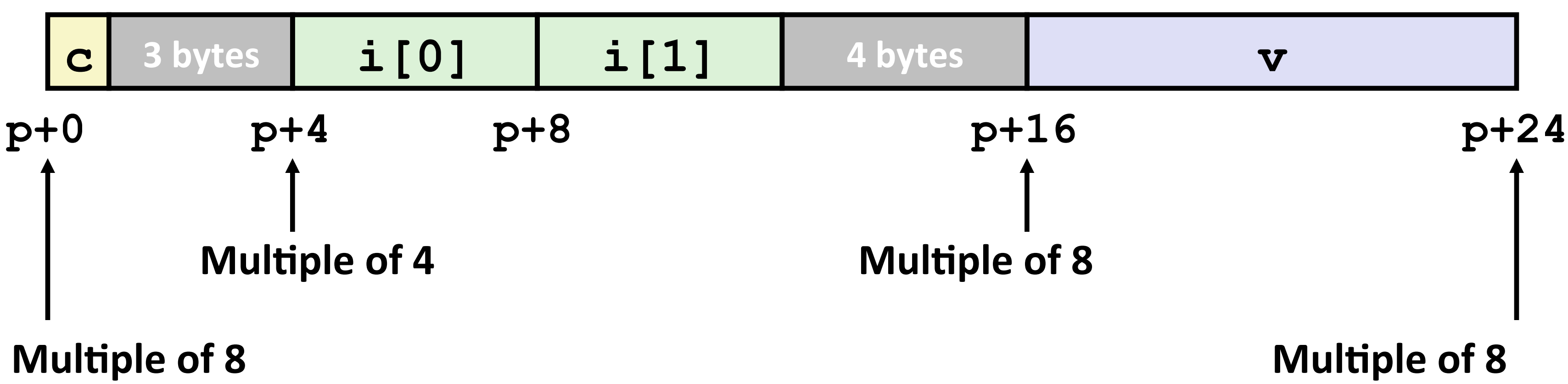
Structures & Alignment

- **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



Alignment Principles

- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
- **Aligned data is required on some machines; it is *advised* on IA32**
  - Treated differently by IA32 Linux, x86-64 Linux, and Windows!
- **What is the motivation for alignment?**

Alignment Principles

- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
- **Aligned data is required on some machines; it is *advised* on IA32**
  - Treated differently by IA32 Linux, x86-64 Linux, and Windows!
- **Motivation for Aligning Data**
  - Physical memory is accessed by aligned chunks of 4 or 8 bytes (system-dependent)
    - Inefficient to load or store datum that spans quad word boundaries
  - Also, virtual memory is very tricky when datum spans two pages (later…)
- **Compiler**
  - Inserts padding in structure to ensure correct alignment of fields
  - `sizeof()` should be used to get true size of structs

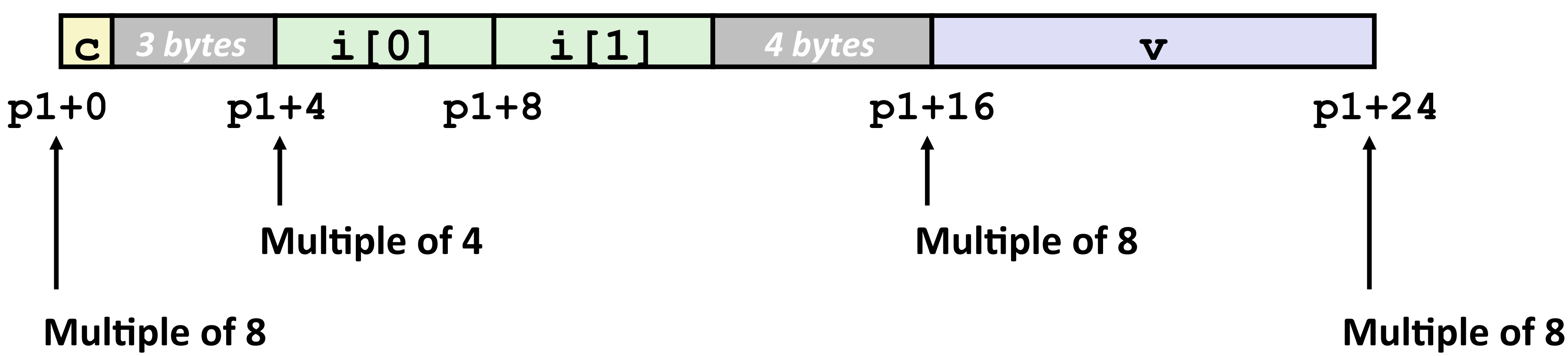
Specific Cases of Alignment (IA32)

- **1 byte: char, …**
  - no restrictions on address
- **2 bytes: short, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: int, float, char *, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: double, …**
  - Windows (and most other OSs & instruction sets): lowest 3 bits 000_2
  - Linux: lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type
- **12 bytes: long double**
  - Windows, Linux: lowest 2 bits of address must be 00_2

Satisfying Alignment with Structures

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

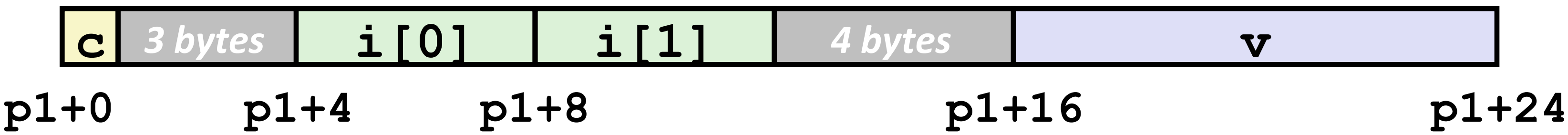
- **Within structure:**
  - Must satisfy every member's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K
- **Example (under Windows or x86-64):** K = ?
  - K = 8, due to **double** member



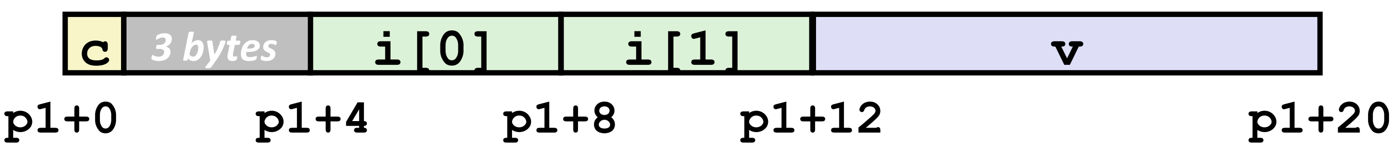
Different Alignment Conventions

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

- **IA32 Windows or x86-64:**
  - K = 8, due to **`double`** member



- **IA32 Linux:** K = ?
  - K = 4; **`double`** aligned like a 4-byte data type



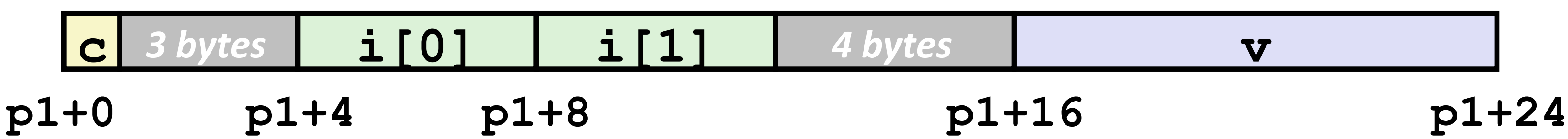
Saving Space

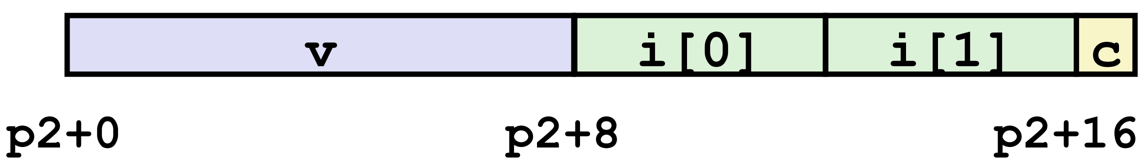
- **Put large data types first:**

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p2;
```

- **Effect (example x86-64, both have K=8)**



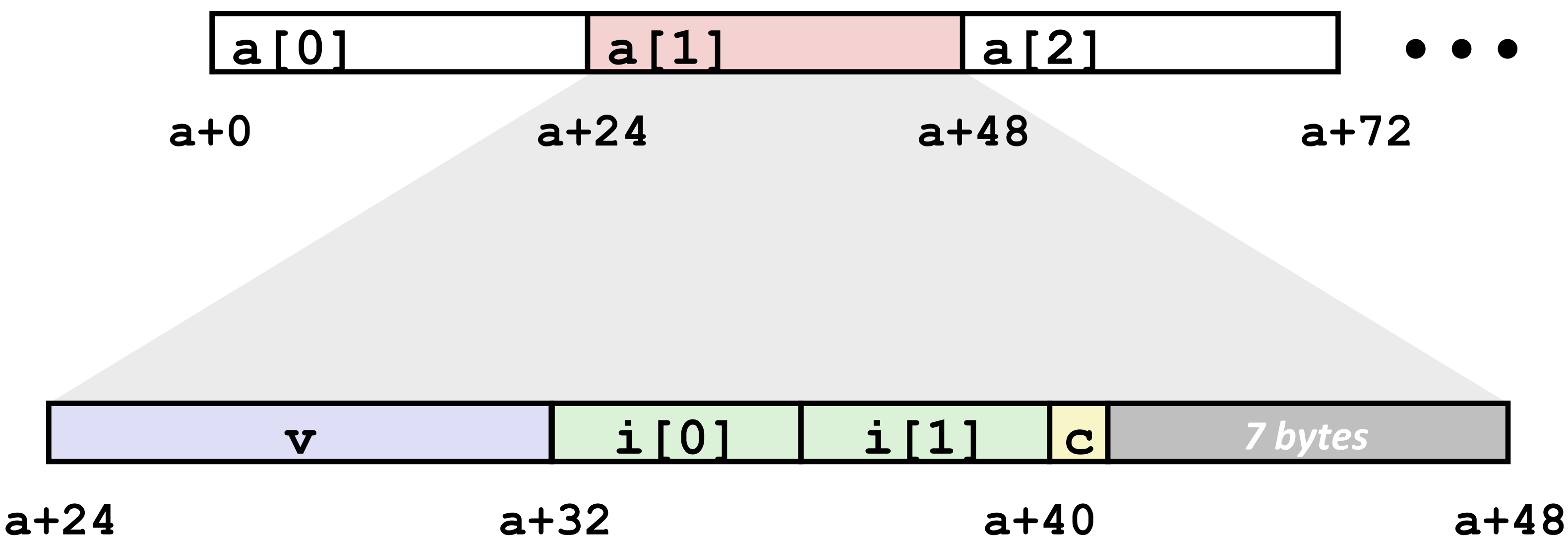


Unfortunately, doesn't satisfy requirement that struct's *total size* is a multiple of K

Arrays of Structures

- **Satisfy alignment requirement for every element**

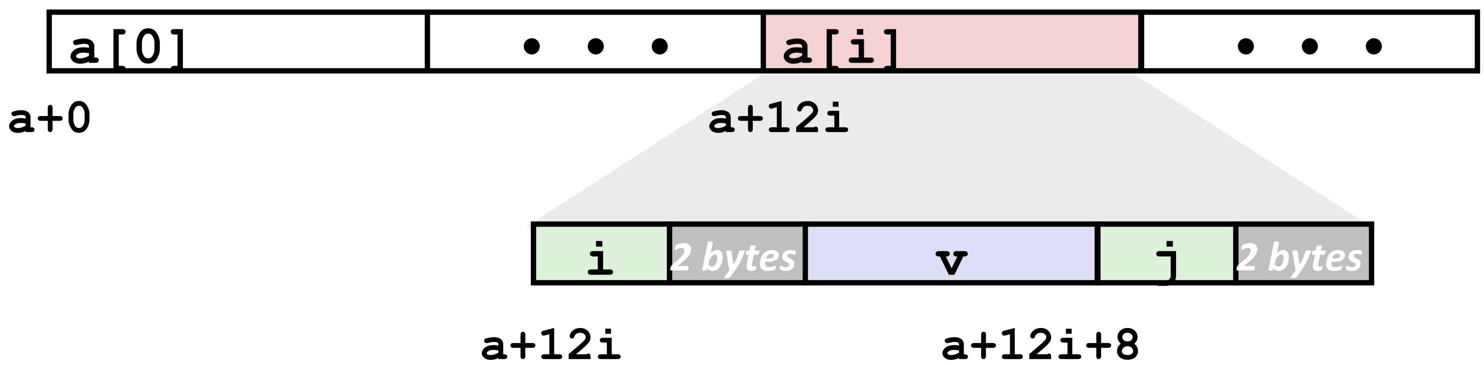
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

- Compute array offset 12i (`sizeof(S3)`)
- Element `j` is at offset 8 within structure
- Since a is static array, assembler gives offset a+8




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

```
# %eax = idx
leal (%eax,%eax,2),%eax   # 3*idx
movswl a+8(,%eax,4),%eax # a+12*idx+8
```