X86 Data Arrays and Structures

Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

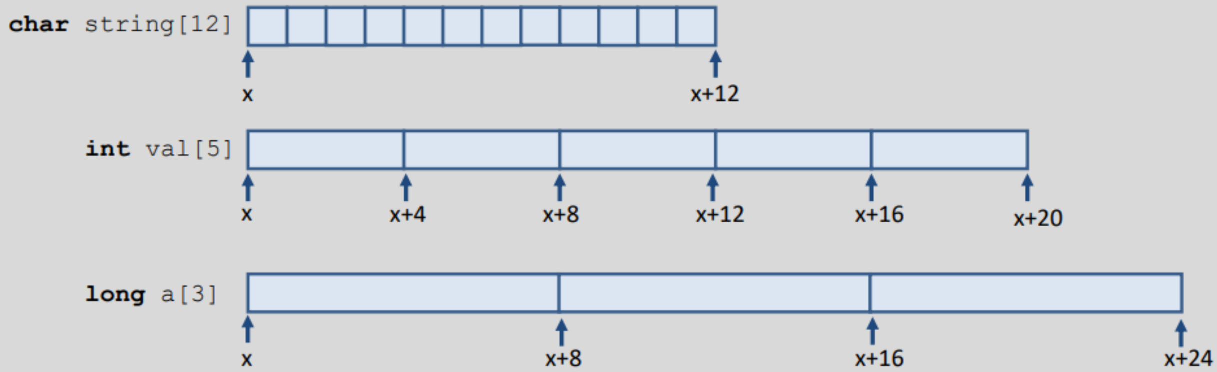
Basic data types

- **Integral**
  - Stored and operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used
  - Example:
    - **byte** (size: 1 bytes, appendix: `b`, in C/Java: `char`)
    - **word** (size: 2 bytes, appendix: `w`, in C/Java: `short`)
    - **double word** (size: 4 bytes, appendix: `l`, in C/Java: `int`)
    - **quad word** (size: 8 bytes, appendix: `q`, in C/Java: `long` (x86-64)
- **Floating point**
  - Stored and operated on floating point registers

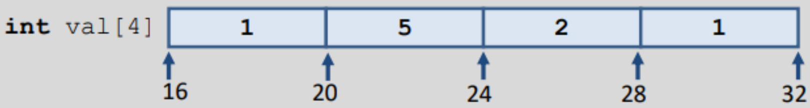
Array allocation

- **Basic Principle**

  T **A[*L*];**
  - Array of data type *T* and length *L*
  - *Contiguously* allocated region of *L* * **sizeof** (*T*) bytes in memory



Array Accessing Example




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

```
get_digit:
    movslq  %esi, %rsi
    movl    (%rdi,%rsi,4), %eax
    ret
```

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

Array Loop Example

```c
void digit_inc(int val[], int len){
  int i;
  for (i = 0; i < len; i++)
    val[i]++;
  return;
}
```

```c
void digit_inc(int val[], int len){
  if (len>0){
    int *ptr = val;
    int *vend = val + len;
    do {
      *ptr = *ptr + 1;
      ptr++;
    } while (ptr != vend);
  }
  return;
}
```

```
digit_inc:
   testl    %esi, %esi
   jle      .L1
   movq     %rdi, %rax
   leal     -1(%rsi), %edx
   leaq     4(%rdi,%rdx,4), %rdx
.L3:
   addl     $1,(%rax)
   addq     $4,%rax
   cmpq     %rdx, %rax
   jne      .L3
.L1:
   rep ret
```

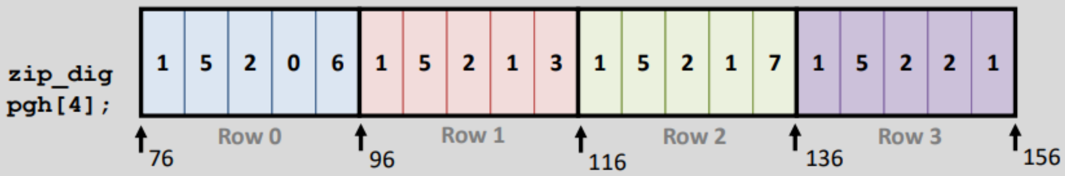
Nested Arrays

Nested Arrays

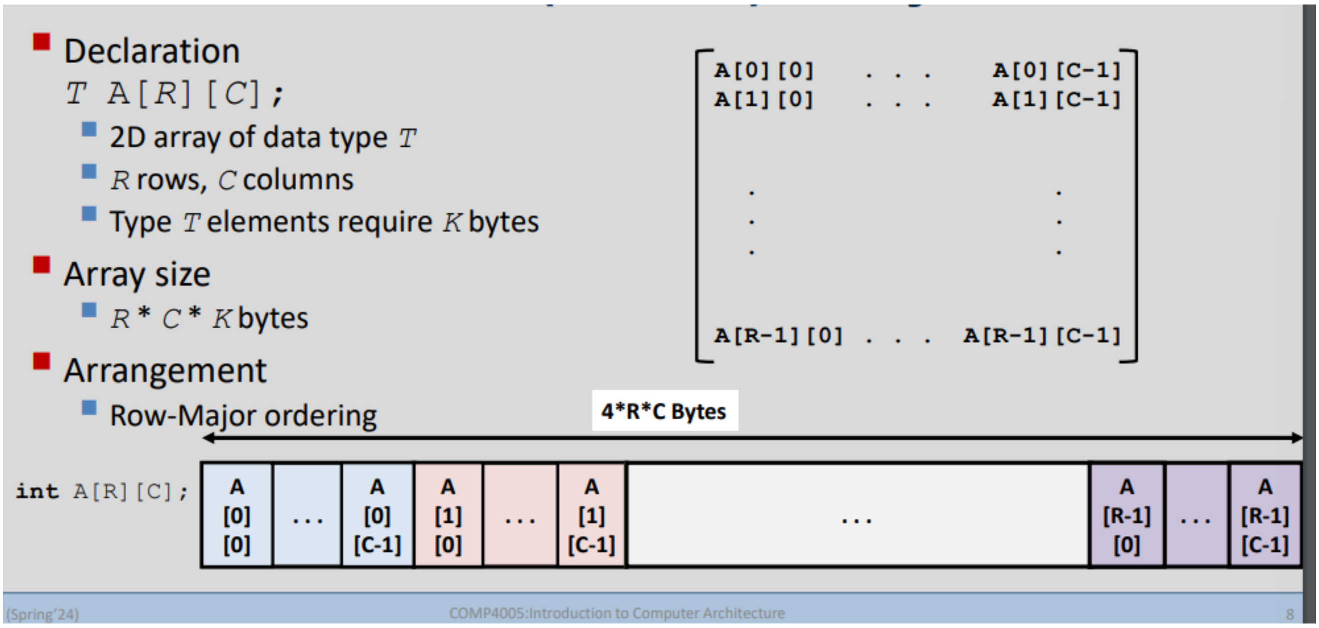
```c
typedef int zip_dig[5];
zip_dig pgh[4] =
{{1, 5, 2, 0, 6},
 {1, 5, 2, 1, 3},
 {1, 5, 2, 1, 7},
 {1, 5, 2, 2, 1}};
```

- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
- Variable **pgh**: array of 4 elements, allocated contiguously
- Each element is an array of 5 **int**'s, allocated contiguously
- "**Row-Major**" ordering of all elements guaranteed

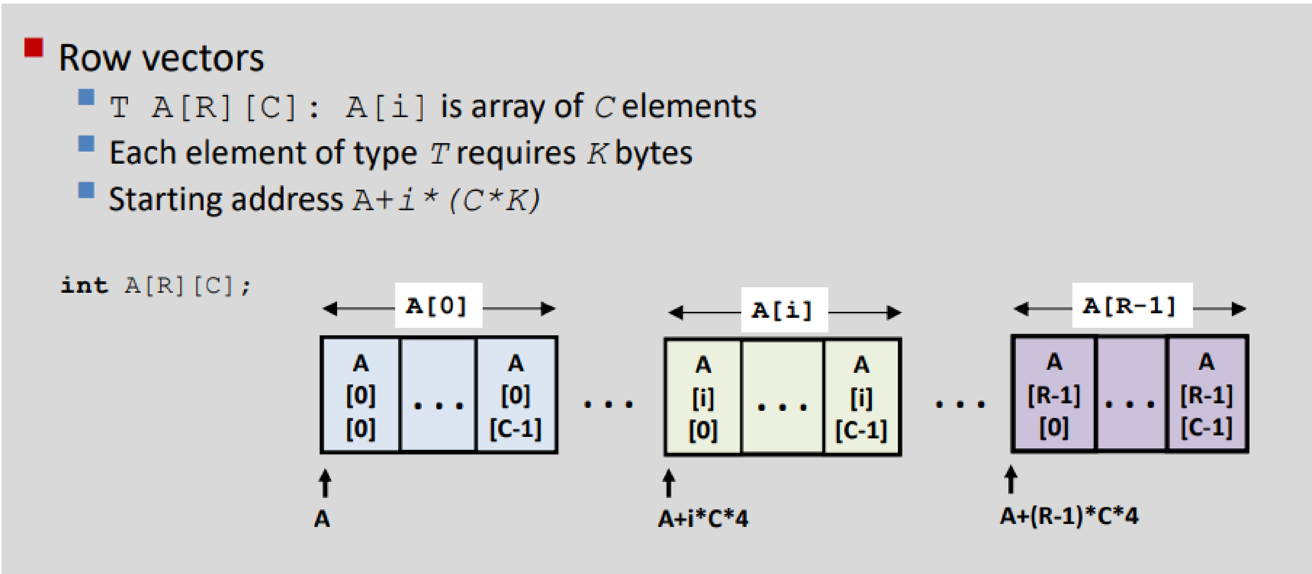
Recall, `T A[L]` is an array with `L` elements of type `T`.



Multidimensional (nested) Arrays



Nested Array Row Access



Nested Array Row Access Mode

```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
zip_dig pgh[4] =
{{1, 5, 2, 0, 6},
 {1, 5, 2, 1, 3},
 {1, 5, 2, 1, 7},
 {1, 5, 2, 2, 1}};
```

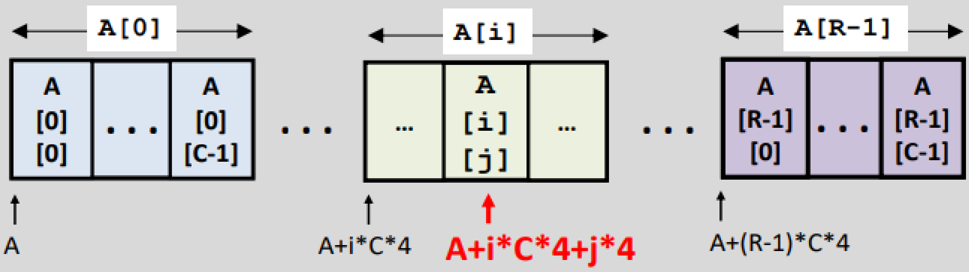
```
leaq    (%rdi,%rdi,4), %rax   # 5 * index
leaq    pgh(,%rax,4), %rax    # pgh + (20 * index)
```

- Row vector:
  - `pgh[index]` is array of 5 `int`'s,
  - starting address `pgh+20*index`
- x86_64 code:
  - Computes and returns address
  - Compute as `pgh+4*(index+4*index)`

Nested Array Element 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_pgh_digit(int index, int digit ){
  return pgh[index][digit];
}
```

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

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

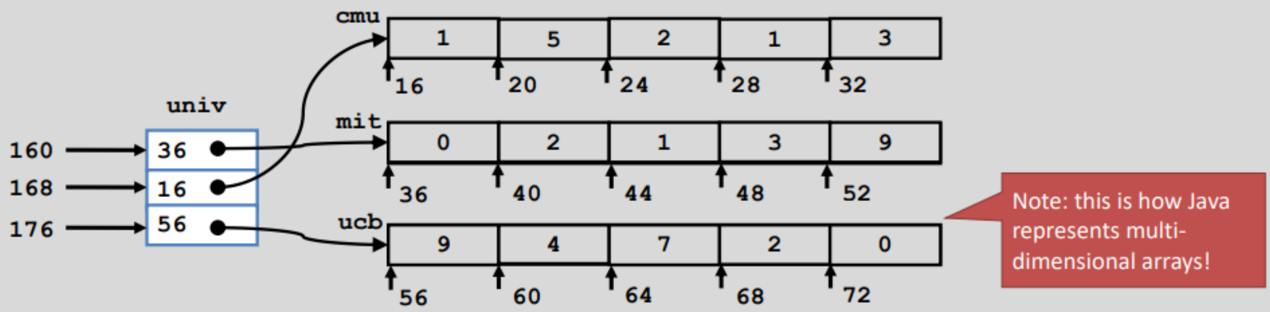
Multi-Level Array Example

Multi-Level Array Declaration(s):

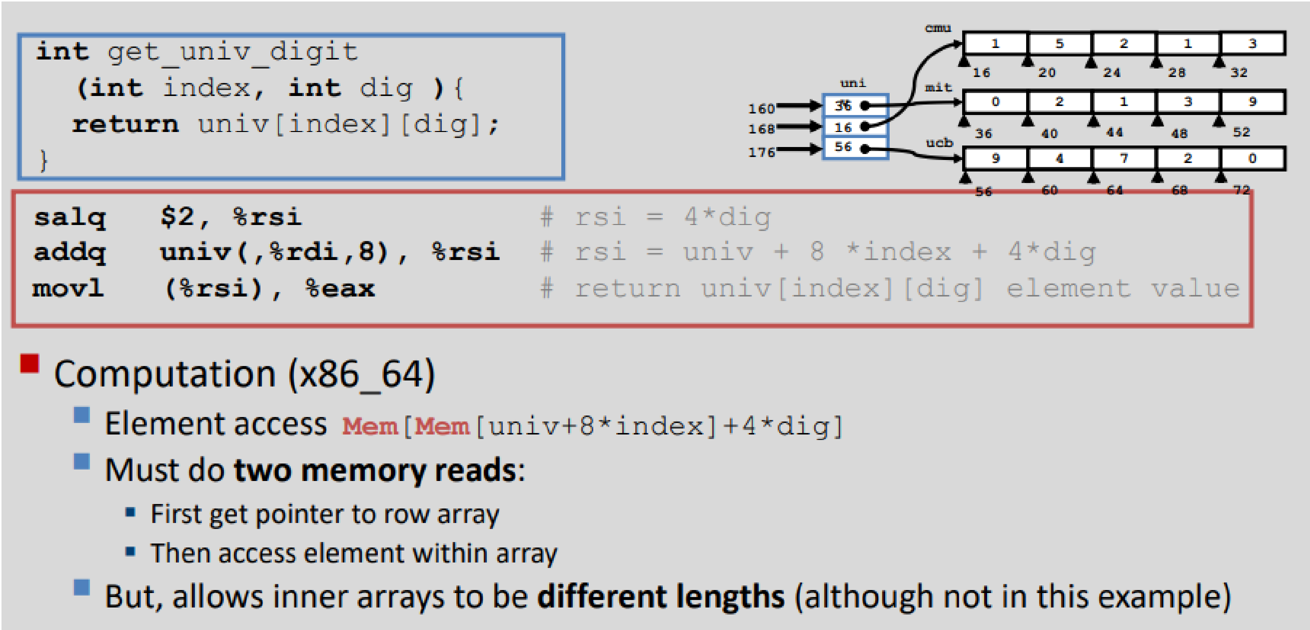
```
int cmu[5] = {1, 5, 2, 1, 3};
int mit[5] = {0, 2, 1, 3, 9};
int ucb[5] = {9, 4, 7, 2, 0};
```

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

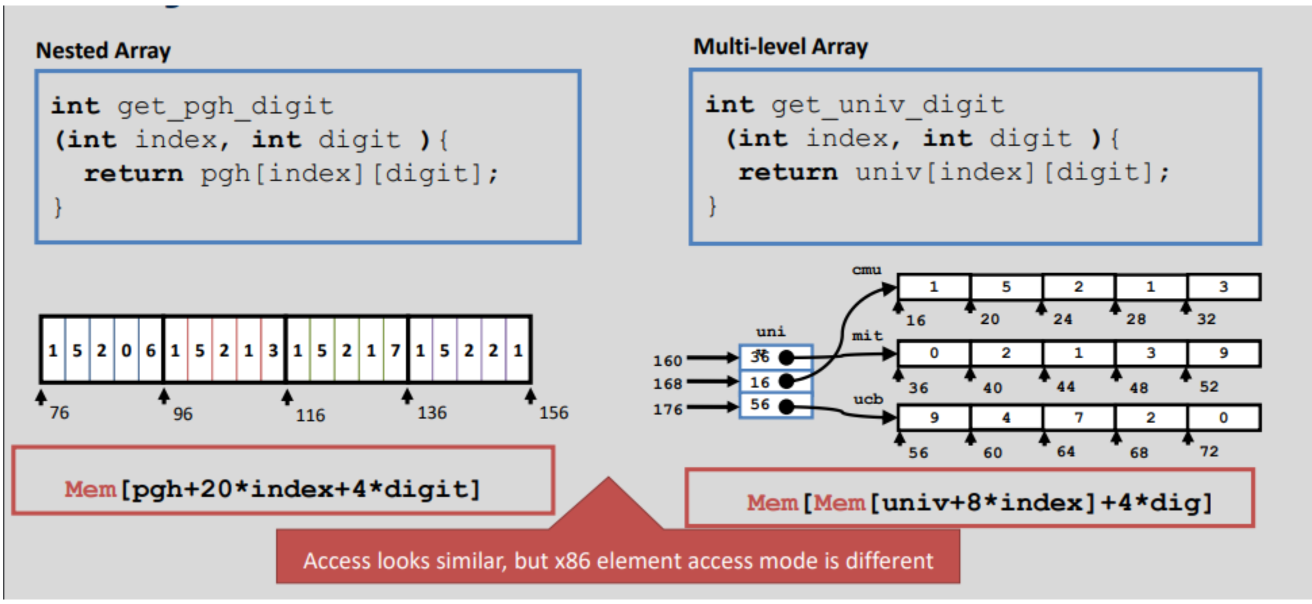
- Variable `univ` denotes array of 3 elements
- Each element is a pointer: 8 bytes
- Each pointer points to array of `int`s



Element Access in Multi-Level Array



Array Element Accesses



Data Structures in Assembly

Structures

- Alignment

Structs in C

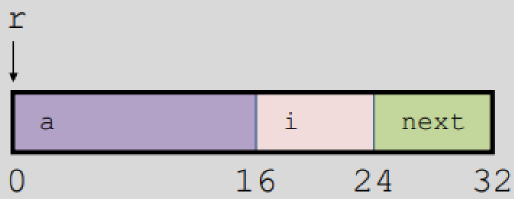
- Way of defining compound data types
- A structured group of variables, possibly including other `structs`

```
struct song{
  int lengthInSeconds;
  int yearRecorded;
};

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

struct song song2;
song2.lengthInSeconds =  214;
song2.yearRecorded     = 1988;
```

- Given a struct instance, access member using the . operator
- In assembly: pointer holds address of the first byte. Access elements with offsets.

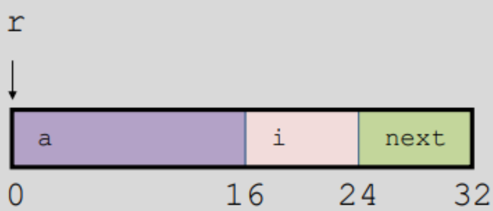
Structure Representation

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



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

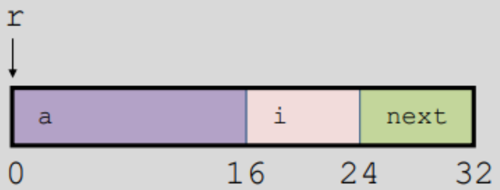
```
typedef struct rec {
  int a[4];
  long i;
  struct rec *next;
} *rec_p;
```



- **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 programs has no understanding of the structures in the source code

Accessing a Structure Member

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



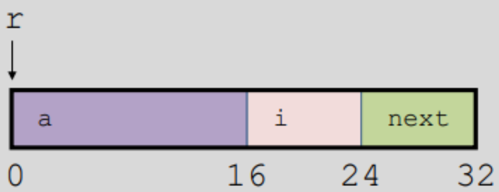
- Compiler knows the ***offset*** of each member within a struct
- Compute as
  - `* (r+offset)`
  - Referring to absolute offset, so no pointer arithmetic

```
long get_i(rec_p r)
{
  return r->i;
}
```

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

Exercise: Pointer to Structure Member

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




```
long get_i(rec_p r){
  return r->i;
}
```

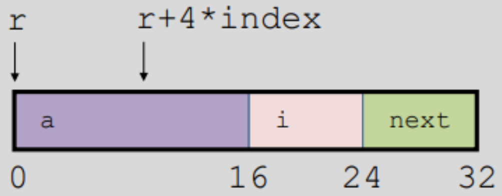
```
# r in %rdi
movq    16(%rdi) , %rax
ret
```

```
rec_p get_next(rec_p r){
  return r->next;
}
```

```
# r in %rdi
movq    24(%rdi) , %rax
ret
```

Generating Pointer to Array Element

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



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

```
int find_addr_of_array_elem
  (rec_p r, long index){
   return r->a[index];
}
```

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

Memory Alignment in x86-64

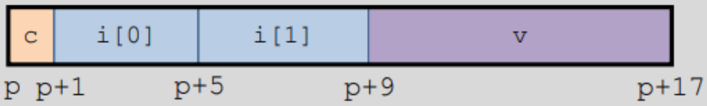
- For good memory system performance, Intel recommends data to be aligned
  - However, the x86-64 hardware will work correctly regardless of alignment of data
- *Aligned* means that any primitive object of *K* bytes must have an address that is multiple of *K*

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

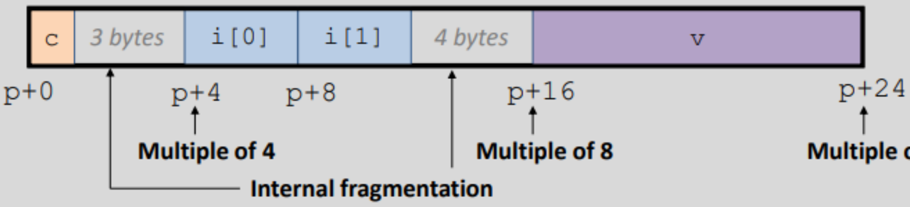
- **Aligned Data**
  - Primitive data type requires *K* bytes
  - Address must be multiple of *K*
  - Required on some machines; advised on x86-64
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store value that spans word boundaries

Structures and Alignment



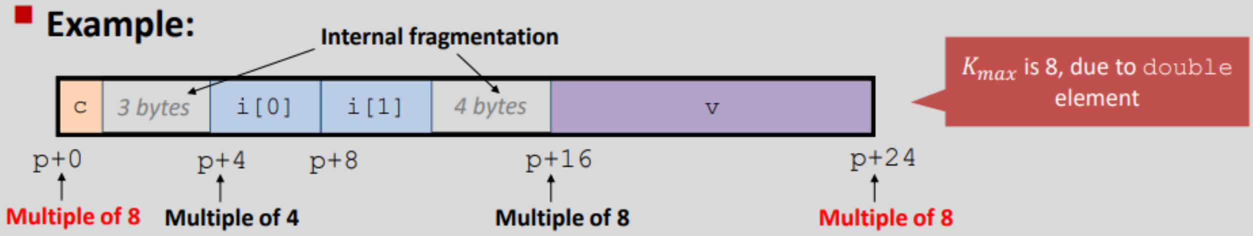

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

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



Satisfying Alignment with Structures

- **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
  - Address of structure and structure length must be multiples of K_{max}
- **Example:**

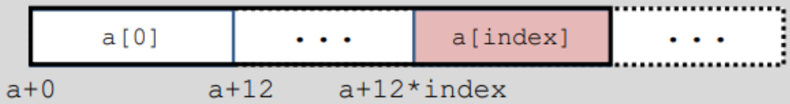


Alignment of Structs

- **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`

Accessing Array's Struct 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`

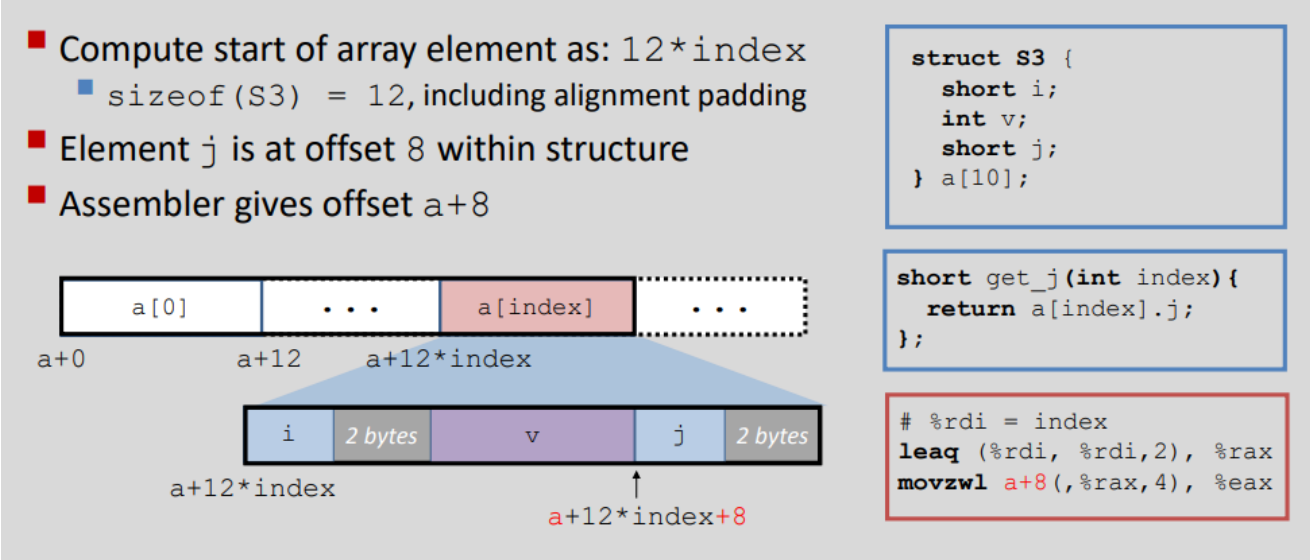



```
struct S3 {
  short i;
  int v;
  short j;
} a[10];
```

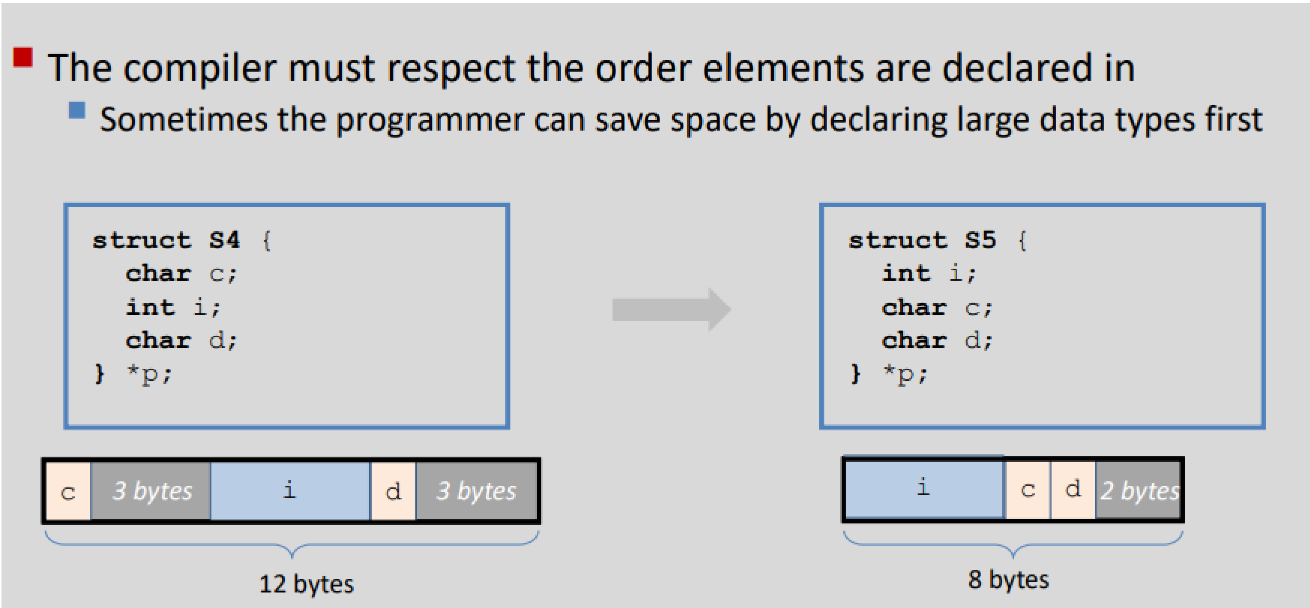
```
short get_j(int index){
  return a[index].j;
};
```

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

Accessing Array's Struct Elements



How the Programmer Can Save Space



Summary

- **Arrays**
  - Contiguous allocations of memory
  - Can usually be treated like a pointer to first element
  - Nested arrays
    - all levels in one contiguous block of memory
  - Multi-Level 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
  - Padding in middle and at end to satisfy alignment