Imperial College London

113: Architecture

Spring 2018

Lecture: X86 Data: Arrays and Structures

Instructor: Dr. Jana Giceva

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures

Basic data types

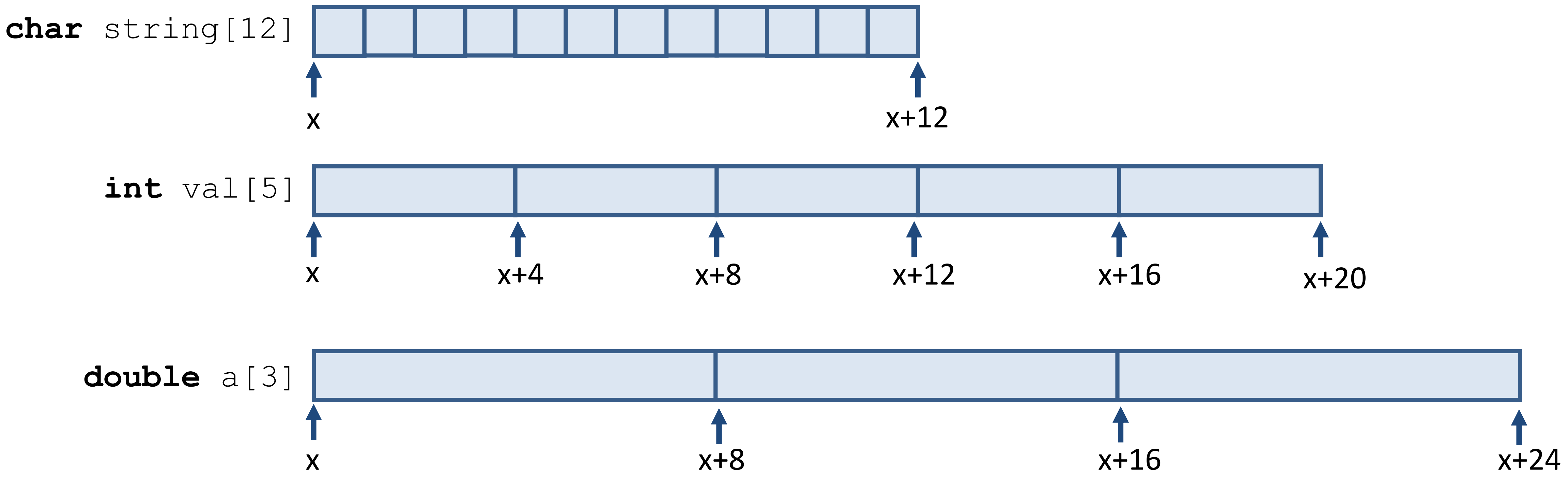
- **Integral**
  - Stored and operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used
  - Example:
    - **byte** (size: 1 bytes, appendix: `b`, in C/Java: `char`)
    - **word** (size: 2 bytes, appendix: `w`, in C/Java: `short`)
    - **double word** (size: 4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

Array allocation

- **Basic Principle**

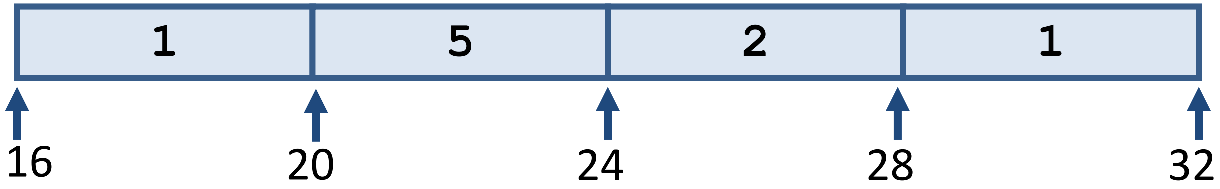
  T **A[***L***];**

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



Array Accessing Example

`int val[5]`




```
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

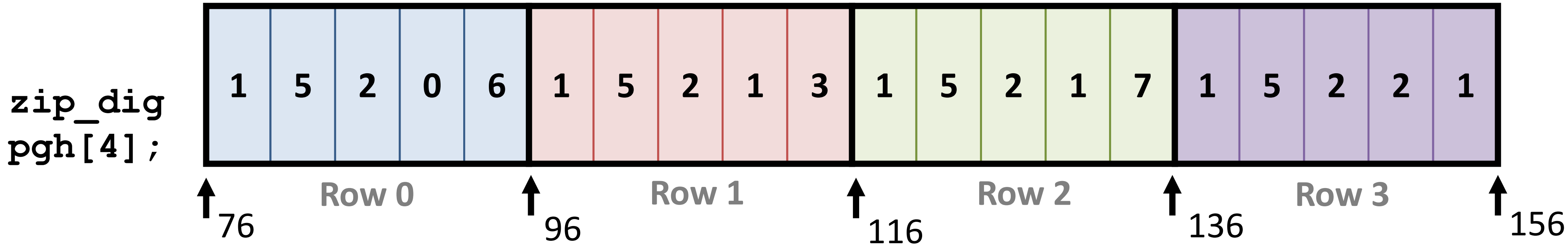
```
void digit_inc(int val[], int len){
  int i;
  for (i = 0; i < len; i++)
    val[i]++;
  return;
}
```

```
void digit_inc(int val[], int len){
  if (!len){
    int *vend = val + len-1;
    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

```
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

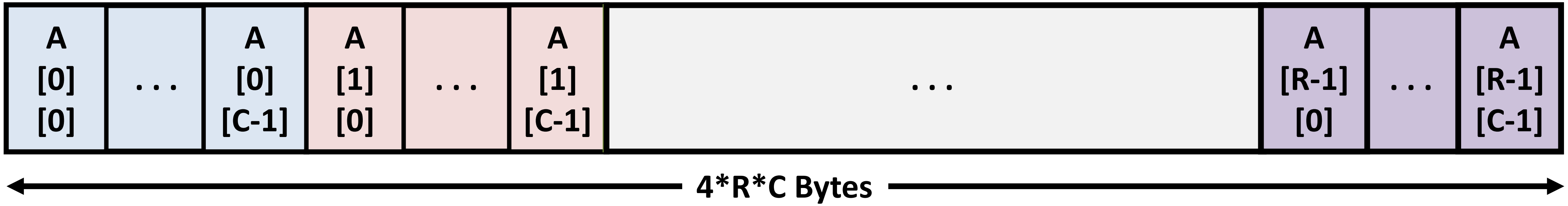
Multidimensional (nested) Arrays

- Declaration

  T `A[`R`][`C`];`
  - 2D array of data type T
  - R rows, C columns
  - Type T elements require K bytes
- Array size
  - R * C * K bytes
- Arrangement
  - Row-Major ordering

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

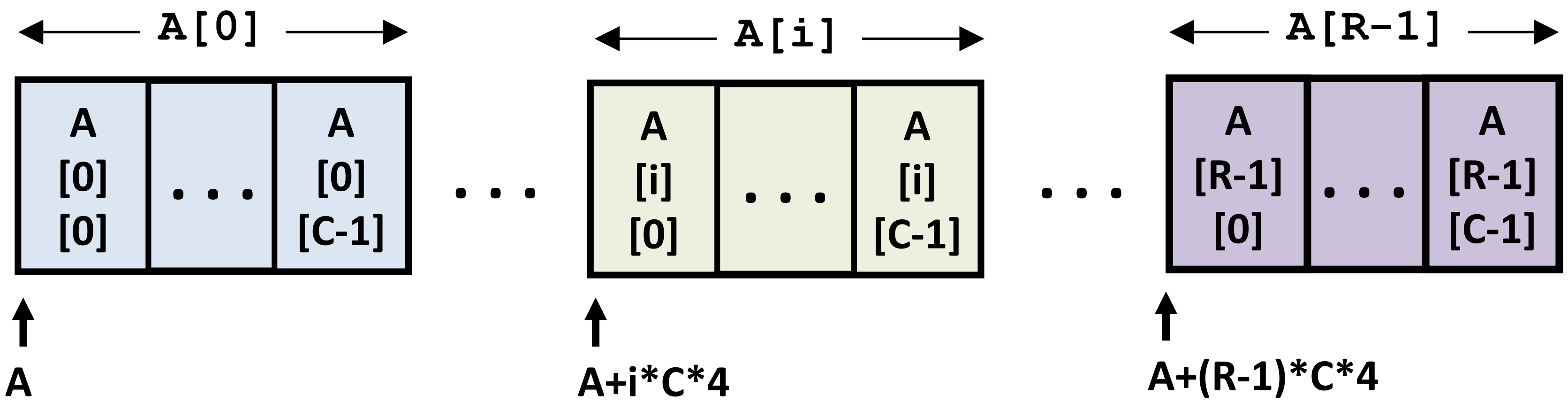
```
int A[R][C];
```



Nested Array Row Access

- Row vectors
  - `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 Mode

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

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

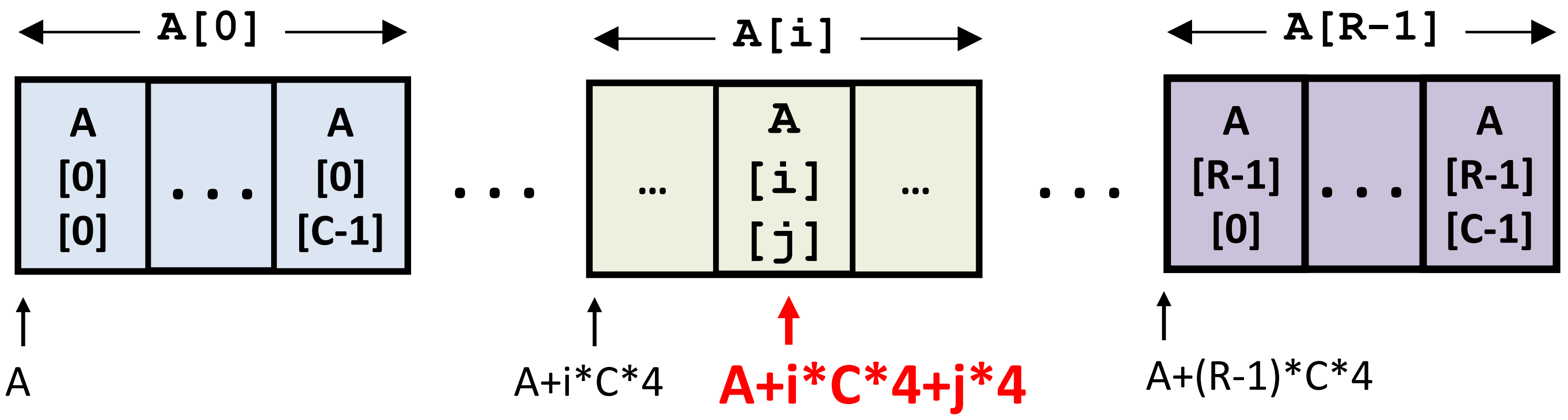
```
    leaq    (%rdi,%rdi,4), %rax    # 5 * index
    leaq    pgh(,%rax,4), %rax     # pgh + (20 * index)
```

- 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

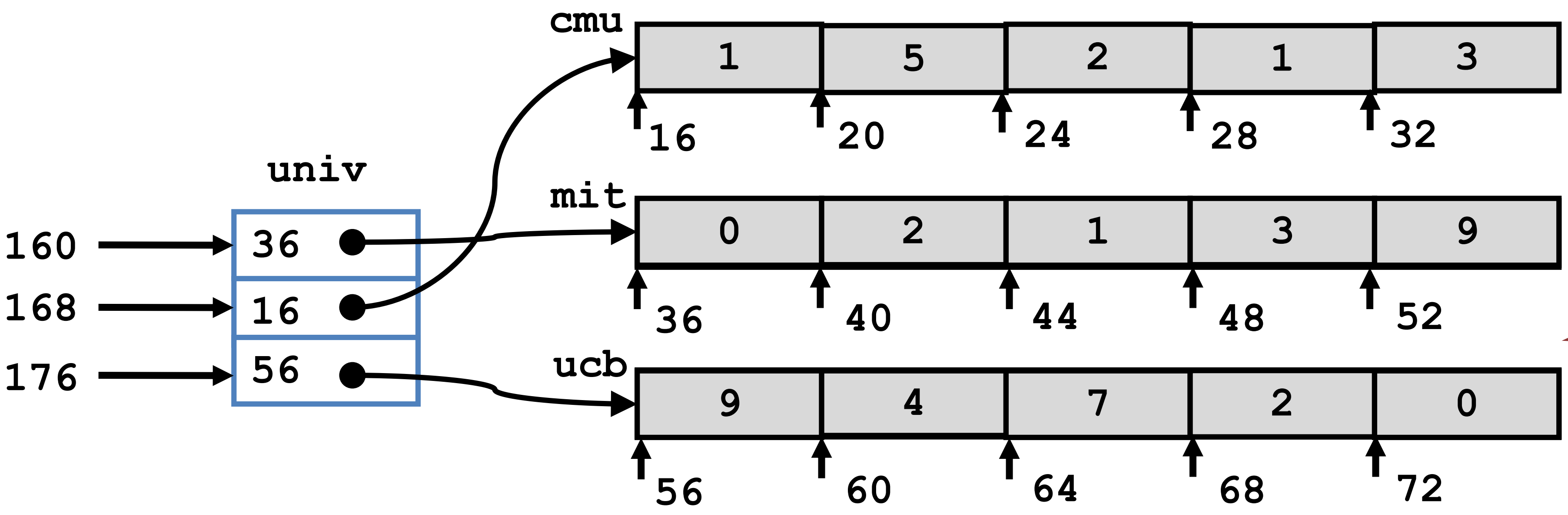
Multi-Level Array Example

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};
```

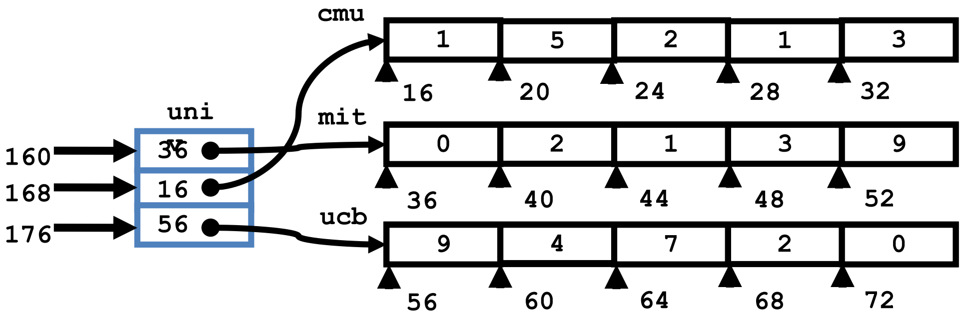
- Variable `univ` denotes array of 3 elements
- Each element is a pointer: 8 bytes
- Each pointer points to array of `int`'s



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];
}
```



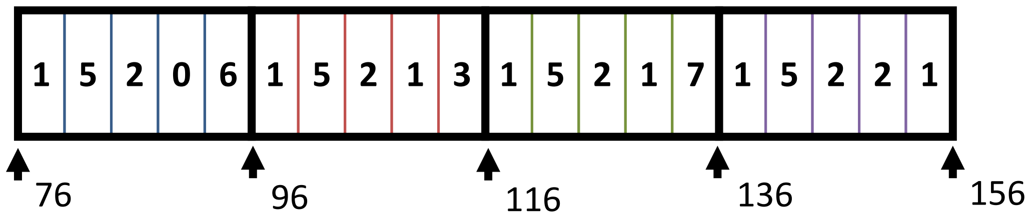

```
salq    $2, %rsi              # rsi = 4*digit
addq    univ(,%rdi,8), %rsi   # p = univ[index] + 4*digit
movl    (%rsi), %eax          # return *p
```

- Computation (x86_64)
  - Element access `Mem[Mem[univ+8*index]+4*dig]`
  - Must do **two memory reads**:
    - First get pointer to row array
    - Then access element within array
  - But, allows inner arrays to be **different lengths** (although not in this example)

Array Element Accesses

Nested Array

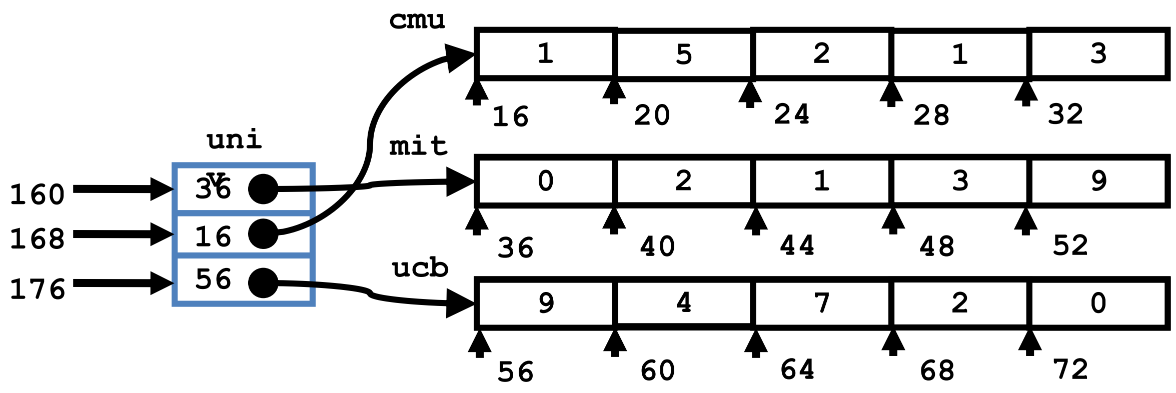
```
int get_pgh_digit
(int index, int digit ){
  return pgh[index][digit];
}
```




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

Multi-level Array

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




```
Mem[Mem[univ+8*index]+4*dig]
```

Access looks similar, but x86 element access mode is different

Data Structures in Assembly

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

Structs in C

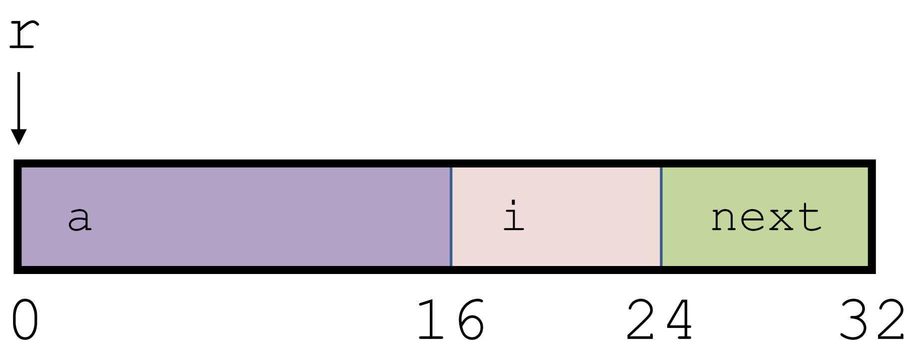
- Way of defining compound data types
- A structured group of variables, possibly including other `structs`

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

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

Song song2;
song2.lengthInSeconds =  214;
song2.yearRecorded    = 1988;
```

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

Structure Representation

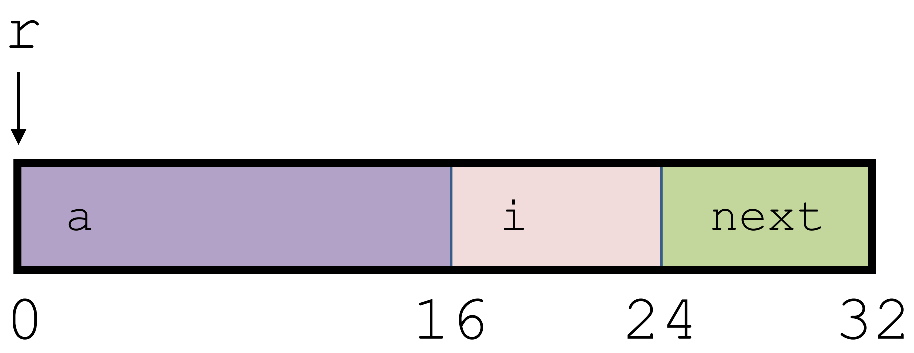
```
typedef rec {
  int a[4];
  long i;
  struct rec *next;
} *r;
```



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

Structure Representation

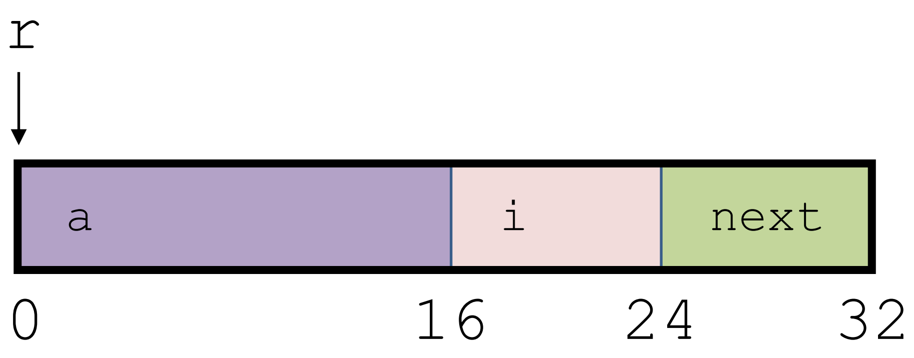
```
typedef rec {
  int a[4];
  long i;
  struct rec *next;
} *r;
```



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

Accessing a Structure Member

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



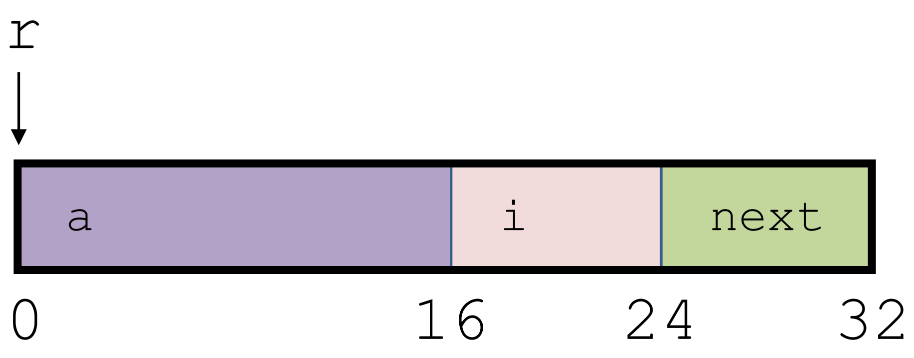
- Compiler knows the ***offset*** of each member within a struct
- Compute as
  - `* (r+offset)`
  - Referring to absolute offset, so no pointer arithmetic

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

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

Exercise: Pointer to Structure Member

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




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

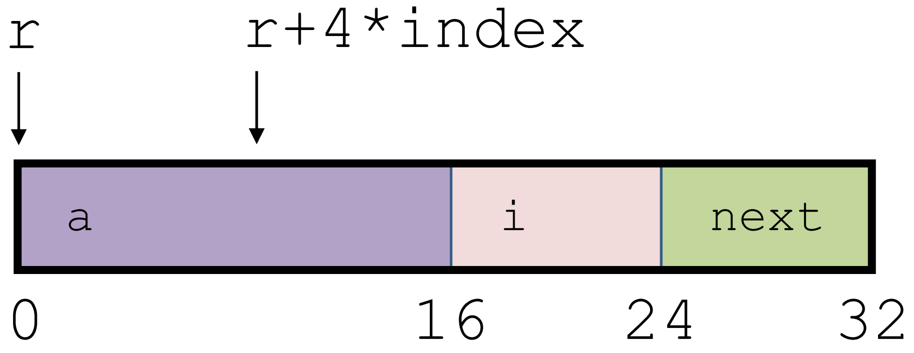
```
# r in %rdi
leaq    16(%rdi), %rax
ret
```

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

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

Generating Pointer to Array Element

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



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

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

```
# r in %rdi, index in %rsi
leaq  (%rdi,%rsi,4) , %rax
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*

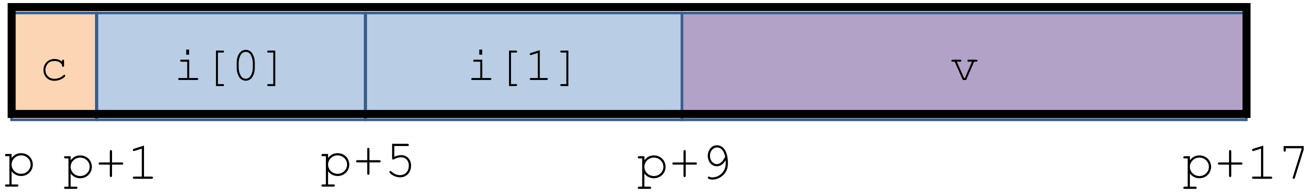
K	Type	Addresses
1	`char`	No restrictions
2	`short`	Lowest bit must be zero: …0_2
4	`int, float`	Lowest 2 bits zero: …00_2
8	`long, double`	Lowest 3 bits zero: …000_2
16	`long double`	Lowest 4 bits zero: …0000_2

Alignment Principles

- **Aligned Data**
  - Primitive data type requires *K* bytes
  - Address must be multiple of *K*
  - Required on some machines; advised on x86-64

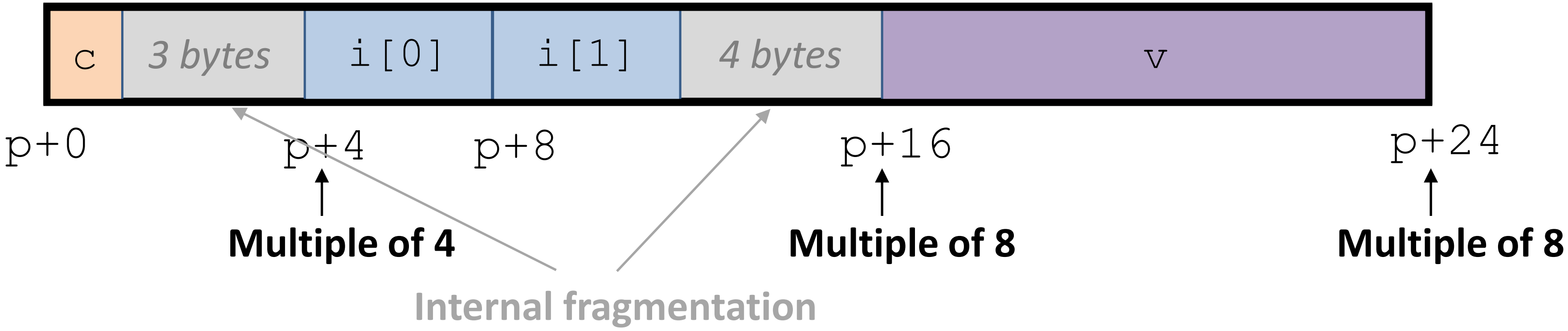
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store value that spans word boundaries

Structures and Alignment



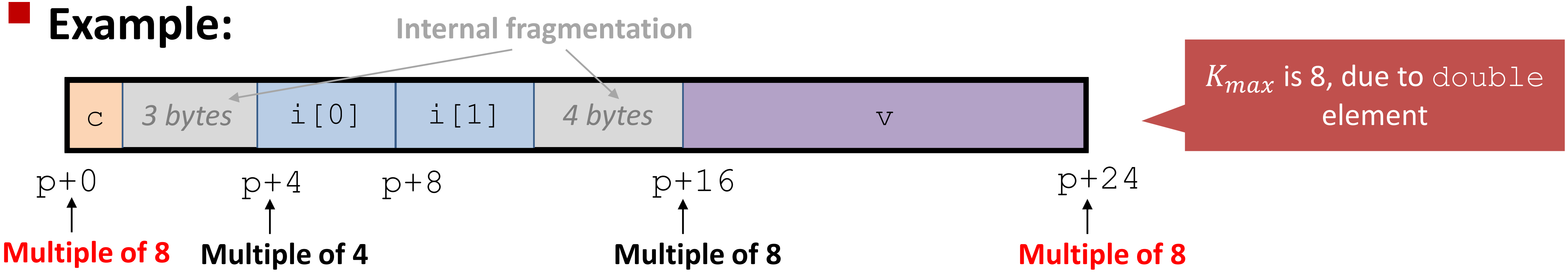

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- **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:**

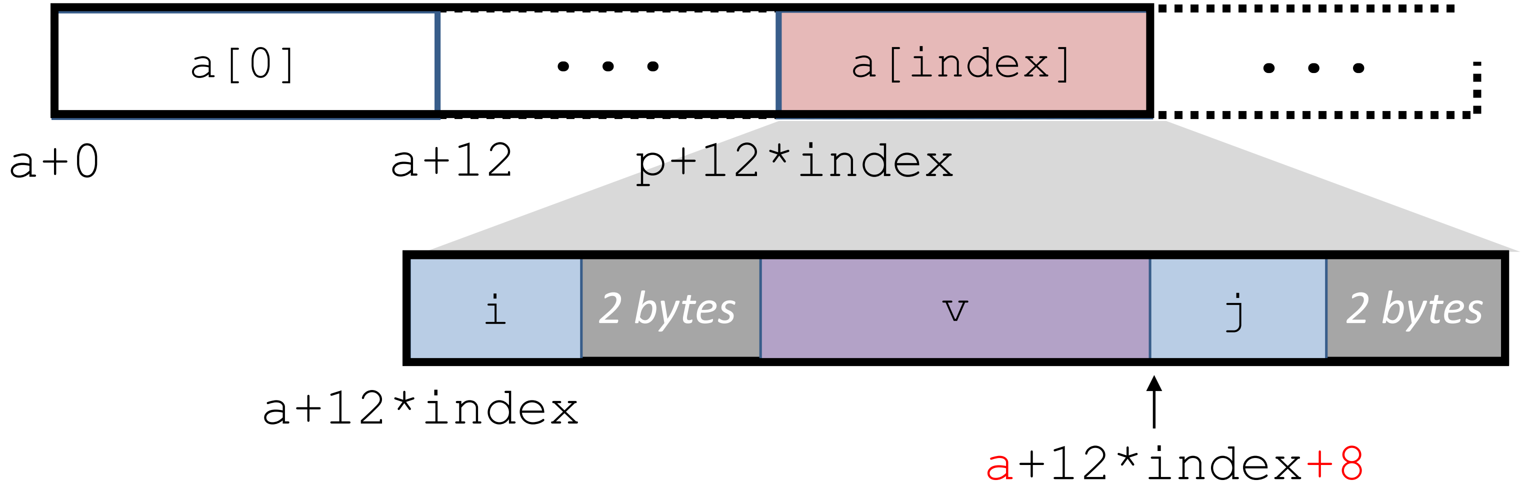


Alignment of Structs

- **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 Elements within Structures

- 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;
};
```

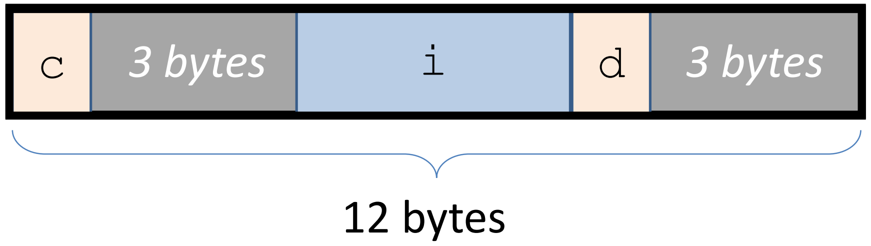
```
# %rdi = index
leaq (%rdi,%rdi,2), %rax
movzql a+8(,%rax,4), %eax
```

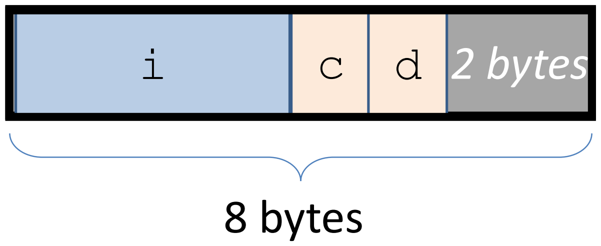
How the Programmer Can Save Space

- The compiler must respect the 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;
```





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