Machine-Level Programming IV: Data

15-213/18-213/14-513/15-513/18-613: Introduction to Computer Systems
8th Lecture, September 24, 2020

Announcements

- **Recitation Monday: C Review / C Bootcamp**

- **Lab 2 (bomblab)**
  - Due Tues, Sept. 29, 11:59pm ET
- **Written Assignment 2 peer grading**
  - Due Wed, Sept. 30, 11:59pm ET
- **Written Assignment 3 available on Canvas**
  - Due Wed, Sept. 30, 11:59pm ET

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
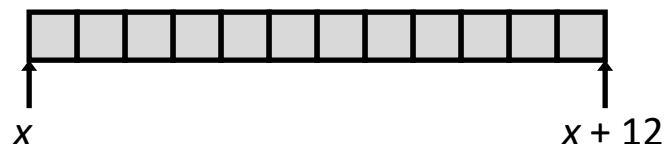 - Allocation
  - Access
  - Alignment
- **Floating Point**

Array Allocation
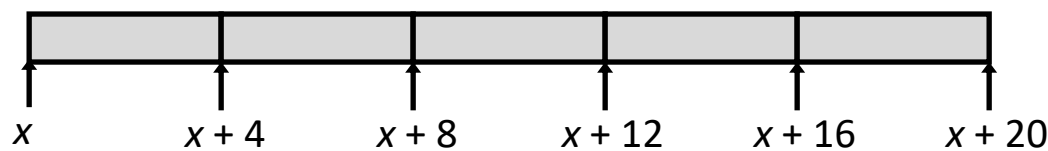
- ## Basic Principle

  T **A**[L];

  - Array of data type T and length L
  - Contiguously allocated region of L * **sizeof**(T) bytes in memory
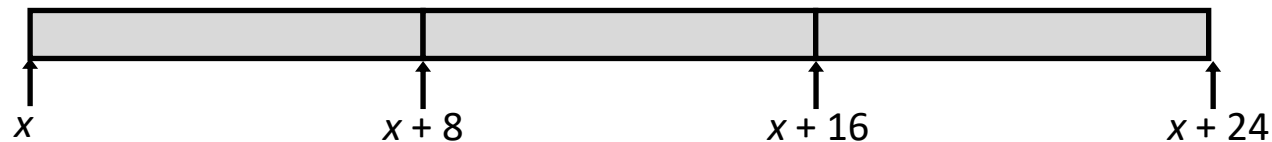
char string[12];

x $x + 12$

int val[5];

x $x + 4$ $x + 8$ $x + 12$ $x + 16$ $x + 20$

double a[3];

x $x + 8$ $x + 16$ $x + 24$

char *p[3];
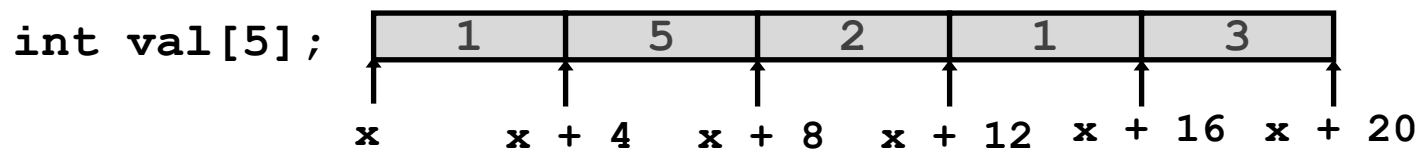
x $x + 8$ $x + 16$ $x + 24$

Array Access

- ## Basic Principle

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

  - Array of data type *T* and length *L*
  - Identifier `A` can be used as a pointer to array element 0: Type *T**

  `int val[5];`

1	5	2	1	3

  `x`   `x + 4`   `x + 8`   `x + 12`   `x + 16`   `x + 20`

- ## Reference   Type   Value

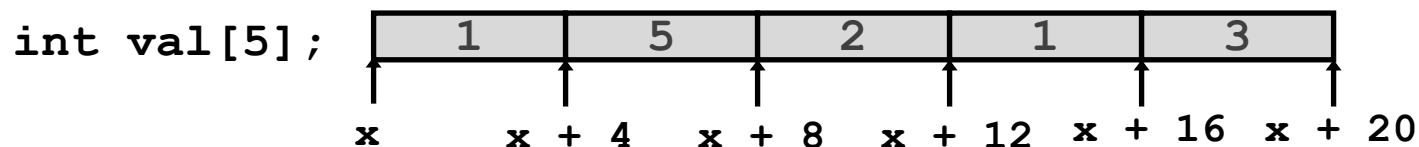 | Reference | Type | Value |
  |-----------|------|-------|
  | `val[4]` | `int` | 3 |
  | `val` | `int *` | |
  | `val+1` | `int *` | |
  | `&val[2]` | `int *` | |
  | `val[5]` | `int` | |
  | `*(val+1)` | `int` | |
  | `val + `*i* | `int *` | |

Array Access

- ## Basic Principle

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

  - Array of data type *T* and length *L*
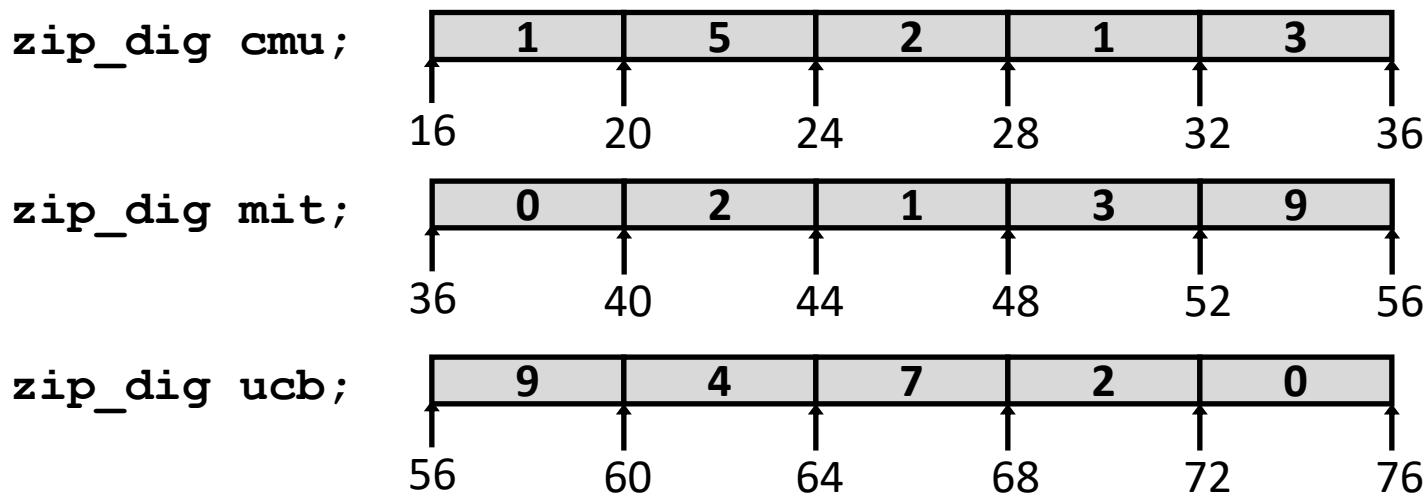 - Identifier `A` can be used as a pointer to array element 0: Type *T**

  `int val[5];`

1	5	2	1	3

  x   x + 4   x + 8   x + 12   x + 16   x + 20

- ## Reference   Type   Value

 | Reference | Type | Value | |
  |-----------|------|-------|--|
  | `val[4]` | `int` | `3` | |
  | `val` | `int *` | `x` | |
  | `val+1` | `int *` | `x + 4` | |
  | `&val[2]` | `int *` | `x + 8` | |
  | `val[5]` | `int` | `??` | |
  | `*(val+1)` | `int` | `5` | `//val[1]` |
  | `val + `*i* | `int *` | `x + 4 * i` | `//&val[i]` |

Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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

`zip_dig cmu;`

1	5	2	1	3

16   20   24   28   32   36

`zip_dig mit;`

0	2	1	3	9

36   40   44   48   52   56

`zip_dig ucb;`

9	4	7	2	0

56   60   64   68   72   76

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

Array Accessing Example

```
zip_dig cmu;
```

	1	5	2	1	3	

16   20   24   28   32   36
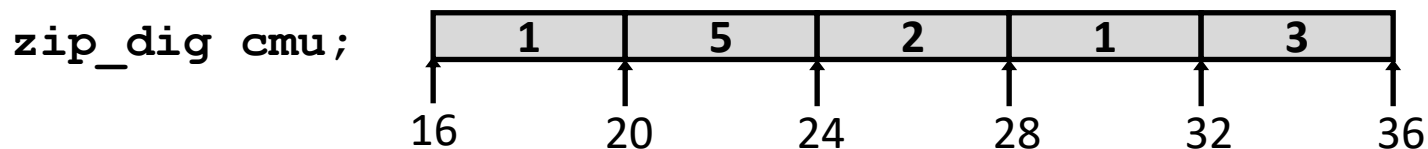
```
int get_digit
   (zip_dig z, int digit)
{
   return z[digit];
}
```

x86-64

```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

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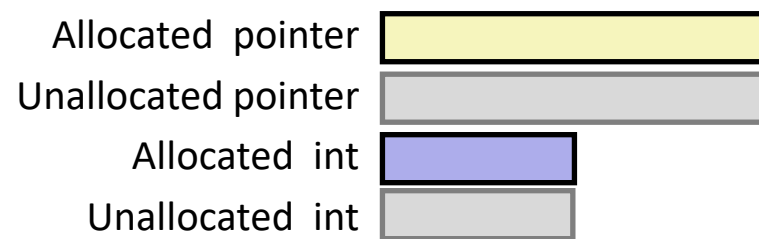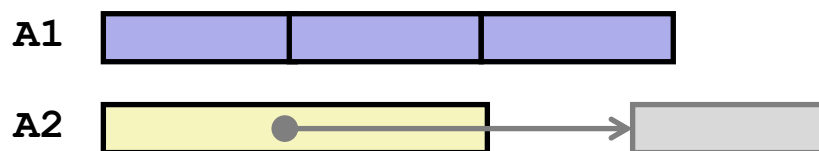- Use memory reference `(%rdi,%rsi,4)`

Array Loop Example

```c
#define ZLEN 5
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax
  jmp     .L3
.L4:
  addl    $1, (%rdi,%rax,4)
  addq    $1, %rax
.L3:
  cmpq    $4, %rax
  jbe     .L4
  rep; ret
```
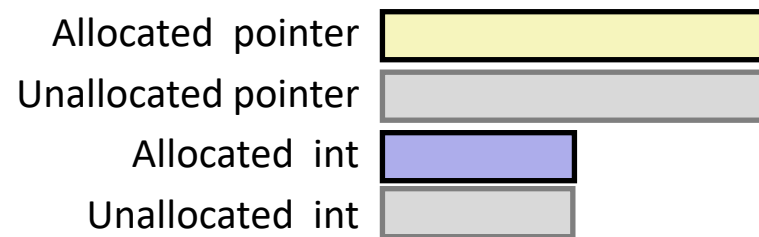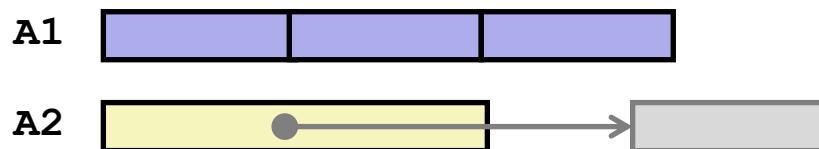
Understanding Pointers & Arrays #1

Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
int A1[3]						
int *A2						

A1

A2

Allocated pointer
Unallocated pointer
Allocated int
Unallocated int

- **Comp: Compiles (Y/N)**

- **Bad: Possible bad pointer reference (Y/N)**

- **Size: Value returned by `sizeof`**

Understanding Pointers & Arrays #1

Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8	Y	Y	4

A1

A2

Allocated pointer

Unallocated pointer

Allocated int

Unallocated int

- **Comp: Compiles (Y/N)**

- **Bad: Possible bad pointer reference (Y/N)**

- **Size: Value returned by `sizeof`**
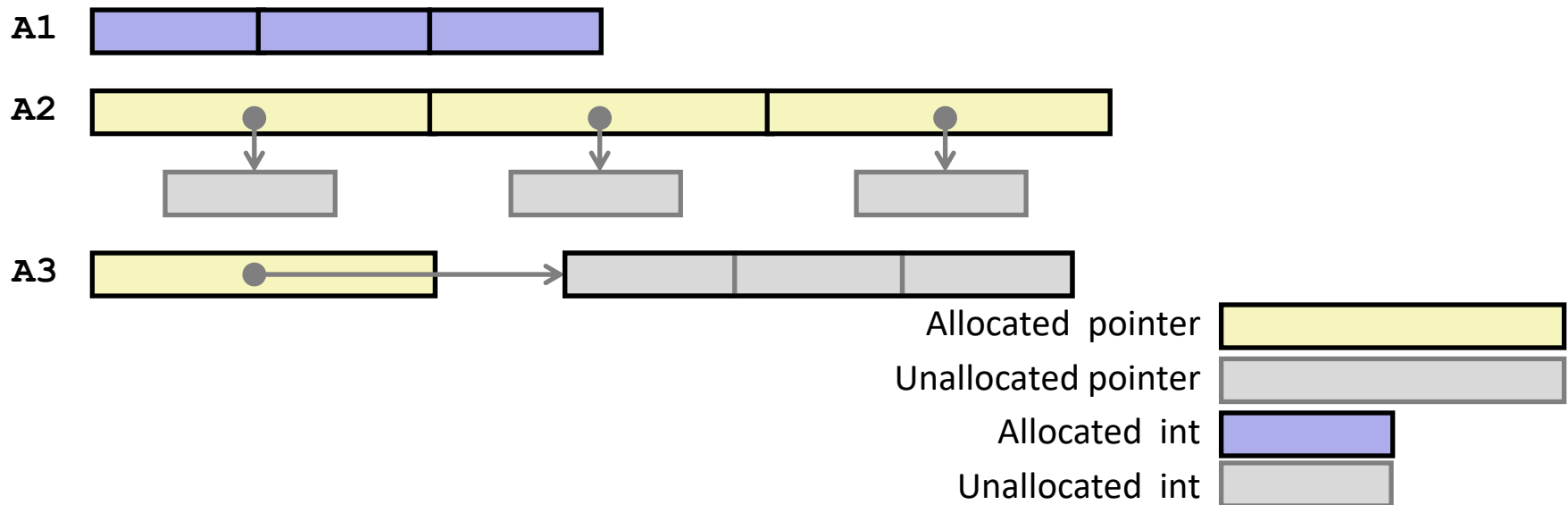
Understanding Pointers & Arrays #2

Decl	An			$*An$			$**An$		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`									
`int *A2[3]`									
`int (*A3)[3]`									

A1

A2

A3

Allocated pointer

Unallocated pointer

Allocated int

Unallocated int

Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4	N	–	–
`int *A2[3]`	Y	N	24	Y	N	8	Y	Y	4
`int (*A3)[3]`	Y	N	8	Y	Y	12	Y	Y	4

A1

A2

A3

Allocated pointer

Unallocated pointer

Allocated int

Unallocated int
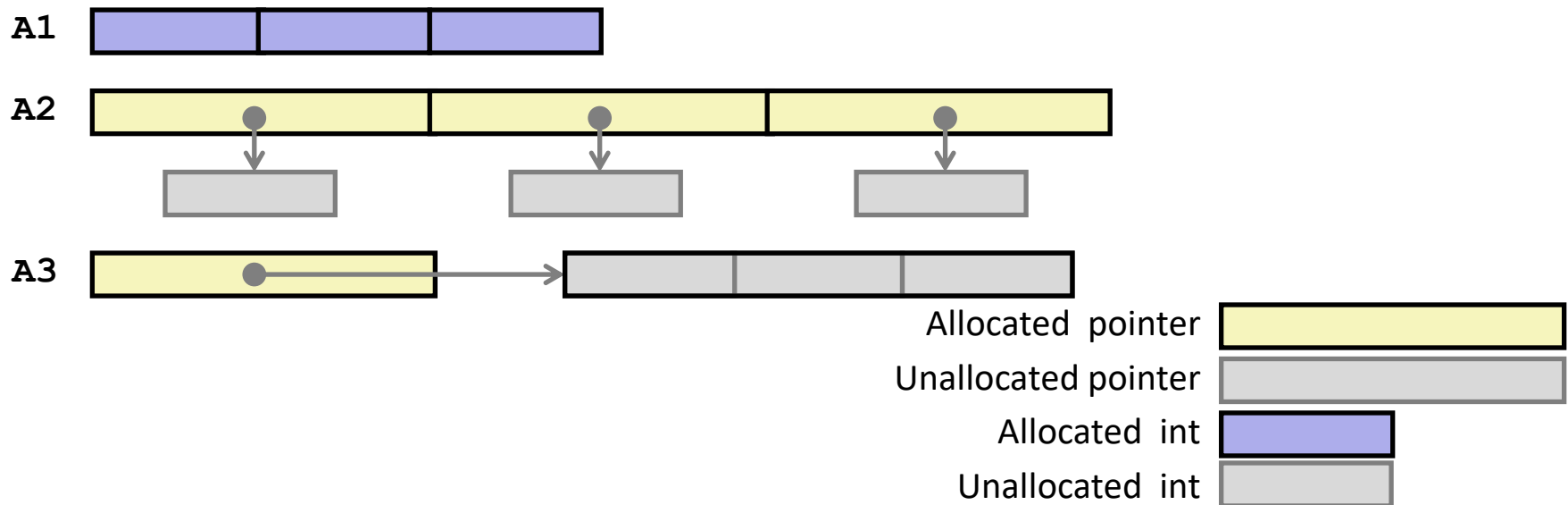
Multidimensional (Nested) Arrays

- **Declaration**

    T $\mathtt{A}[R][C];$

    - 2D array of data type T
    - R rows, C columns

- **Array Size**

    - $R * C *$ `sizeof`(T) bytes

- **Arrangement**

    - Row-Major Ordering

$$\begin{bmatrix} \mathtt{A[0][0]} & \bullet\ \bullet\ \bullet & \mathtt{A[0][C-1]} \\ \bullet & & \bullet \\ \bullet & & \bullet \\ \bullet & & \bullet \\ \mathtt{A[R-1][0]} & \bullet\ \bullet\ \bullet & \mathtt{A[R-1][C-1]} \end{bmatrix}$$

`int A[R][C];`

A [0] [0]	• • •	A [0] [C−1]	A [1] [0]	• • •	A [1] [C−1]	• • •	A [R−1] [0]	• • •	A [R−1] [C−1]

←————————————————— **4*R*C** Bytes —————————————————→
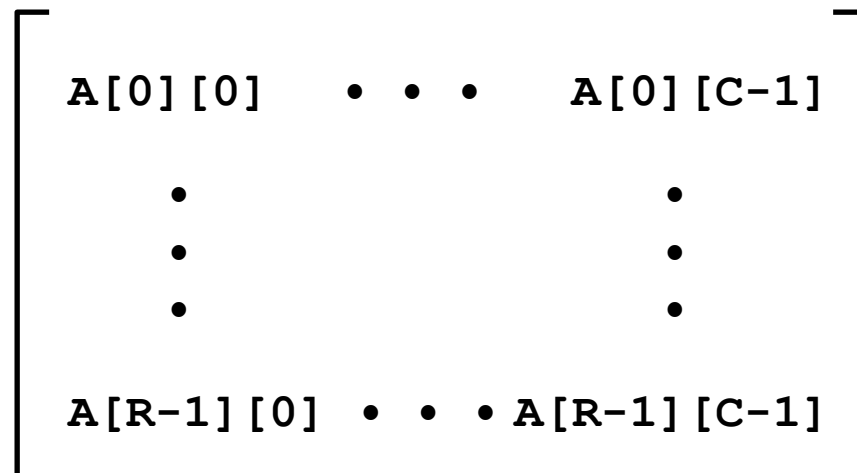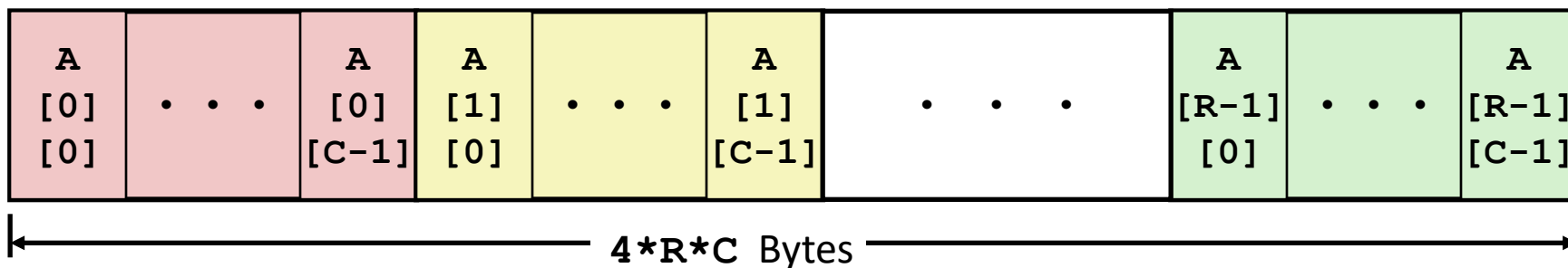
Nested Array Example

```
#define PCOUNT 4
typedef int zip_dig[5];

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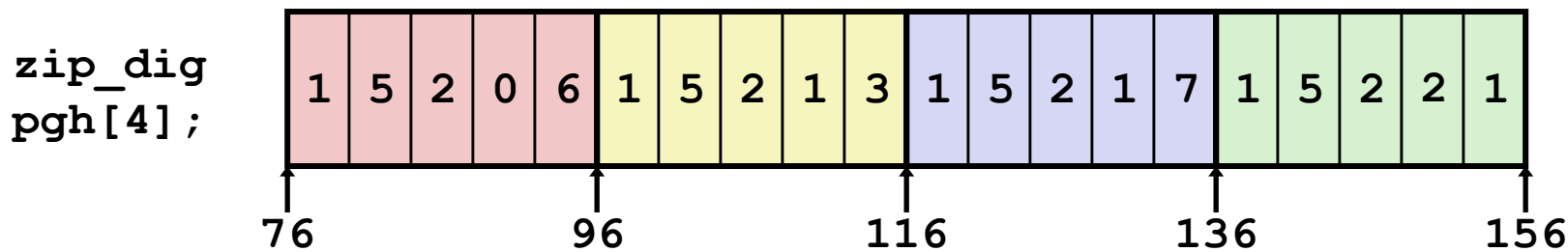
**zip_dig
pgh[4];**

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1

76   96   116   136   156

- **"`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 in memory**
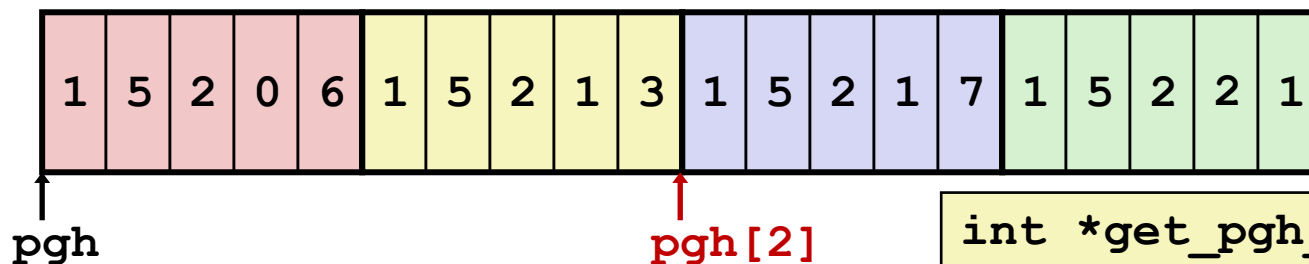
Nested Array Row Access

- **Row Vectors**
  - **A[i]** is array of *C* elements of type *T*
  - Starting address **A + i * (C * sizeof(***T***))**

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

Nested Array Row Access Code

| 1 | 5 | 2 | 0 | 6 | 1 | 5 | 2 | 1 | 3 | 1 | 5 | 2 | 1 | 7 | 1 | 5 | 2 | 2 | 1 |

↑
pgh
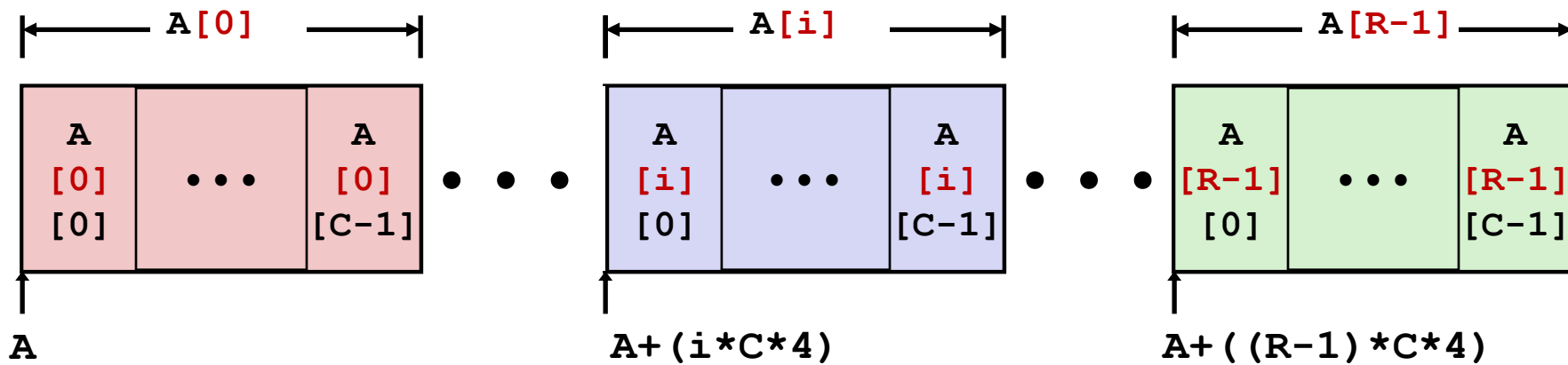
pgh[2]

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

```
 # %rdi = index
 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**
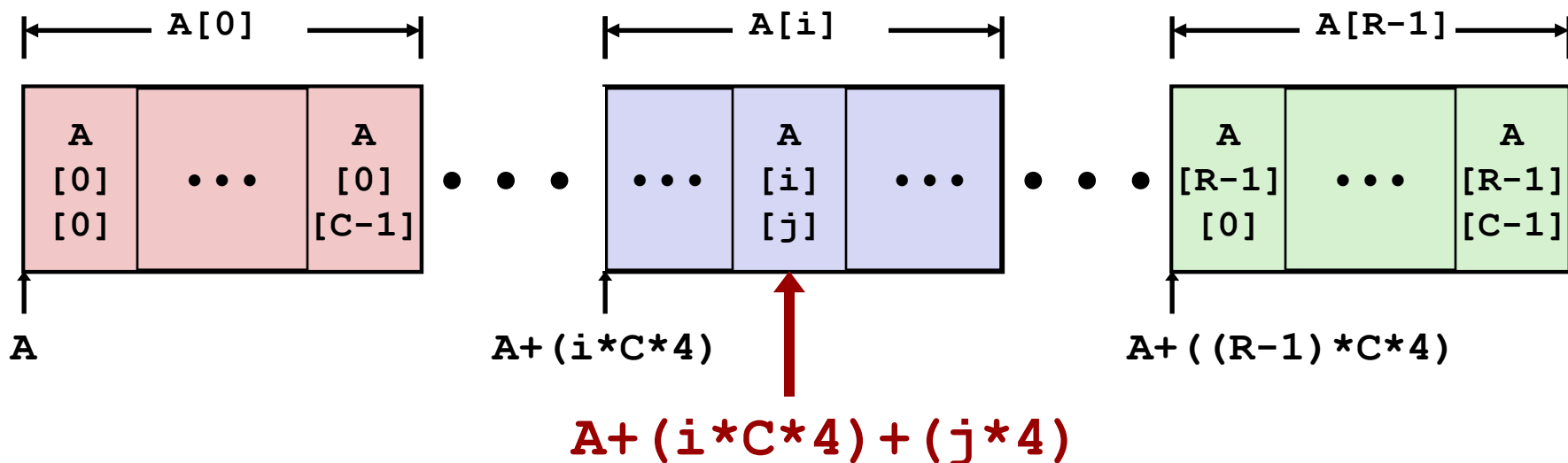- **Machine Code**
  - Computes and returns address
  - Compute as **pgh + 4*(index+4*index)**
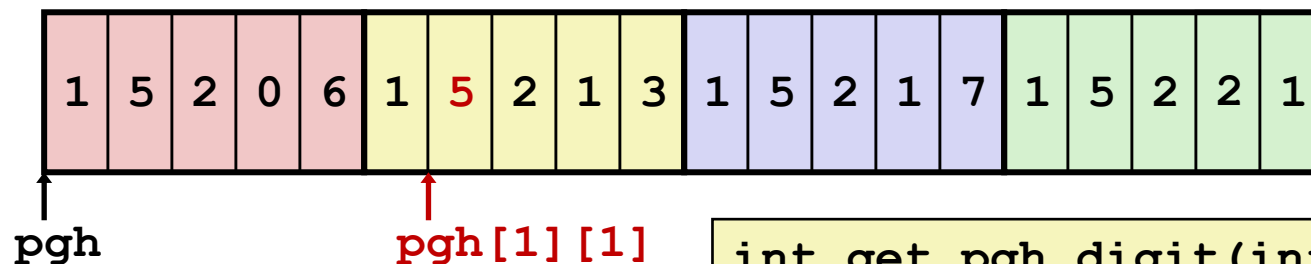
Nested Array Element Access

- ## 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

| 1 | 5 | 2 | 0 | 6 | 1 | 5 | 2 | 1 | 3 | 1 | 5 | 2 | 1 | 7 | 1 | 5 | 2 | 2 | 1 |

↑ **pgh**

↑ **pgh[1][1]**

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

```
leaq   (%rdi,%rdi,4), %rax      # 5*index
addl   %rax, %rsi               # 5*index+dig
movl   pgh(,%rsi,4), %eax       # M[pgh + 4*(5*index+dig)]
```
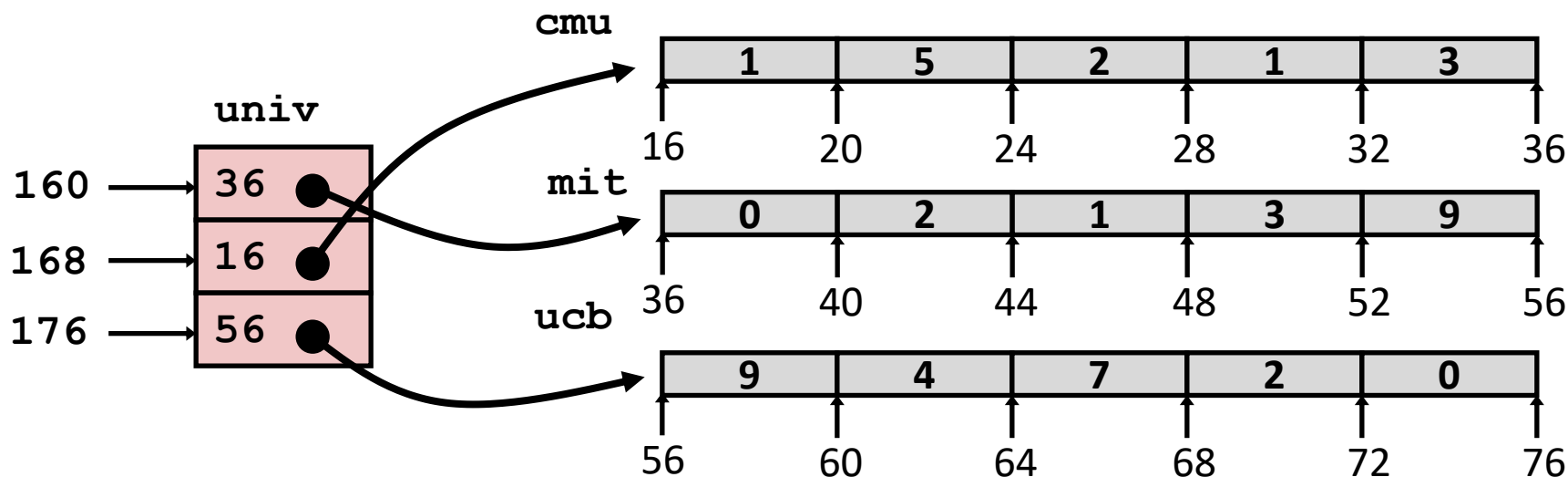
- **Array Elements**
  - ▪ **pgh[index][dig]** is **int**
  - ▪ Address: **pgh + 20*index + 4*dig**
    - **= pgh + 4*(5*index + dig)**

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```
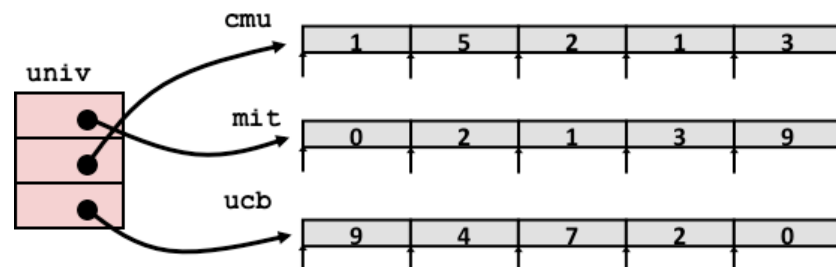
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- **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

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



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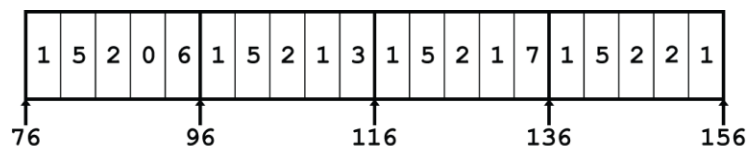
■ **Computation**

  ▪ 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
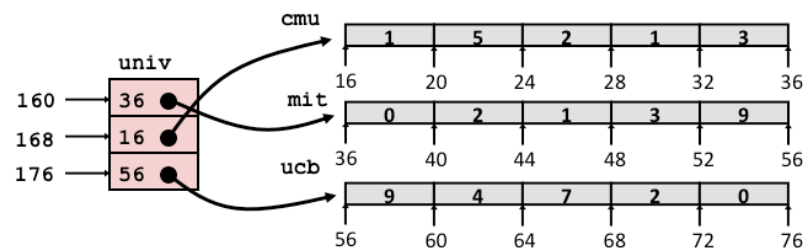
Array Element Accesses

Nested array

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

Multi-level array

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


Accesses looks similar in C, but address computations very different:

```
Mem[pgh+20*index+4*digit]    Mem[Mem[univ+8*index]+4*digit]
```

N X *N* Matrix Code

- **Fixed dimensions**
  - Know value of *N* at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A,
              size_t i, size_t j)
{
  return A[i][j];
}
```

- **Variable dimensions, explicit indexing**
  - Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element A[i][j] */
int vec_ele(size_t n, int *A,
              size_t i, size_t j)
{
  return A[IDX(n,i,j)];
}
```

- **Variable dimensions, implicit indexing**
  - Now supported by gcc

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n],
              size_t i, size_t j) {
  return A[i][j];
}
```

16 X 16 Matrix Access

- **Array Elements**
  - `int A[16][16];`
  - Address `A + i * (C * K) + j * K`
  - C = 16, K = 4

```
/* Get element A[i][j] */
int fix_ele(fix_matrix A, size_t i, size_t j) {
  return A[i][j];
}
```

```
# A in %rdi, i in %rsi, j in %rdx
salq    $6, %rsi            # 64*i
addq    %rsi, %rdi          # A + 64*i
movl    (%rdi,%rdx,4), %eax # Mem[A + 64*i + 4*j]
ret
```

n X *n* Matrix Access

- **Array Elements**
  - **size_t n;**
  - **int A[n][n];**
  - Address **A + i * (C * K) + j * K**
  - C = n, K = 4
  - Must perform integer multiplication

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n], size_t i, size_t j)
{
  return A[i][j];
}
```

```
# n in %rdi, A in %rsi, i in %rdx, j in %rcx
imulq   %rdx, %rdi            # n*i
leaq    (%rsi,%rdi,4), %rax   # A + 4*n*i
movl    (%rax,%rcx,4), %eax   # A + 4*n*i + 4*j
ret
```

Example: Array Access

```c
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];


int main(int argc, char** argv) {
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
result: 9
```

Example: Array Access

```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];


int main(int argc, char** argv) {
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
result: 9
```

Quiz Time!

Check out:

https://canvas.cmu.edu/courses/17808

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
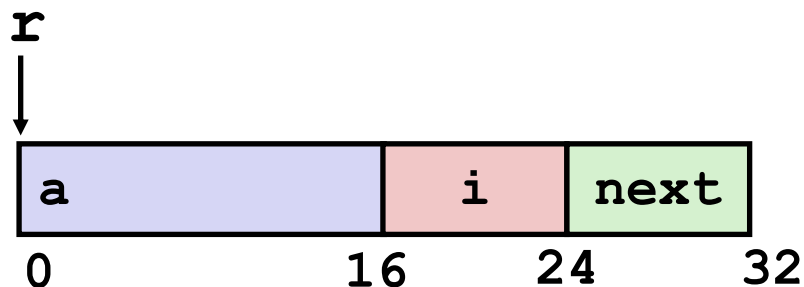 - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Floating Point**

Structure Representation

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

r

a	i	next
0	16	24	32

- **Structure represented as block of memory**
  - **Big enough to hold all of the fields**

- **Fields ordered according to declaration**
  - **Even if another ordering could yield a more compact representation (due to alignment rules—coming soon)**

- **Compiler determines overall size + positions of fields**
  - **Machine-level program has no understanding of the structures in the source code**

Generating Pointer to Structure Member

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

r   r+4*idx

a		i	next
0	16	24	32
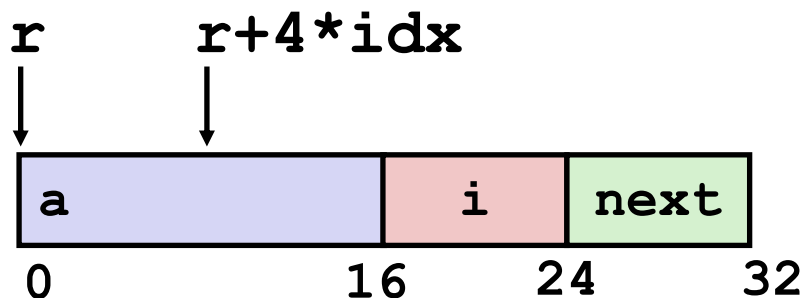
- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time
  - Compute as `r + 4*idx`

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```
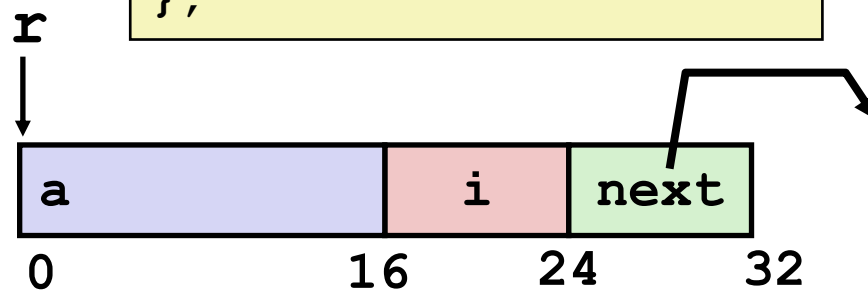
```
# r in %rdi, idx in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

Following Linked List #1

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

- **C Code**

```
long length(struct rec*r) {
    long len = 0L;
    while (r) {
        len ++;
        r = r->next;
    }
    return len;
}
```

r

a		i	next
0	16	24	32

Register	Value
%rdi	r
%rax	len

- **Loop assembly code**

```
.L11:                        #  loop:
  addq     $1, %rax          #    len ++
  movq     24(%rdi), %rdi    #    r = Mem[r+24]
  testq    %rdi, %rdi        #    Test r
  jne      .L11              #    If != 0, goto loop
```
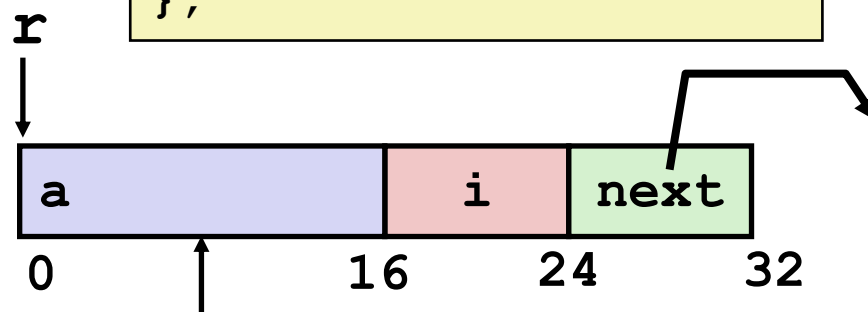
Following Linked List #2

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

- **C Code**

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    size_t i = r->i;
    // No bounds check
    r->a[i] = val;
    r = r->next;
  }
}
```

r

a		i	next

0   16   24   32

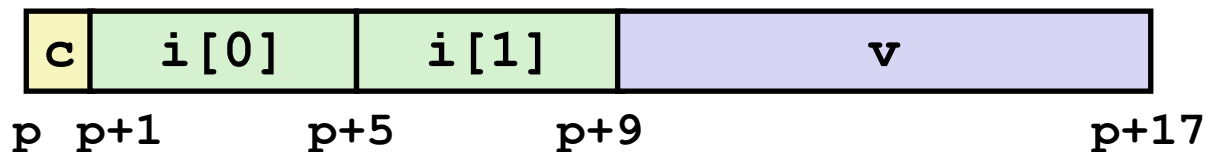**Element i**

Register	Value
%rdi	r
%rsi	val

```
.L11:                         #  loop:
  movq  16(%rdi), %rax        #    i = Mem[r+16]
  movl  %esi, (%rdi,%rax,4)   #    Mem[r+4*i] = val
  movq  24(%rdi), %rdi        #    r = Mem[r+24]
  testq %rdi, %rdi            #    Test r
  jne   .L11                  #    if !=0 goto loop
```

Structures & Alignment

■ **Unaligned Data**

c	i[0]	i[1]	v

p p+1   p+5   p+9   p+17

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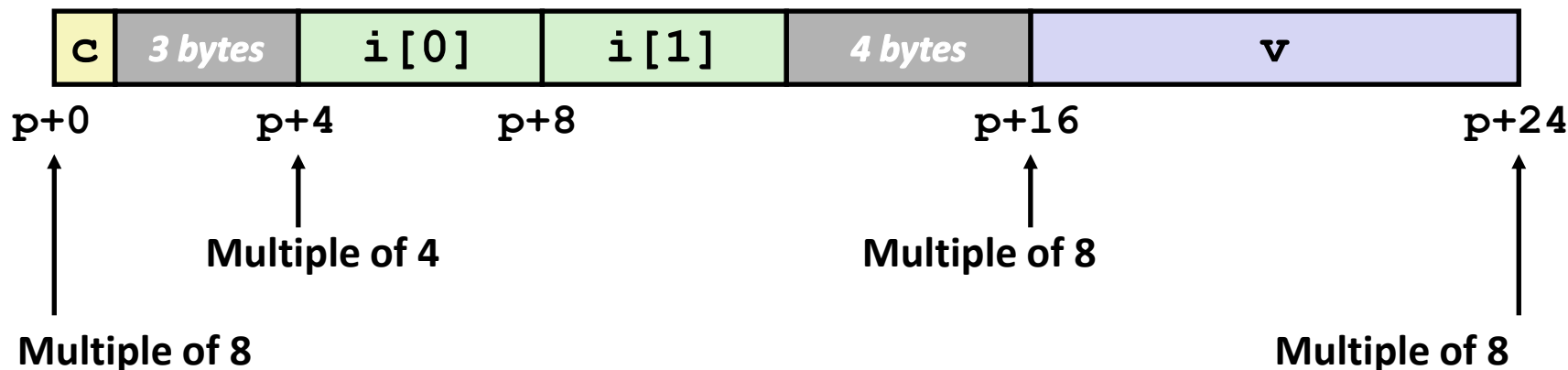
■ **Aligned Data**

  ■ Primitive data type requires **B** bytes implies

    Address must be multiple of **B**

c	*3 bytes*	i[0]	i[1]	*4 bytes*	v

p+0   p+4   p+8   p+16   p+24

Multiple of 4

Multiple of 8

Multiple of 8

Multiple of 8

Alignment Principles

- **Aligned Data**
  - Primitive data type requires **B** bytes
  - Address must be multiple of **B**
  - 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 datum that spans cache lines (64 bytes). Intel states should avoid crossing 16 byte boundaries.

      [Cache lines will be discussed in Lecture 11.]
    - Virtual memory trickier when datum spans 2 pages (4 KB pages)

      [Virtual memory pages will be discussed in Lecture 17.]

- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment (x86-64)

- **1 byte: `char, …`**
  - no restrictions on address

- **2 bytes: `short, …`**
  - lowest 1 bit of address must be 0_2

- **4 bytes: `int, float, …`**
  - lowest 2 bits of address must be 00_2

- **8 bytes: `double, long, char *, …`**
  - lowest 3 bits of address must be 000_2

Satisfying Alignment with Structures

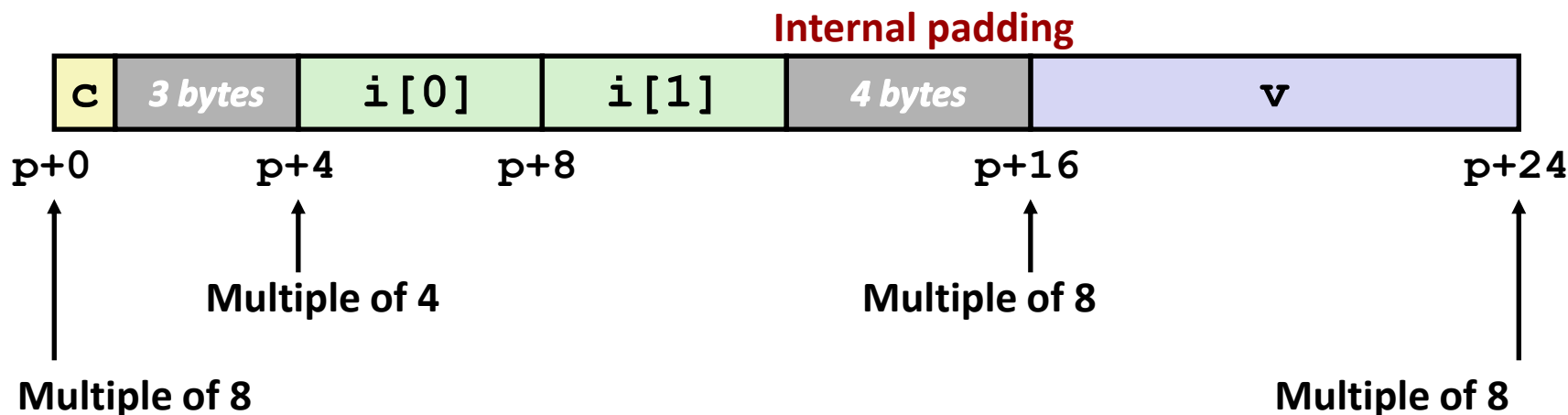- **Within structure:**
  - Must satisfy each element's alignment requirement

- **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:**
  - **K** = 8, due to **double** element

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

Internal padding

c	3 bytes	i[0]	i[1]	4 bytes	v

p+0   p+4   p+8   p+16   p+24

Multiple of 4

Multiple of 8

Multiple of 8

Multiple of 8
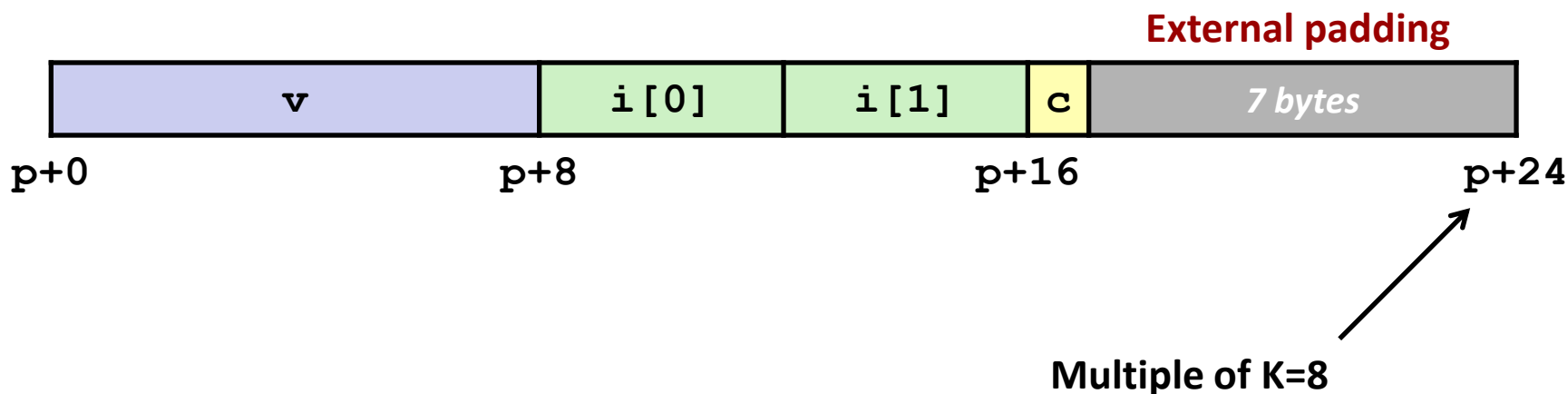
Meeting Overall Alignment Requirement

- **For largest alignment requirement K**
- **Overall structure must be multiple of K**

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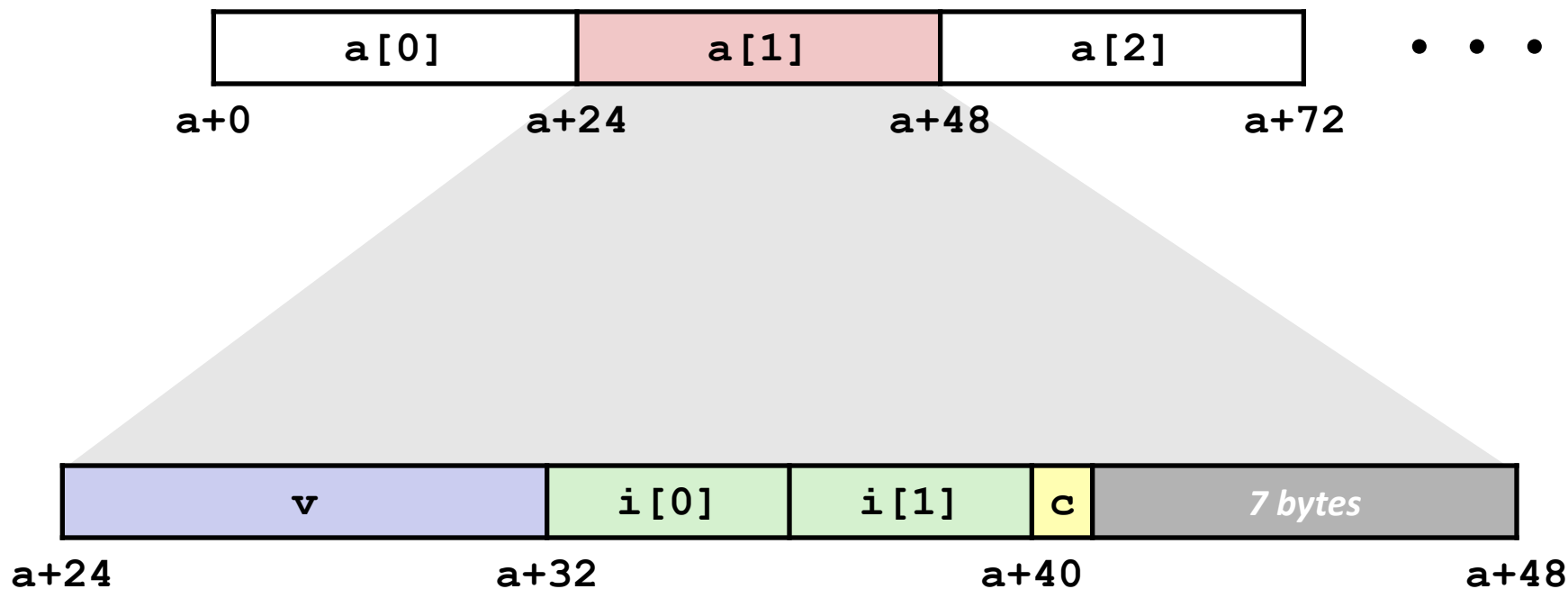
External padding

v	i[0]	i[1]	c	7 bytes

p+0   p+8   p+16   p+24

Multiple of K=8

Arrays of Structures

- **Overall structure length multiple of K**

- **Satisfy alignment requirement for every element**

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

a[0]	a[1]	a[2]

a+0   a+24   a+48   a+72

• • •

v	i[0]	i[1]	c	7 bytes

a+24   a+32   a+40   a+48

Accessing Array Elements

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

- **Compute array offset 12*idx**
  - **sizeof(S3)**, including alignment spacers

- **Element j is at offset 8 within structure**

- **Assembler gives offset a+8**
  - Resolved during linking
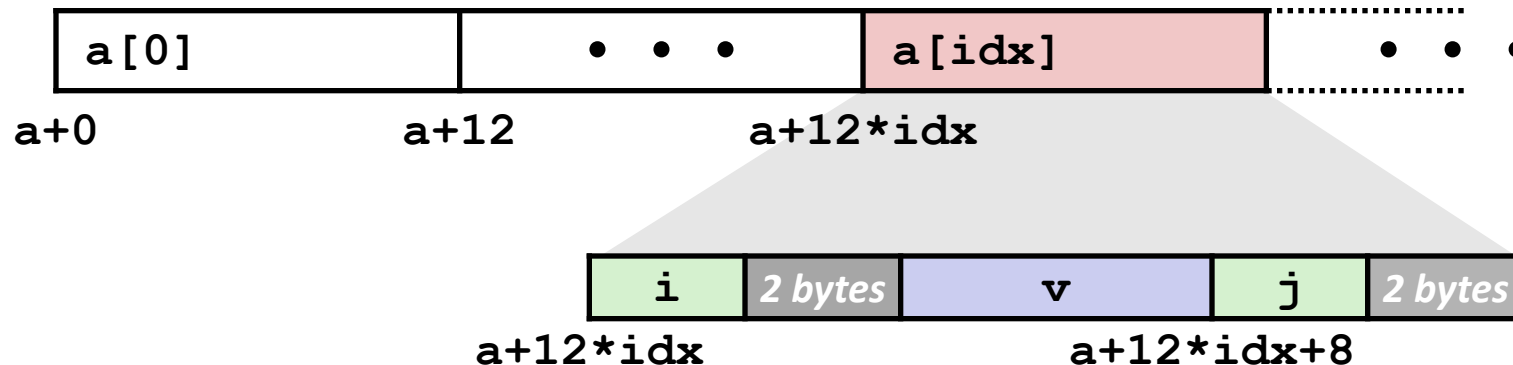

```
short get_j(int idx)
{
   return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```
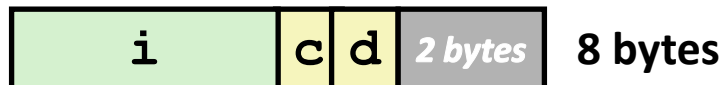
Saving Space

■ **Put large data types first**

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

➡

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

| c | 3 bytes | i | d | 3 bytes | **12 bytes** |

■ **Effect (largest alignment requirement K=4)**

| i | c | d | 2 bytes | **8 bytes** |

Example Struct Exam Question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

1. Show how foo would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf

Example Struct Exam Question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|a |X |X |X |X |X |X |X |b |b |b |b |b |b |b |b |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|c |c |c |c |d |d |d |X |e |e |e |e |e |e |e |e |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|f |f |f |f |f |f |f |f |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf

Example Struct Exam Question (Cont'd)

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

2. Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf

Example Struct Exam Question (Cont'd)

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

2. Rearrange the elements of foo to conserve the most space in memory. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|a |d |d |d |c |c |c |c |b |b |b |b |b |b |b |b |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|e |e |e |e |e |e |e |e |f |f |f |f |f |f |f |f |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Floating Point**

Background
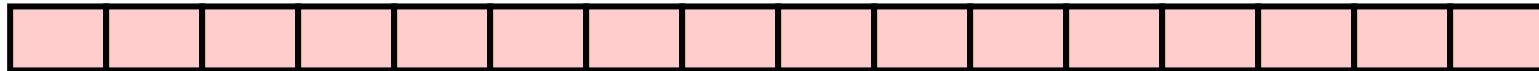
- **History**
  - x87 FP
    - Legacy, very ugly
  - SSE FP
    - Supported by Shark machines
    - Special case use of vector instructions
  - AVX FP
    - Newest version
    - Similar to SSE (but registers are 32 bytes instead of 16)
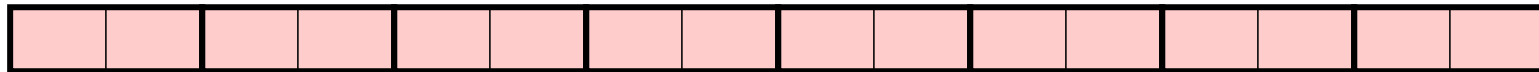   - Documented in book

Programming with SSE4

XMM Registers

- 16 total, each 16 bytes

- 16 single-byte integers

- 8 16-bit integers

- 4 32-bit integers
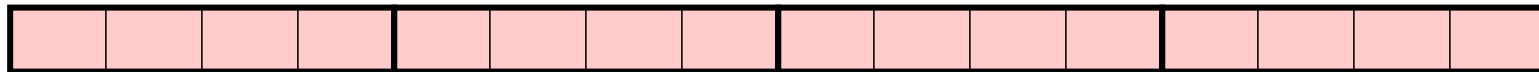
- 4 single-precision floats

- 2 double-precision floats

- 1 single-precision float

- 1 double-precision float

Scalar & SIMD Operations

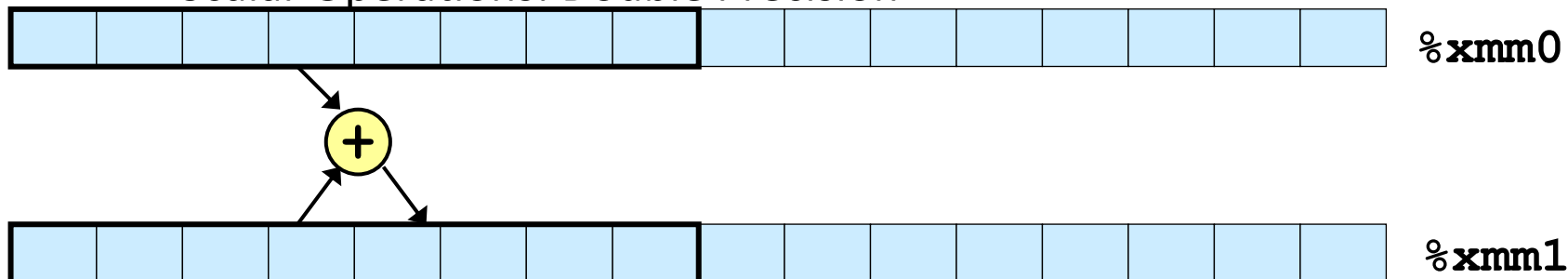■ Scalar Operations: Single Precision

`addss %xmm0,%xmm1`
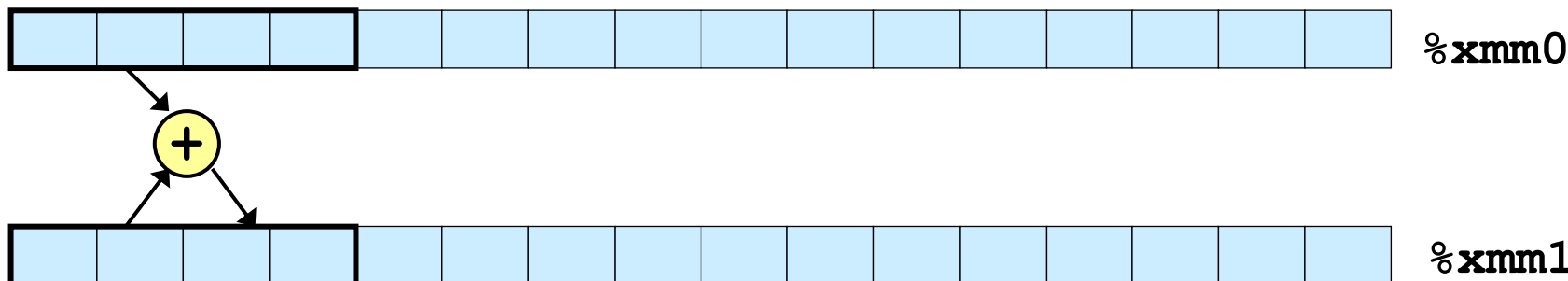
%xmm0

%xmm1

■ SIMD Operations: Single Precision

`addps %xmm0,%xmm1`
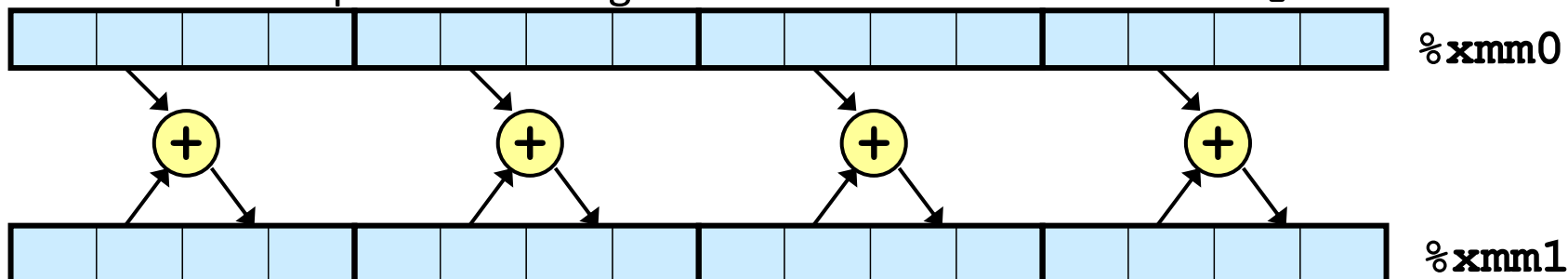
%xmm0

%xmm1

■ Scalar Operations: Double Precision

`addsd %xmm0,%xmm1`

%xmm0

%xmm1

FP Basics

- **Arguments passed in `%xmm0`, `%xmm1`, …**
- **Result returned in `%xmm0`**
- **All XMM registers caller-saved**

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

```
  # x in %xmm0, y in %xmm1
  addss   %xmm1, %xmm0
  ret
```

```
  # x in %xmm0, y in %xmm1
  addsd   %xmm1, %xmm0
  ret
```

FP Memory Referencing

- **Integer (and pointer) arguments passed in regular registers**
- **FP values passed in XMM registers**
- **Different mov instructions to move between XMM registers, and between memory and XMM registers**

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd   %xmm0, %xmm1     # Copy v
movsd    (%rdi), %xmm0    # x = *p
addsd    %xmm0, %xmm1     # t = x + v
movsd    %xmm1, (%rdi)    # *p = t
ret
```

Other Aspects of FP Code

- ***Lots* of instructions**
  - Different operations, different formats, ...

- **Floating-point comparisons**
  - Instructions **ucomiss** and **ucomisd**
  - Set condition codes ZF, PF and CF
  - Zeros OF and SF

  Parity Flag

  > UNORDERED: ZF,PF,CF←111
  > GREATER_THAN: ZF,PF,CF←000
  > LESS_THAN: ZF,PF,CF←001
  > EQUAL: ZF,PF,CF←100

- **Using constant values**
  - Set XMM0 register to 0 with instruction **xorpd %xmm0, %xmm0**
  - Others loaded from memory

Summary

- **Arrays**
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements

- **Structures**
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment

- **Combinations**
  - Can nest structure and array code arbitrarily
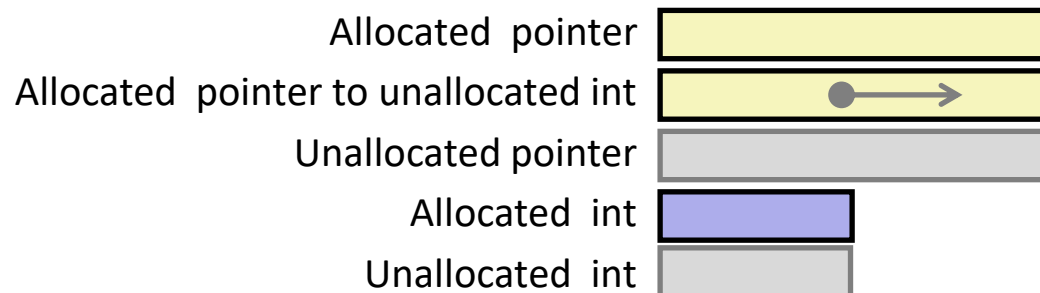
- **Floating Point**
  - Data held and operated on in XMM registers

Understanding Pointers & Arrays #3

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3][5]									
int *A2[3][5]									
int (*A3)[3][5]									
int *(A4[3][5])									
int (*A5[3])[5]									

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

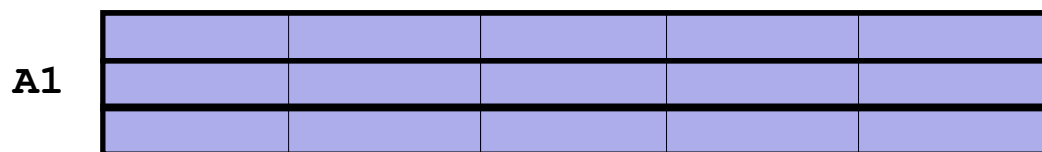| Decl	***An		
	Cmp	Bad	Size
int A1[3][5]			
int *A2[3][5]			
int (*A3)[3][5]			
int *(A4[3][5])			
int (*A5[3])[5]			

Allocated pointer

Allocated pointer to unallocated int

Unallocated pointer

Allocated int

Unallocated int

Declaration
`int A1[3][5]`
`int *A2[3][5]`
`int (*A3)[3][5]`
`int *(A4[3][5])`
`int (*A5[3])[5]`

A1

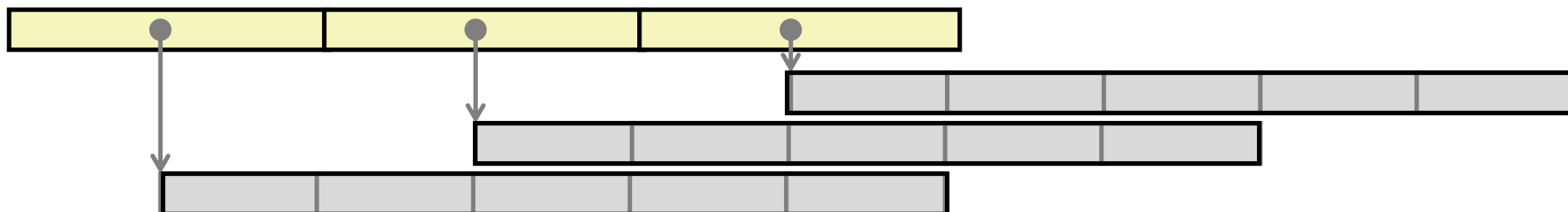**A2/A4**
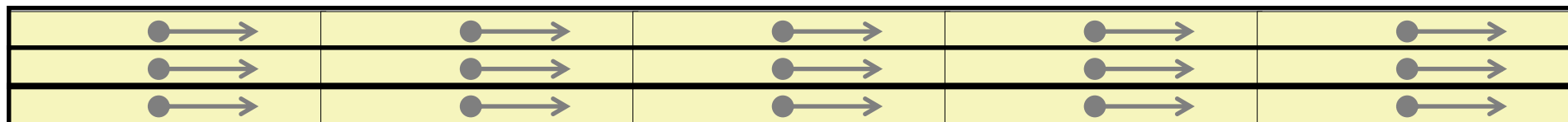
A3
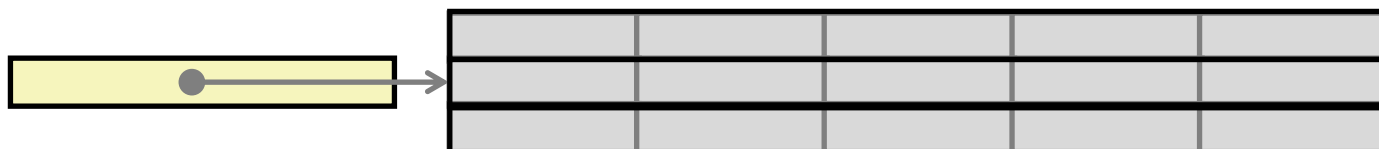
A5

Understanding Pointers & Arrays #3

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`	Y	N	60	Y	N	20	Y	N	4
`int *A2[3][5]`	Y	N	120	Y	N	40	Y	N	8
`int (*A3)[3][5]`	Y	N	8	Y	Y	60	Y	Y	20
`int *(A4[3][5])`	Y	N	120	Y	N	40	Y	N	8
`int (*A5[3])[5]`	Y	N	24	Y	N	8	Y	Y	20

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`	N	–	–
`int *A2[3][5]`	Y	Y	4
`int (*A3)[3][5]`	Y	Y	4
`int *(A4[3][5])`	Y	Y	4
`int (*A5[3])[5]`	Y	Y	4