
WELCOME
TO
15-213
18-213
15-513

Activities are posted on the website (we won't have time today but you can review them after class).

```
wget http://www.cs.cmu.edu/~213/activities/machine-data.pdf
wget http://www.cs.cmu.edu/~213/activities/machine-data.tar
tar xf machine-data.tar
cd machine-procedures
```

Announcements

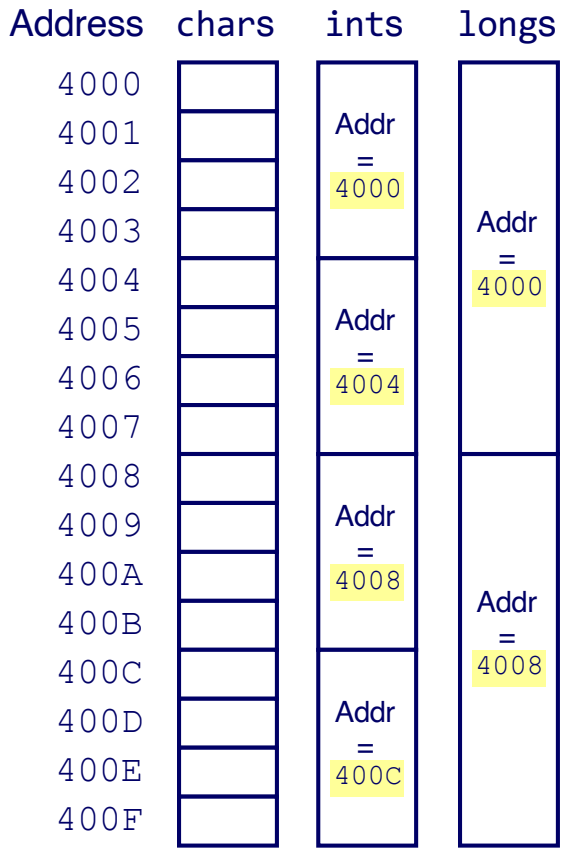
- bomblab out, due Feb 6
- writing assignment 2 out Jan 29, due Feb 5
- go to recitation!

Today: How does x86-64 implement C arrays and structures?

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point (won’t get to today)

How data is stored in memory depends on the type!

- Memory locations do not have data types
  - Types are implicit in how machine instructions *use* memory
- Addresses specify byte locations
  - Address of a larger datum is the address of its first byte
  - Addresses of successive items differ by the item's size



Recall addressing modes…

Example with `%rsi`, `%rdi`, and `%rax`

General form:

```
D(%rsi, %rdi, S) = Memory[%rsi + %rdi*S + D]
```

Special Cases

`(%rsi)`	`Memory[%rsi]`
`(%rsi, %rdi)`	`Memory[%rsi + %rdi]`
`D(%rsi, %rdi)`	`Memory[%rsi + %rdi + D]`
`(%rsi, %rdi, S)`	`Memory[%rsi + %rdi*S]`

- D is “displacement”, a constant in 1,2, or 4 bytes
- %rsi is a “**base register”**
  - Could be any of 16 integer registers
- %rsi is an **“index register”**
  - Any, except for %rsp
- S is scale is 1, 2, 4, 8

Poll: Who read the textbook?

Consider the following array declaration:

```
short x[4][4];
```

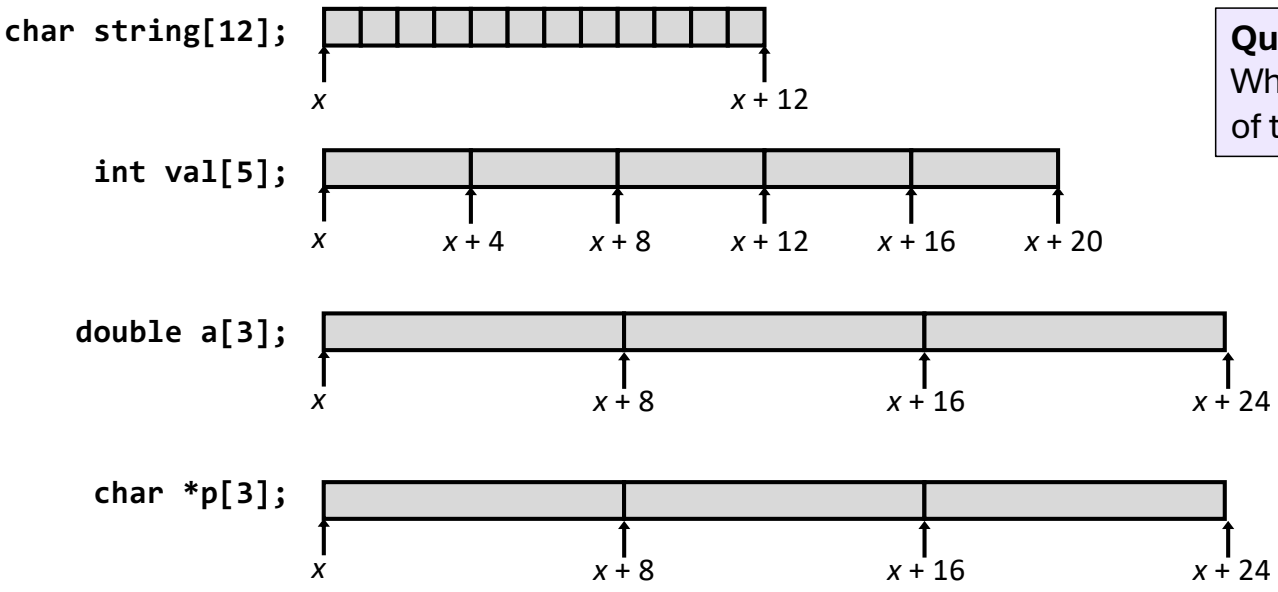
If x is stored at memory address x_a,
where is `x[2][3]` stored in memory?

[1] x_a + 11
[2] x_a + 22
[3] x_a + 44
[4] x_a + 88
[5] I don't know

https://tinyurl.com/213-array

Array Allocations: *Type* name[*Length*]

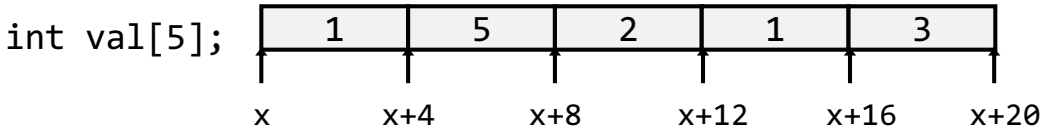
C declarations of this form allocate a contiguous region of Length * **`sizeof`**(Type) bytes in memory with the identifier `name`.



Question:
What is the total size of each of these arrays in memory?

Array Element Access: *Type* name[*Length*]

The identifier name mostly acts like a pointer[1] to array element 0.

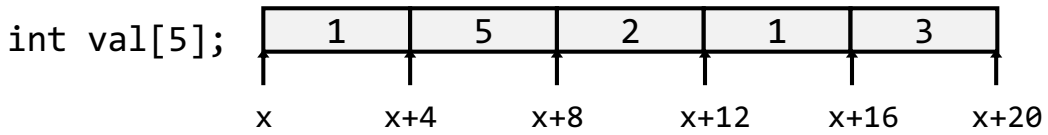


Expression	Type	Value	Assembly (val in %rdx, result in %rax)

[1] **Question:** When does an array name not act like a pointer in C?

Array Element Access: *Type* name[*Length*]

The identifier name mostly acts like a pointer[1] to array element 0.

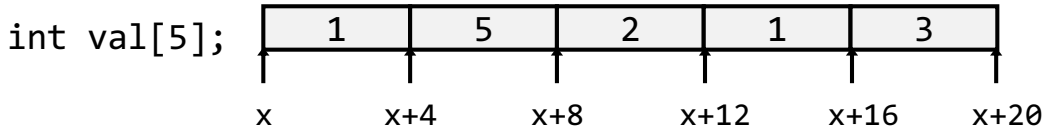


Expression	Type	Value	Assembly (val in %rdx, result in %rax)
val[4]	int	3	movl 16(%rdx), %eax

[1]**Question:** When does an array name not act like a pointer in C?

Array Element Access: *Type* name[*Length*]

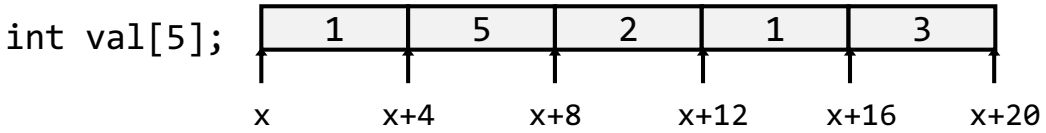
The identifier name mostly acts like a pointer[1] to array element 0.



Expression	Type	Value	Assembly (val in %rdx, result in %rax)	
val[4]	int	3	movl 16(%rdx), %eax	
val[5]	int	??	movl 20(%rdx), %eax	// access past end

Array Element Access: *Type* name[*Length*]

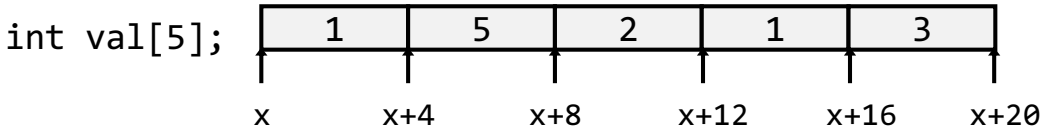
The identifier name mostly acts like a pointer[1] to array element 0.



Expression	Type	Value	Assembly (val in %rdx, result in %rax)	
val[4]	int	3	movl 16(%rdx), %eax	
val[5]	int	??	movl 20(%rdx), %eax	// access past end
***(val+3)**	int	1	movl 12(%rdx), %eax	// same as val[3]

Array Element Access: *Type* name[*Length*]

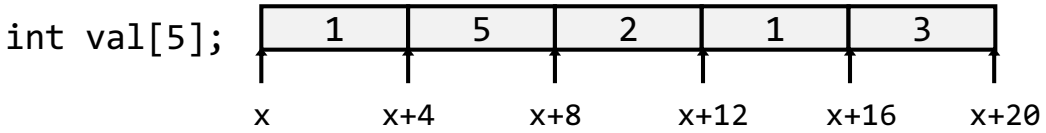
The identifier name mostly acts like a pointer[1] to array element 0.



Expression	Type	Value	Assembly (val in %rdx, result in %rax)	
val[4]	int	3	movl 16(%rdx), %eax	
val[5]	int	??	movl 20(%rdx), %eax	// access past end
***(val+3)**	int	1	movl 12(%rdx), %eax	// same as val[3]
val	int *	x	movq %rdx, %rax	

Array Element Access: *Type* name[*Length*]

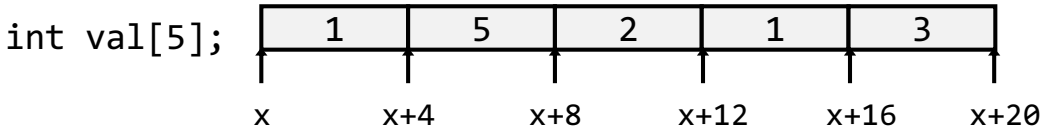
The identifier name mostly acts like a pointer[1] to array element 0.



Expression	Type	Value	Assembly (val in %rdx, result in %rax)	
val[4]	int	3	movl 16(%rdx), %eax	
val[5]	int	??	movl 20(%rdx), %eax	// access past end
***(val+3)**	int	1	movl 12(%rdx), %eax	// same as val[3]
val	int *	x	movq %rdx, %rax	
val+1	int *	x + 4	leaq 4(%rdx), %rax	

Array Element Access: *Type* name[*Length*]

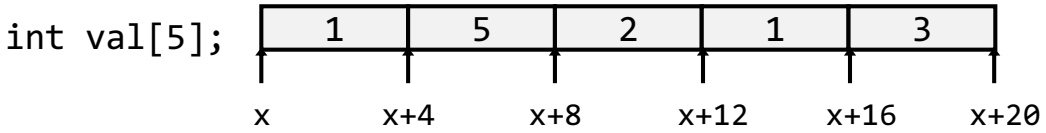
The identifier name mostly acts like a pointer[1] to array element 0.



Expression	Type	Value	Assembly (val in %rdx, result in %rax)	
val[4]	int	3	movl 16(%rdx), %eax	
val[5]	int	??	movl 20(%rdx), %eax	// access past end
***(val+3)**	int	1	movl 12(%rdx), %eax	// same as val[3]
val	int *	x	movq %rdx, %rax	
val+1	int *	x + 4	leaq 4(%rdx), %rax	
&val[2]	int *	x + 8	leaq 8(%rdx), %rax	// same as val+2

Array Element Access: *Type* name[*Length*]

The identifier name mostly acts like a pointer[1] to array element 0.

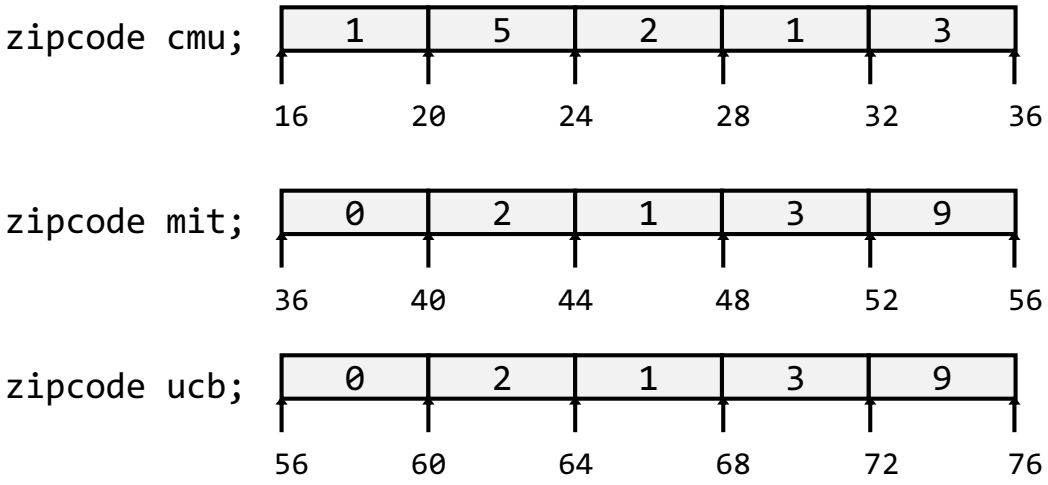


Expression	Type	Value	Assembly (val in %rdx, result in %rax)	
val[4]	int	3	movl 16(%rdx), %eax	
val[5]	int	??	movl 20(%rdx), %eax	// access past end
***(val+3)**	int	1	movl 12(%rdx), %eax	// same as val[3]
val	int *	x	movq %rdx, %rax	
val+1	int *	x + 4	leaq 4(%rdx), %rax	
&val[2]	int *	x + 8	leaq 8(%rdx), %rax	// same as val+2
val + i	int *	x + 4*i	leaq (%rdx,%rcx,4), %rax	// same as &val[i]

Array Allocation Example:

```
1  #define ZLEN 5
2  typedef int zipcode[ZLEN];
3
4  zipcode cmu = { 1, 5, 2, 1, 3 };
5  zipcode mit = { 0, 2, 1, 3, 9 };
6  zipcode ucb = { 9, 4, 7, 2, 0 };
```

"zipcode cmu" is equivalent to "int cmu[5]"

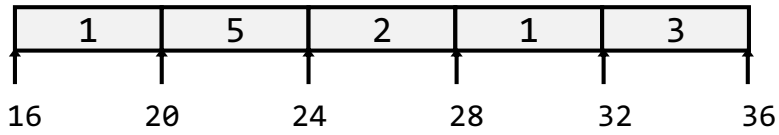


Example arrays were allocated in successive 20 byte blocks (not guaranteed to happen in general).

Array Accessing Example:

```
1  int get_digit
2    (zipcode z, int digit)
3  {
4    return z[digit];
5  }
```

zipcode cmu;



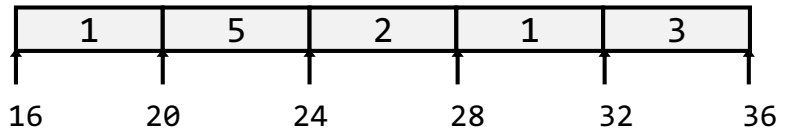

```
1  # %rdi = z, %rsi = digit
2  0000000000001129 <get_digit>:
3      1129:   48 63 f6          movslq %esi,%rsi
4      112c:   8b 04 b7          mov    (%rdi,%rsi,4),%eax   # z[digit]
5      112f:   c3                ret
```

- `%rdi` contains starting address of array: “base register”
- `%rsi` contains array index: “index register”
- `zipcode` is an array of integers so “scale” is 4
- Desired digit is at `%rdi + 4*%rsi` which is equivalent to `(%rdi, %rsi, 4)`

Array Loop Example:

```
1  void increment(zipcode z) {
2    size_t i;
3    for (i = 0; i < ZLEN; i++)
4      z[i]++;
6  }
```

zipcode cmu;

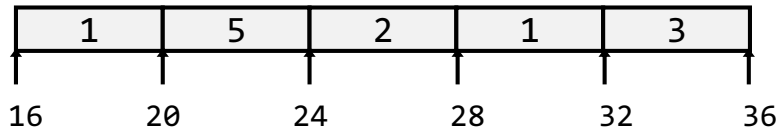



```
 1   # %rdi = z
 2    movl    $0, %eax            #   i = 0
 3    jmp     .L3                 #   goto middle
 4  .L4:                          # loop:
 5    addl    $1, (%rdi,%rax,4)   #   z[i]++
 6    addq    $1, %rax            #   i++
 7  .L3:                          # middle
 8    cmpq    $4, %rax            #   i:4
 9    jbe     .L4                 #   if <=, goto loop
10    rep; ret
```

Array Loop Example:

```
1  void increment(zip_dig z) {
2    size_t i;
3    for (i = 0; i < ZLEN; i++)
4      z[i]++;
6  }
```

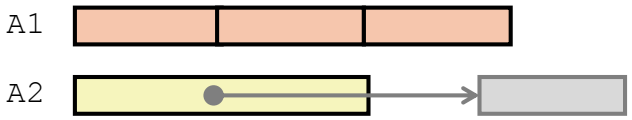
zipcode cmu;

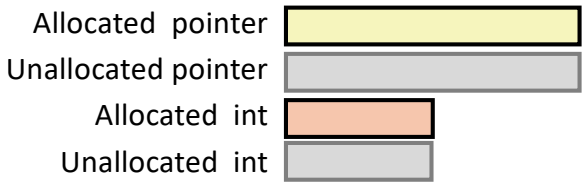



```
1   # %rdi = z
2     movl    $0, %eax            #   i = 0
3     jmp     .L3                 #   goto middle
4   .L4:                          # loop:
5     addl    $1, (%rdi,%rax,4)   #   z[i]++
6     addq    $1, %rax            #   i++
7   .L3:                          # middle
8     cmpq    $4, %rax            #   i:4
9     jbe     .L4                 #   if <=, goto loop
10    rep; ret
```

Understanding Pointers & Arrays #1

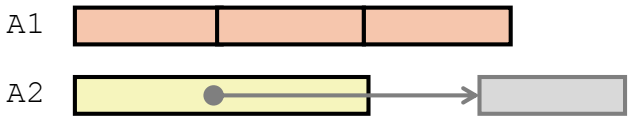
Declaration	A1 , A2			*A1 , *A2		
	Compiles?	Possible bad pointer reference?	`sizeof`	Compiles?	Possible bad pointer reference?	`sizeof`
int A1[3]	**Y**	**N**	**12**	**Y**	**N**	**4**
int *A2	**Y**	**N**	**8**	**Y**	**Y**	**4**

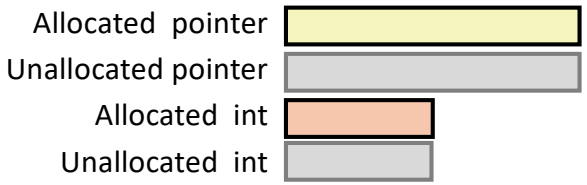




Understanding Pointers & Arrays #1

Declaration	A1 , A2			*A1 , *A2		
	Compiles?	Possible bad pointer reference?	`sizeof`	Compiles?	Possible bad pointer reference?	`sizeof`
int A1[3]	**Y**	**N**	**12**	**Y**	**N**	**4**
int *A2	**Y**	**N**	**8**	**Y**	**Y**	**4**

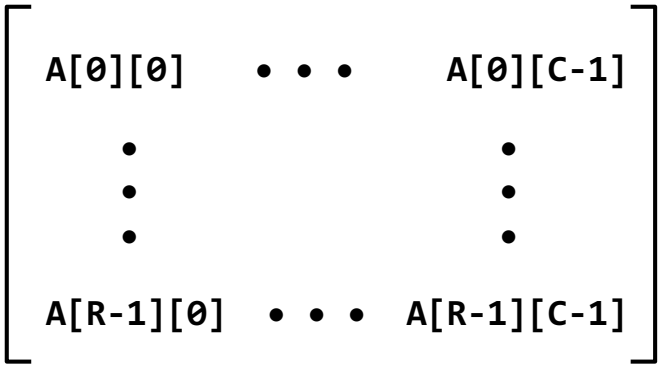




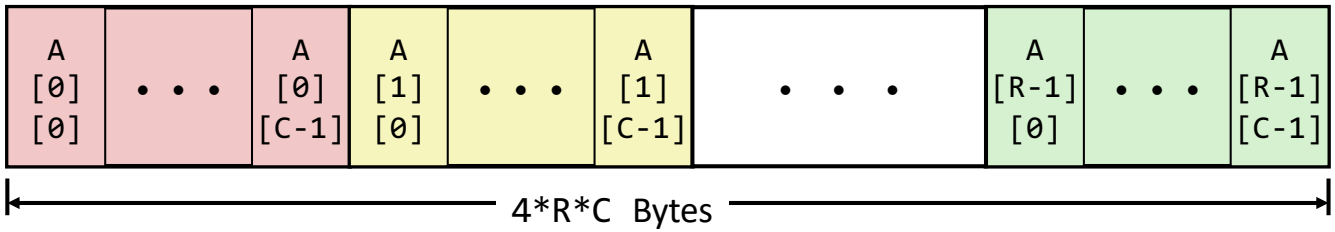
Multidimensional (Nested) Array Allocation

Declaration: *T* A[*numRows*][*numCols*]

- A 2D array of data type *T*
- *numRows* rows, *numCols* columns
- Array size:
  numRows * *numCols* * **sizeof**(*T*) bytes
- Row-major ordering



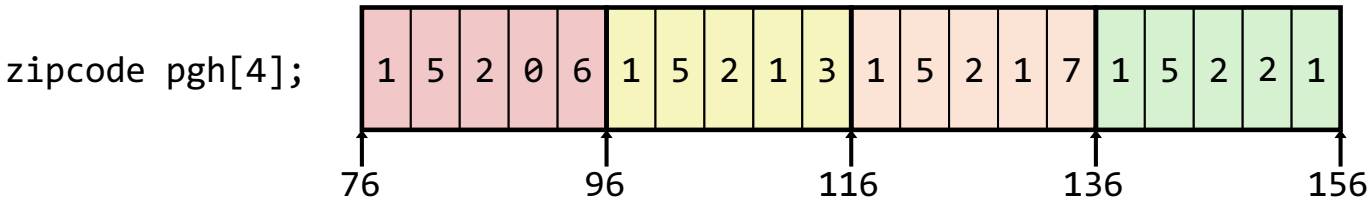

```
int A[R][C];
```



Nested Array Allocation Example

```
1  #define PCOUNT 4
2  typedef int zipcode[5];
3
4  zipcode pgh[PCOUNT] =
5     {{ 1, 5, 2, 0, 6 },
6      { 1, 5, 2, 1, 3 },
7      { 1, 5, 2, 1, 7 },
8      { 1, 5, 2, 2, 1 }};
```

- A 2D array of data type int
- *4* rows, *5* columns
- Array size: *5* * 4* **`sizeof`**`(int)` = 80 bytes
- Row-major ordering



"`zipcode pgh[4]`" is equivalent to "`int pgh[4][5]`"

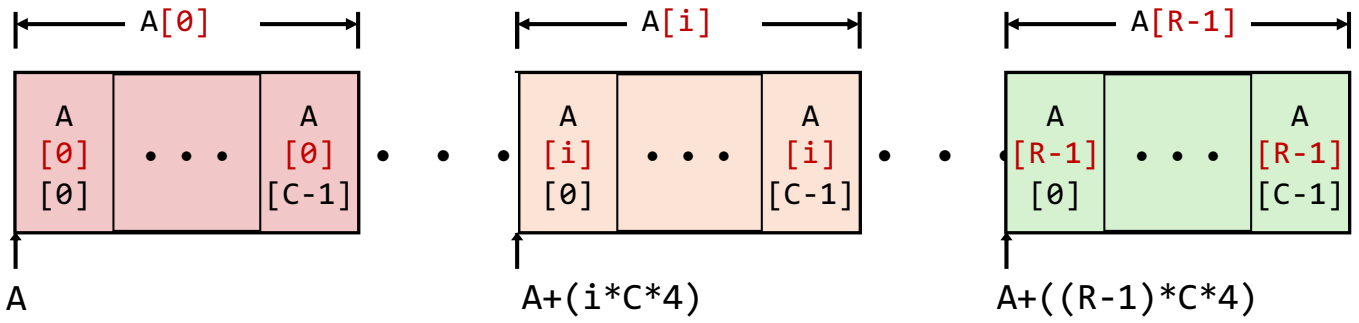
Nested Array Row Access

Declaration: *T* A[*numRows*][*numCols*]

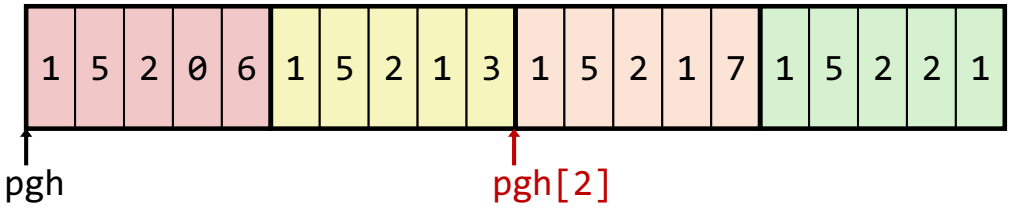
- A[i] is array of *numCols* elements of type *T*
- Starting address: A + i * (*numCols* * sizeof(*T*))

size of a row

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



Nested Array Row Access Code




```
1  int *get_pgh_zip(int index)
2  {
3    return pgh[index];
4  }
```

- `pgh[index]` is an array of 5 `int`s
- Starting address pgh+20*index
- which is is equivalent to pgh + 4 * `(index+4*index)`

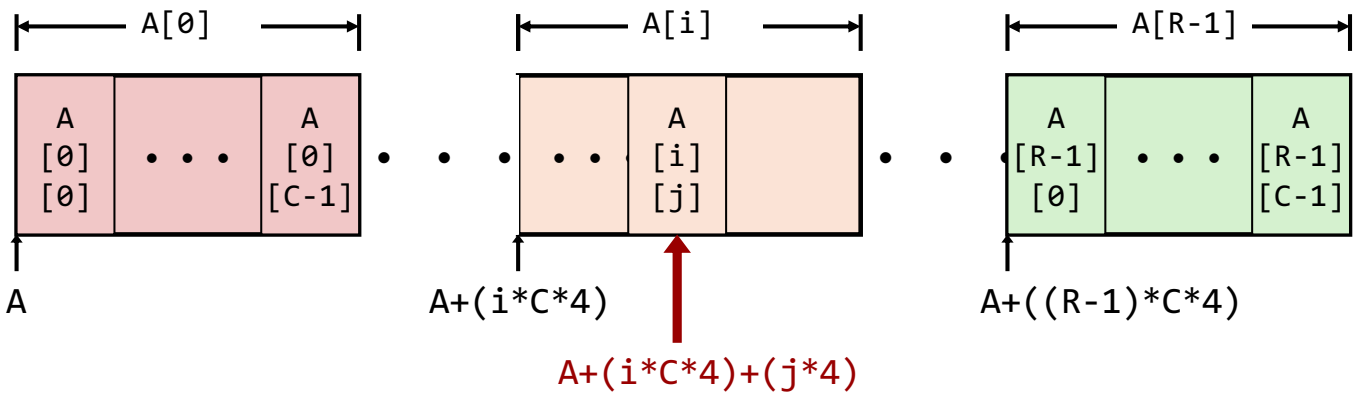
```
1  # %rdi = index
2  leaq (%rdi,%rdi,4),%rax     # 5 * index
3  leaq pgh(,%rax,4),%rax      # pgh + (20 * index)
```

Nested Array Element Access

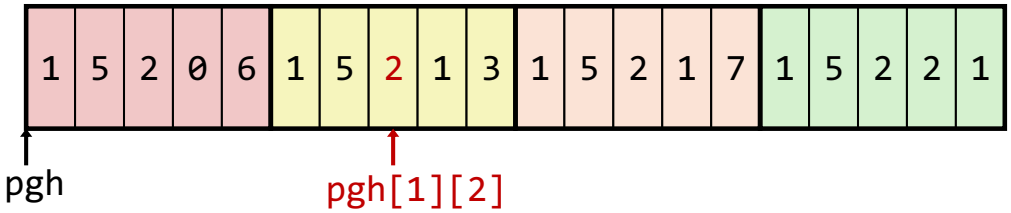
Declaration: *T* `A[i][j]`

- `A[i][j]` is array of type *T*, which requires *K* bytes
- Address `A + i * (numCols * K) + j * K`
  `= A + (i * numCols + j) * K`

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



Nested Array Element Access Code




```
1  int get_pgh_digit(int index, int dig)
2  {
3    return pgh[index][dig];
4  }
```

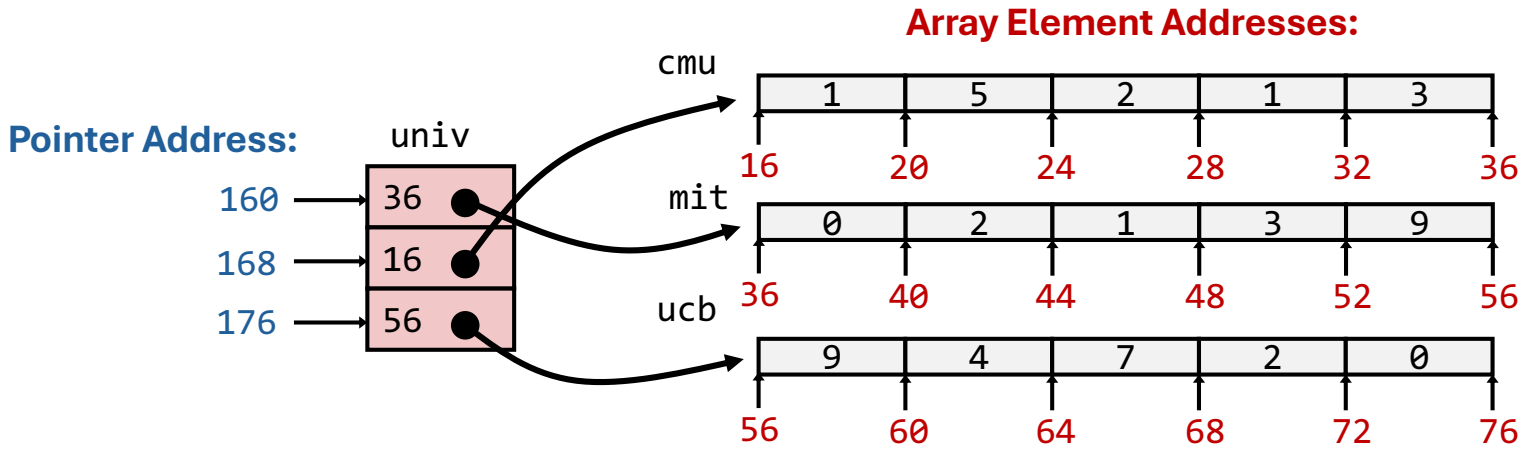
- `pgh[index][dig]` is `int`
- Address computation:
  `pgh + 20*index + 4*dig`
  `= pgh + 4*(5*index + dig)`

```
1  # %rdi = index, %rsi = dig
2      leaq   (%rdi,%rdi,4), %rax     # 5*index
3      addl   %rax, %rsi              # 5*index+dig
4      movl   pgh(,%rsi,4), %eax      # M[pgh + 4*(5*index+dig)]
```

Multi-Level Array Allocation Example

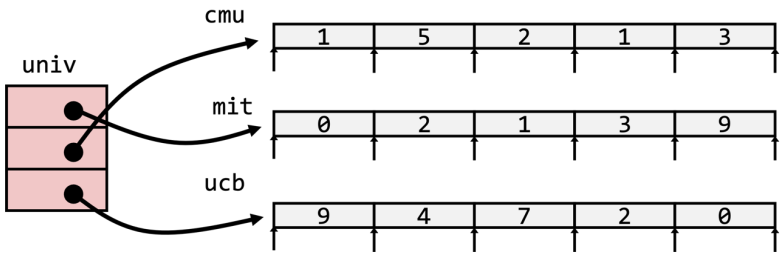
```
1  #define UCOUNT 3
2
3  zipcode cmu = { 1, 5, 2, 1, 3 };
4  zipcode mit = { 0, 2, 1, 3, 9 };
5  zipcode ucb = { 9, 4, 7, 2, 0 };
6
7  int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Variable `univ` denotes array of 3 elements
- Each element is an 8 byte pointer
- Each pointer points to array of 5 `int`'s



Multi-Level Array Element Access Example

```
1  int get_univ_digit
2    (size_t index, size_t digit)
3  {
4    return univ[index][digit];
5  }
```




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

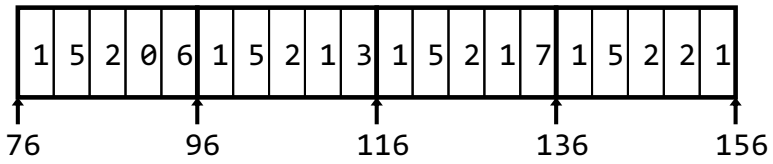
- Address computation:
  `Mem[Mem[univ+8*index]+4*digit]`

- Must do two memory reads to get pointer to row array and then access element within array

Array element access looks similar in C but address computations are very different.

Nested array

```
1  int get_pgh_digit
2    (size_t index, size_t digit
3  {
4    return pgh[index][digit];
5  }
```

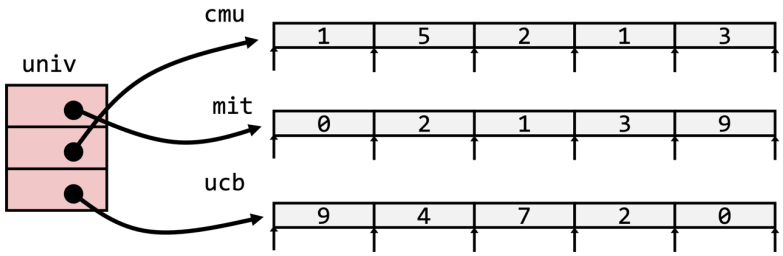


Address computation:

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

Multi-level array

```
1  int get_univ_digit
2    (size_t index, size_t digit)
3  {
4    return univ[index][digit];
5  }
```



Address computation:

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

N x *N* Matrix Code

- Fixed dimensions
  - Know value of *N* at compile time
- Variable dimensions, explicit indexing
  - Traditional way to implement dynamic arrays
- Variable dimensions, implicit indexing
  - Not in K&R; added to language in 1999

```
1  # define N 16
2  typedef int fix_matrix[N][N];
3  /* Get element A[i][j] */
4  int fix_ele(fix_matrix A,
5              size_t i, size_t j)
6  {
7    return A[i][j];
8  }
```

```
1  # define IDX(n, i, j) ((i)*(n)+(j))
2  /* Get element A[i][j] */
3  int vec_ele(size_t n, int *A,
4              size_t i, size_t j)
5  {
6    return A[IDX(n,i,j)];
7  }
```

```
1  /* Get element A[i][j] */
2  int var_ele(size_t n, int A[n][n],
3              size_t i, size_t j) {
4    return A[i][j];
5  }
```

N x *N* Matrix Code

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

```
1  # define N 16
2  typedef int fix_matrix[N][N];
3  /* Get element A[i][j] */
4  int fix_ele(fix_matrix A,
5              size_t i, size_t j)
6  {
7    return A[i][j];
8  }
```

Example: 16 x 16 Matrix Access

`int A[16][16];`

- Address `A + i * (numCols * K) + j * K`
- numCols = 16, K = 4 (size of an `int`)

```
1  /* Get element A[i][j] */
2  int fix_ele(fix_matrix A, size_t i, size_t j) {
3    return A[i][j];
4  }
```

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

N x *N* Matrix Code

- Variable dimensions, implicit indexing
  - Not in K&R; added to language in 1999

```
1  /* Get element A[i][j] */
2  int var_ele(size_t n, int A[n][n],
3              size_t i, size_t j) {
4    return A[i][j];
5  }
```

N x N Matrix Access

`int A[n][n];`

- Address `A + i * (numCols * K) + j * K`
- numCols = n, K=4 (size of an `int`)

```
1  /* Get element A[i][j] */
2  int var_ele(size_t n, int A[n][n], size_t i, size_t j) {
3    return A[i][j];
4  }
```

```
1  # n in %rdi, A in %rsi, i in %rdx, j in %rcx
2   imulq   %rdx, %rdi             # n*i
3   leaq    (%rsi,%rdi,4), %rax    # A + 4*n*i
4   movl    (%rax,%rcx,4), %eax    # Mem[A + 4*n*i + 4*j]
5   ret
```

Quiz Time!

Canvas Quiz: Day 6 - Machine Data

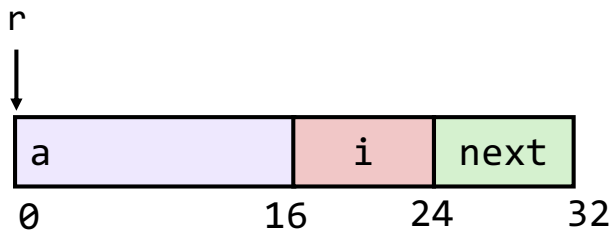
https://tinyurl.com/213-lec6

Today: How does x86-64 implement C arrays and structures?

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

Structures are represented as blocks of memory big enough to hold all fields.

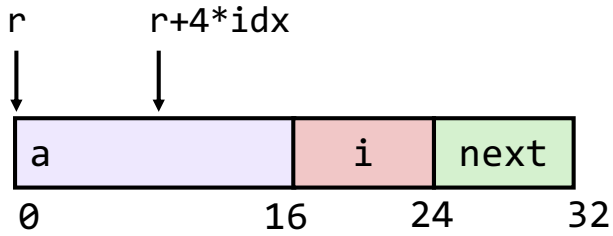
```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```



- Fields are ordered according to their declaration
- Compiler determines overall size and positions of fields
- In assembly, we only see offsets

Offset of each structure member is determined at compile time.

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```




```
1  int *get_ap
2   (struct rec *r, size_t idx)
3  {
4    return &r->a[idx];
5  }
```

```
1  # r in %rdi, idx in %rsi
2   leaq  (%rdi,%rsi,4), %rax
3   ret
```

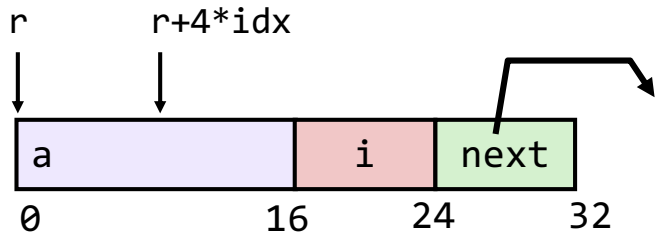
Generating pointer to array element:

- Compute as `r + 4*idx`

Example: Following Linked List

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```

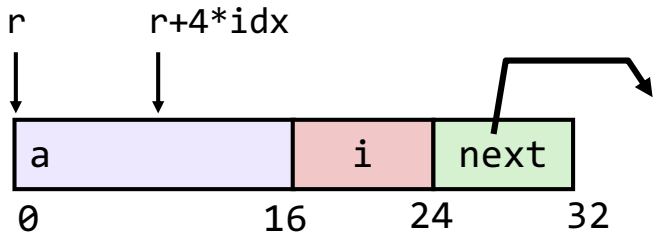
```
1  long length(struct rec *r) {
2      long len = 0L;
3      while (r) {
4          len++;
5          r = r->next;
6      }
7      return len;
8  }
```



Example: Following Linked List

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```

```
1  long length(struct rec *r) {
2      long len = 0L;
3      while (r) {
4          len++;
5          r = r->next;
6      }
7      return len;
8  }
```



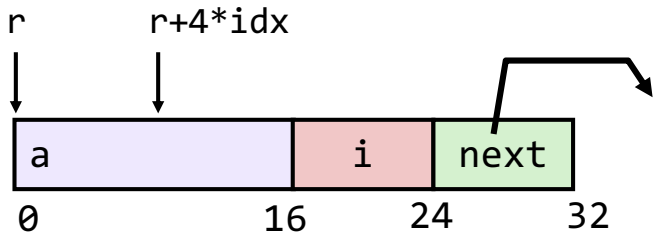

```
1  .L11:                          # loop:
2    addq    $1, %rax             #   len++
3    movq    24(%rdi), %rdi       #   r = Mem[r+24]
4    testq   %rdi, %rdi           #   Test r
5    jne     .L11                 #   If != 0, goto loop
```

Registers	
%rdi	r
%rax	len

Example: Following Linked List

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```

```
1  long length(struct rec *r) {
2      long len = 0L;
3      while (r) {
4          len++;
5          r = r->next;
6      }
7      return len;
8  }
```



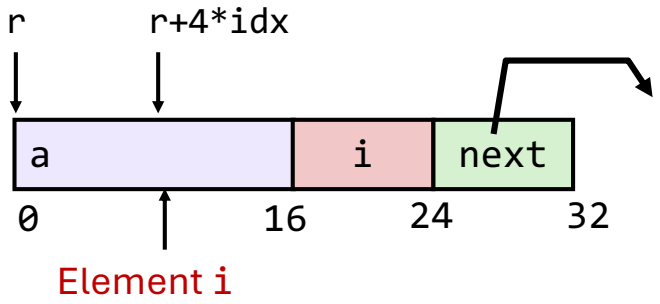

```
1  .L11:                          # loop:
2    addq    $1, %rax             #   len++
3    movq    24(%rdi), %rdi       #   r = Mem[r+24]
4    testq   %rdi, %rdi           #   Test r
5    jne     .L11                 #   If != 0, goto loop
```

Registers	
%rdi	r
%rax	len

Example: Following Linked List

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```

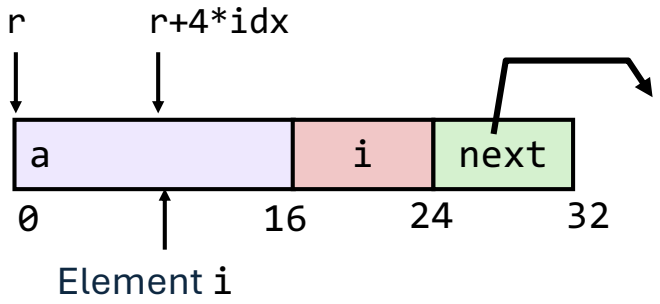
```
1  void set_val
2    (struct rec *r, int val)
3  {
4    while (r) {
5      size_t i = r->i;
6      // No bounds check
7      r->a[r->i] = val;
8      r = r->next;
9    }
10 }
```



Example: Following Linked List

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```

```
1  void set_val
2    (struct rec *r, int val)
3  {
4    while (r) {
5      size_t i = r->i;
6      // No bounds check
7      r->a[r->i] = val;
8      r = r->next;
9    }
10 }
```



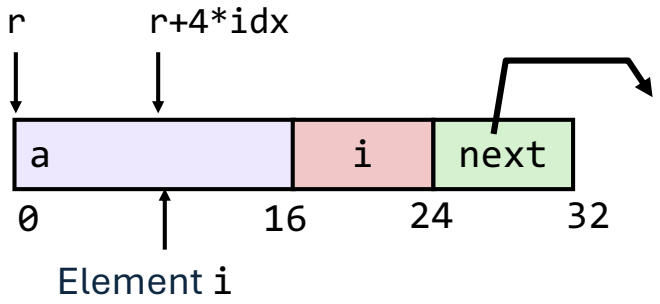

```
1  .L11:                          # loop:
2    movq   16(%rdi), %rax        #   i = Mem[r+16]
3    movl   %esi, (%rdi,%rax,4)   #   Mem[r+4*i] = val
4    movq   24(%rdi), %rdi        #   r = Mem[r+24]
5    testq %rdi, %rdi             #   Test r
6    jne    .L11                  #   if !=0 goto loop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

Registers	
%rdi	r
%rsi	val

Example: Following Linked List

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```

```
1  void set_val
2    (struct rec *r, int val)
3  {
4    while (r) {
5      size_t i = r->i;
6      // No bounds check
7      r->a[r->i] = val;
8      r = r->next;
9    }
10 }
```



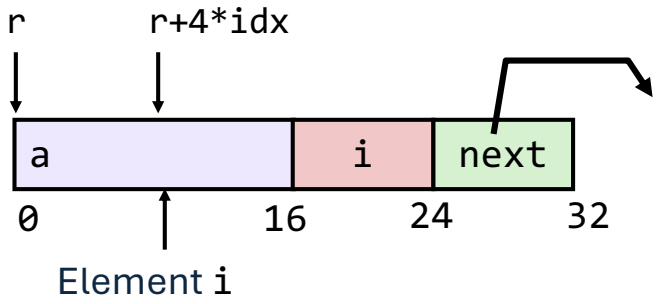

```
1  .L11:                          # loop:
2    movq  16(%rdi), %rax         #   i = Mem[r+16]
3    movl  %esi, (%rdi,%rax,4)    #   Mem[r+4*i] = val
4    movq  24(%rdi), %rdi         #   r = Mem[r+24]
5    testq %rdi, %rdi             #   Test r
6    jne   .L11                   #   if !=0 goto loop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

Registers	
%rdi	r
%rsi	val

Example: Following Linked List

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```

```
1  void set_val
2    (struct rec *r, int val)
3  {
4    while (r) {
5      size_t i = r->i;
6      // No bounds check
7      r->a[r->i] = val;
8      r = r->next;
9    }
10 }
```



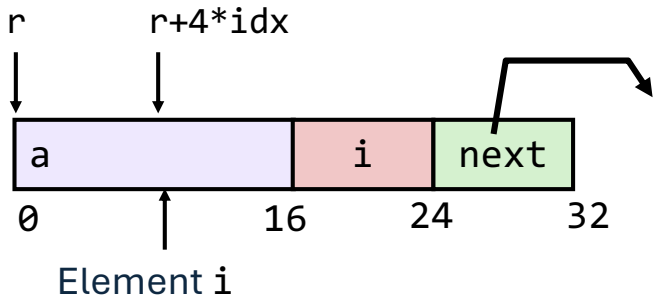

```
1  .L11:                          # loop:
2    movq  16(%rdi), %rax         #   i = Mem[r+16]
3    movl  %esi, (%rdi,%rax,4)    #   Mem[r+4*i] = val
4    movq  24(%rdi), %rdi         #   r = Mem[r+24]
5    testq %rdi, %rdi             #   Test r
6    jne   .L11                   #   if !=0 goto loop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

Registers	
%rdi	r
%rsi	val

Example: Following Linked List

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```

```
1  void set_val
2    (struct rec *r, int val)
3  {
4    while (r) {
5      size_t i = r->i;
6      // No bounds check
7      r->a[r->i] = val;
8      r = r->next;
9    }
10 }
```



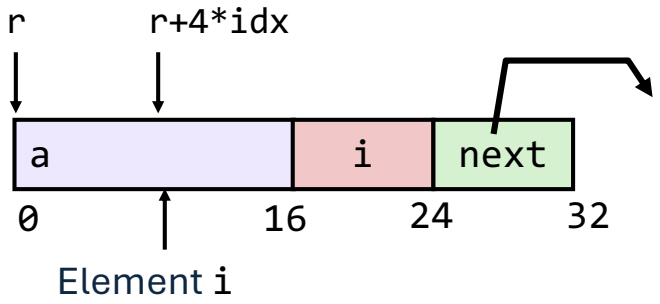

```
1  .L11:                          # loop:
2    movq   16(%rdi), %rax        #   i = Mem[r+16]
3    movl   %esi, (%rdi,%rax,4)   #   Mem[r+4*i] = val
4    movq   24(%rdi), %rdi        #   r = Mem[r+24]
5    testq %rdi, %rdi             #   Test r
6    jne    .L11                  #   if !=0 goto loop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

Registers	
%rdi	r
%rsi	val

Example: Following Linked List

```
1  struct rec {
2      int a[4];
3      size_t i;
4      struct rec *next;
5  };
```

```
1  void set_val
2    (struct rec *r, int val)
3  {
4    while (r) {
5      size_t i = r->i;
6      // No bounds check
7      r->a[r->i] = val;
8      r = r->next;
9    }
10 }
```

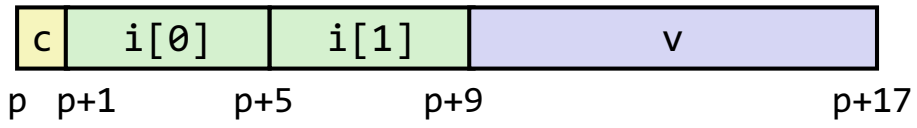



```
1  .L11:                          # loop:
2    movq   16(%rdi), %rax        #   i = Mem[r+16]
3    movl   %esi, (%rdi,%rax,4)   #   Mem[r+4*i] = val
4    movq   24(%rdi), %rdi        #   r = Mem[r+24]
5    testq %rdi, %rdi             #   Test r
6    jne    .L11                  #   if !=0 goto loop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

Registers	
%rdi	r
%rsi	val

Data is aligned if its address is a multiple of its size.

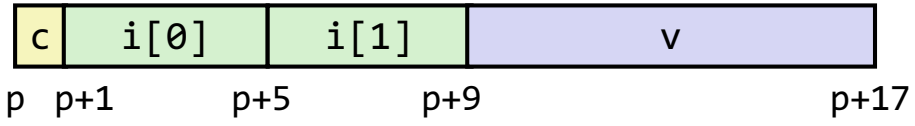
Unaligned data:




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

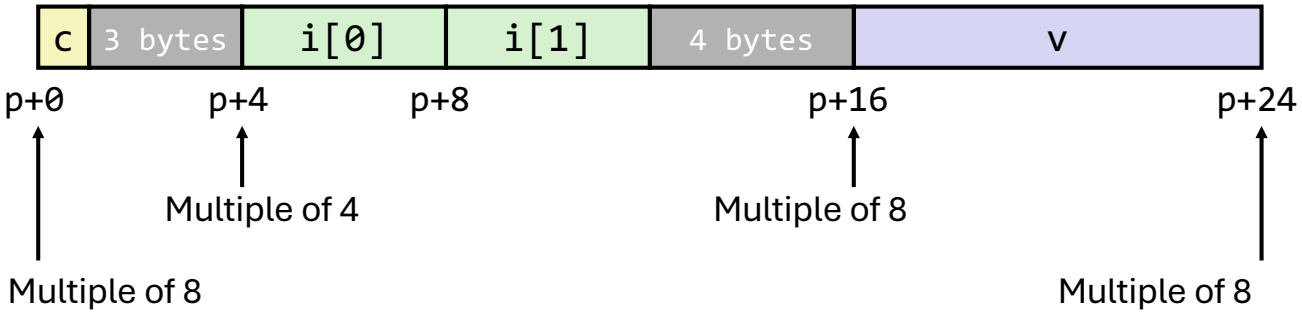
Data is aligned if its address is a multiple of its size.

Unaligned data:




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

Aligned data:



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 10.]
    - Virtual memory trickier when datum spans 2 pages (4 KB pages)

      [Virtual memory pages will be discussed in Lecture 11.]
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

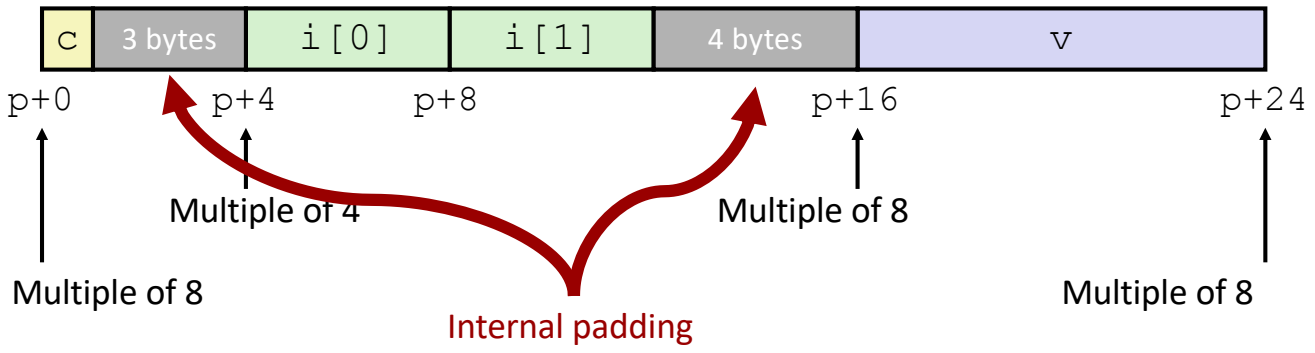
Specific Cases of Alignment (x86-64)

- 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

- 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
    NOTE: K < sizeof(struct S1)

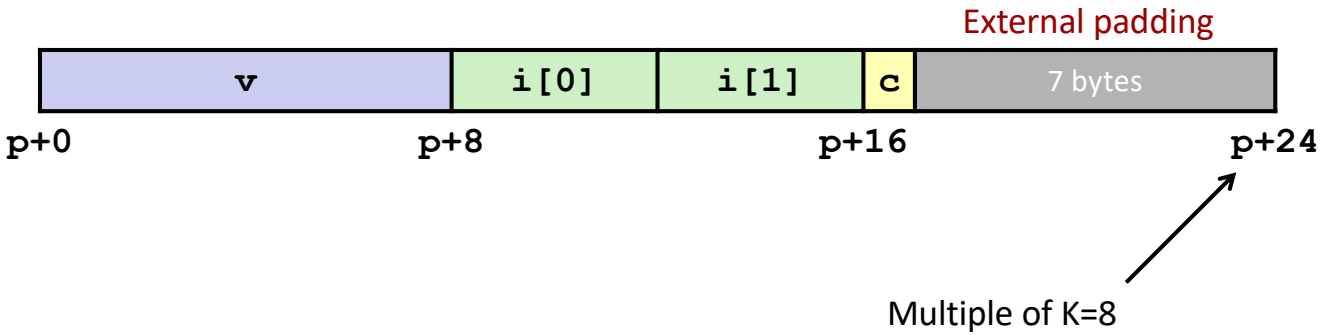
```
1  struct S1 {
2      char c;
3      int i[2];
4      double v;
5  } *p;
```



Meeting Overall Alignment Requirement

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

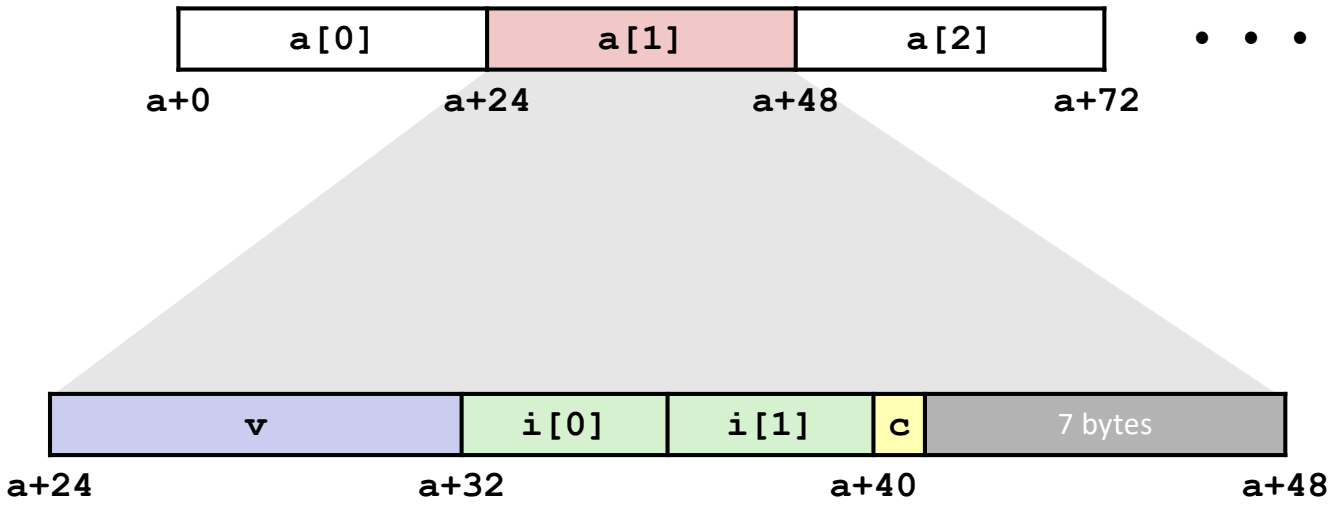
```
1  struct S2 {
2    double v;
3    int i[2];
4    char c;
5  } *p;
```



Arrays of Structures

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

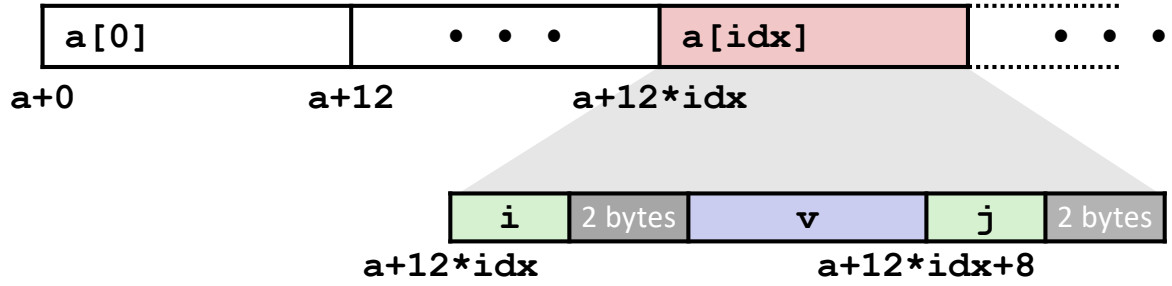
- Overall structure length multiple of K
- Satisfy alignment requirement for every element
- No other padding in between array elements



Accessing Array Elements

```
1  struct S3 {
2    short i;
3    float v;
4    short j;
5  } 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



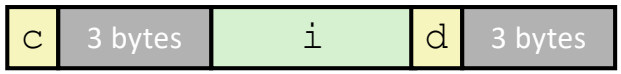

```
1  short get_j(int idx)
2  {
3    return a[idx].j;
4  }
```

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

Saving Space

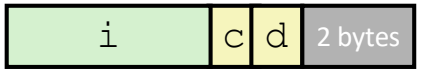
- Put large data types first
- Effect (largest alignment requirement K=4)

```
1  struct S4 {
2    char c;
3    int i;
4    char d;
5  } *p;
```



12 bytes

```
1  struct S5 {
2    int i;
3    char c;
4    char d;
5  } *p;
```



8 bytes

Today: How does x86-64 implement C arrays and structures?

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

FP Basics

- Arguments passed in `%xmm0`, `%xmm1`, …
- Result returned in `%xmm0`
- All XMM registers are call-clobbered

```
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
```

Summary

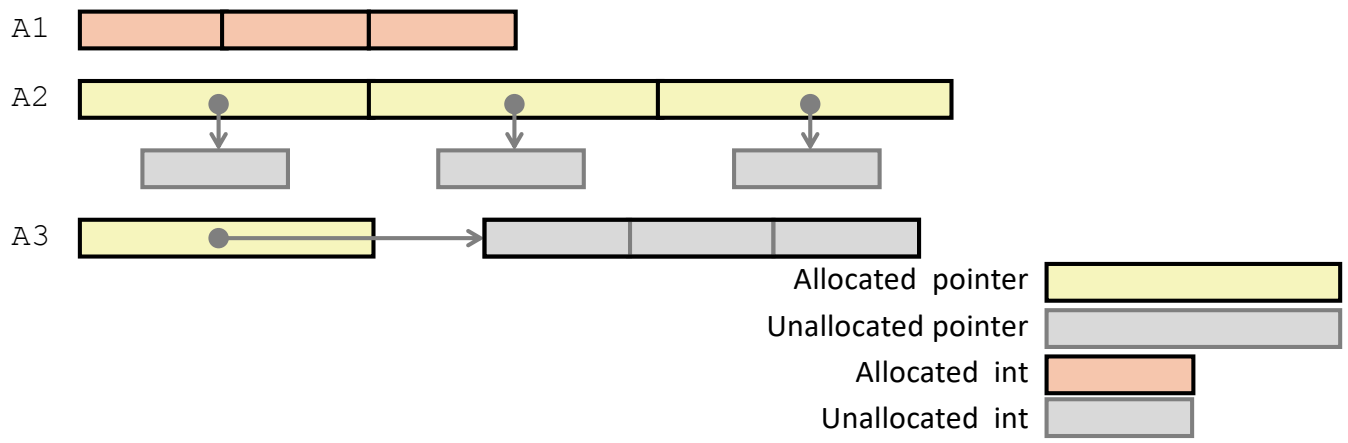
- Arrays
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements
- Structures
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment
- Combinations
  - Can nest structure and array code arbitrarily
- Floating Point
  - Data held and operated on in XMM registers

Optional activity on today's lecture is available on course schedule page

Additional slides

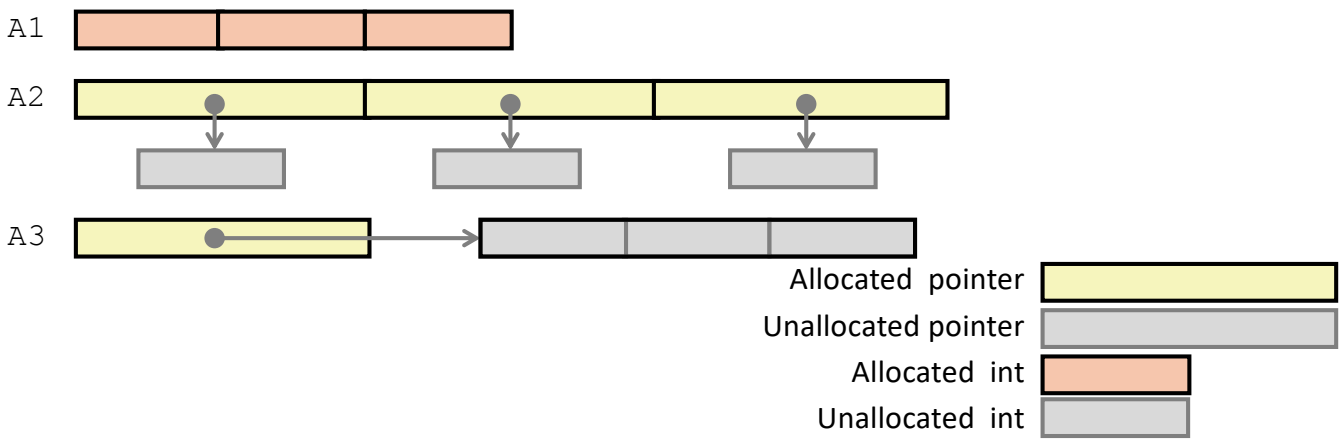
Understanding Pointers & Arrays #2

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



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



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
```

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
```

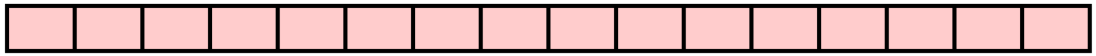
Background

- 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

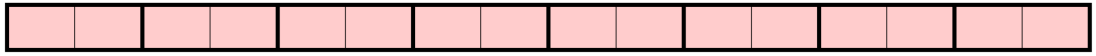
Programming with SSE4

XMM Registers

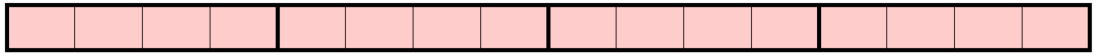
- 16 total, each 16 bytes
- 16 single-byte integers



- 8 16-bit integers



- 4 32-bit integers

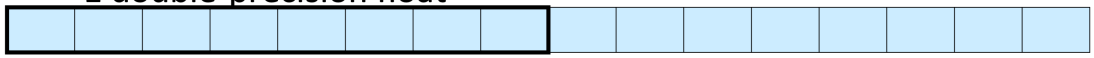


- 4 single-precision floats

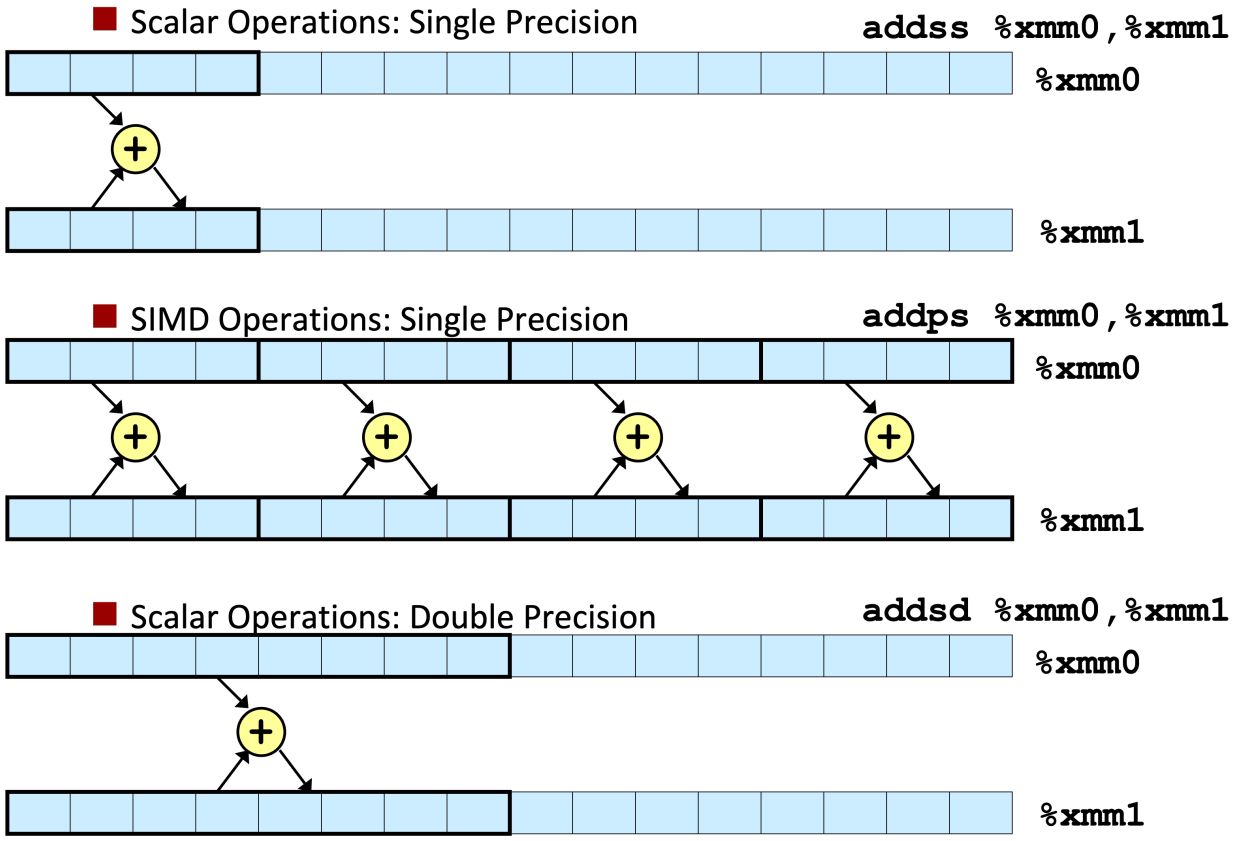
- 2 double-precision floats

- 1 single-precision float

- 1 double-precision float



Scalar & SIMD Operations



Other Aspects of FP Code

- *Lots* of instructions
  - Different operations, different formats, …
- Floating-point comparisons
  - Instructions **`ucomiss`** and **`ucomi`**
  - 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