



**HARVARD**

School of Engineering  
and Applied Sciences

# Machine Programming 4: Structured Data

*CS61, Lecture 6*

Prof. Stephen Chong

September 20, 2011

# Announcements

- Assignment 2 (Binary bomb) due Thursday
- We are trying out Piazza to allow class-wide questions and discussions
  - Go to <http://piazza.com/harvard/fall2011/cs61> to sign up and join in
- Google form to help you form a study group or find a partner
  - See course web page for links
  - Welcome to discuss problems with your classmates, and to collaborate in planning and thinking through solutions.
  - For Assignments 1–3, graded individually
  - For Assignments 4–6, allowed to work in pairs; pair shares grade
  - More info on Course Policies page

# Topics for today

- Structured data
  - Arrays
  - Multidimensional arrays
  - Multi-level arrays
  - Structs
  - Memory alignment
  - Arrays of structs

# Basic data types

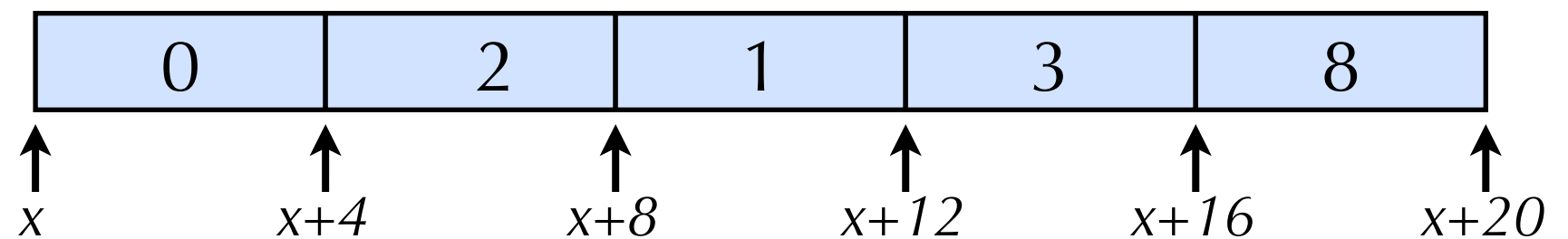
- Integer data types
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used
- Floating Point
  - Stored & operated on in floating point registers

C declaration	Intel data type	Assembly code suffix	32-bit	64-bit
char	Byte	b	1	1
short int	Word	w	2	2
int	Double word	l	4	4
long int	Quad word	q	4	8
float	Single precision	s	4	4
double	Double precision	d	8	8
long double	Extended precision	t	10/12	10/16

# Array allocation

- $T\ A[L];$ 
  - Declares array of data type  $T$  and length  $L$
  - e.g., `int foo[5]`
- Represented as contiguously allocated region of  $L \times \text{sizeof}(T)$  bytes

```
int foo[5];  
foo[0] = 0;  
foo[1] = 2;  
foo[2] = 1;  
foo[3] = 3;  
foo[4] = 8;
```

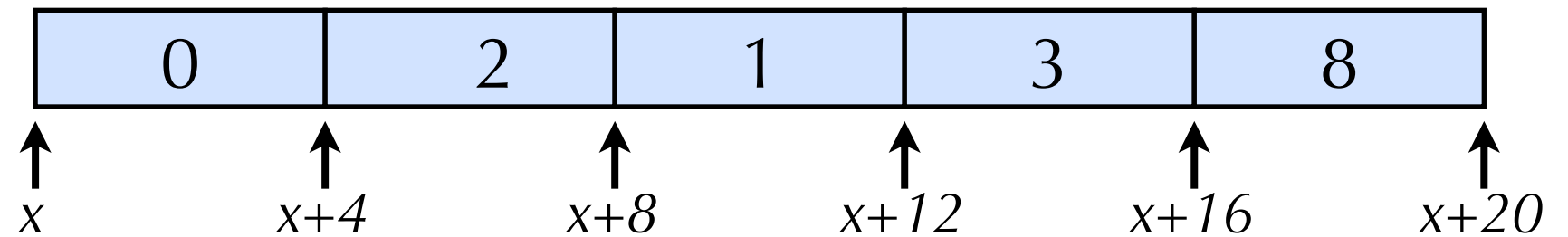


# Array access

- Given  $T\ A[L]$  ;
  - $A[i]$  accesses the  $i$ th element of the array
  - Identifier  $A$  can be used as a pointer to array element 0
    - Type of  $A$  is  $T *$
    - E.g., `int foo[5];`  
`*foo = 4;` is equivalent to `foo[0] = 4;`

# Array notation

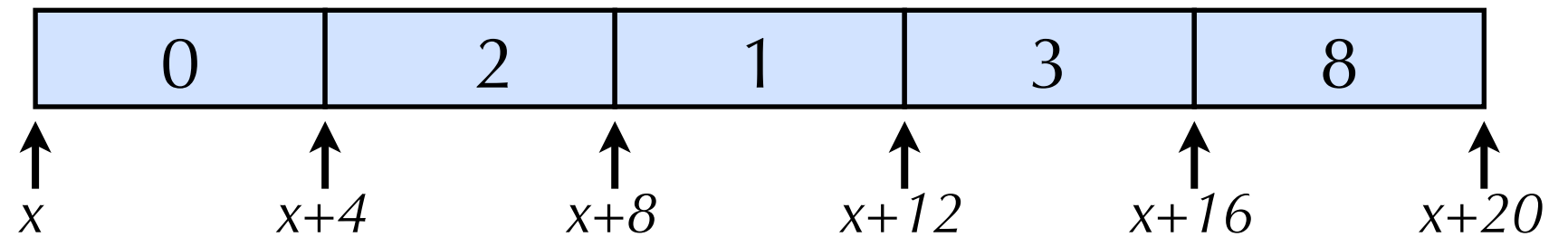
```
int foo[5];
```



Expression	Type	Value
<code>foo[2]</code>		
<code>foo</code>		
<code>foo+1</code>		
<code>&amp;foo[2]</code>		
<code>foo[5]</code>		
<code>*(foo+1)</code>		
<code>foo + 3</code>		

# Array notation

`int foo[5];`



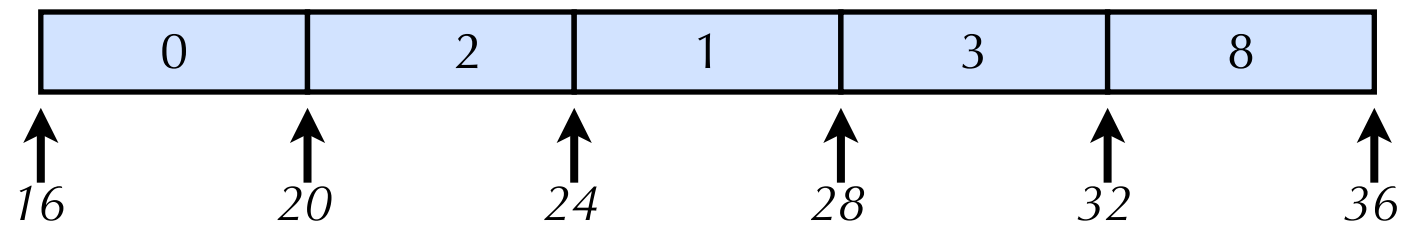
Expression	Type	Value
<code>foo[2]</code>	<code>int</code>	1
<code>foo</code>	<code>int *</code>	$x$
<code>foo+1</code>	<code>int *</code>	$x+4$
<code>&amp;foo[2]</code>	<code>int *</code>	$x+8$
<code>foo[5]</code>	<code>int</code>	???
<code>*(foo+1)</code>	<code>int</code>	2
<code>foo + 3</code>	<code>int *</code>	$x + 4 \times 3$



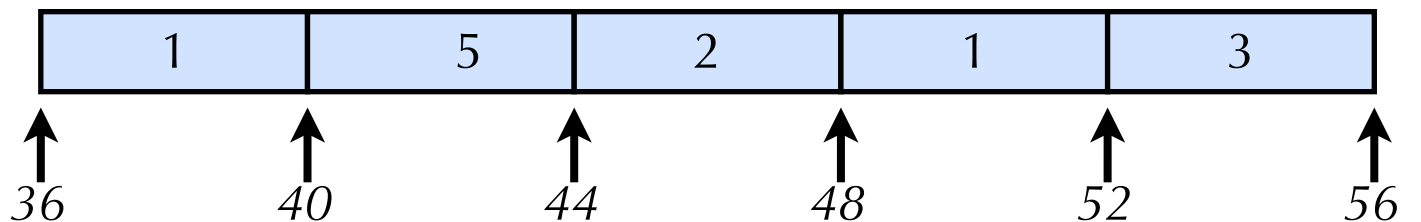
# Array example

```
typedef int zip_dig[5];  
  
zip_dig hvd = { 0, 2, 1, 3, 8 };  
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig cor = { 1, 4, 8, 5, 3 };
```

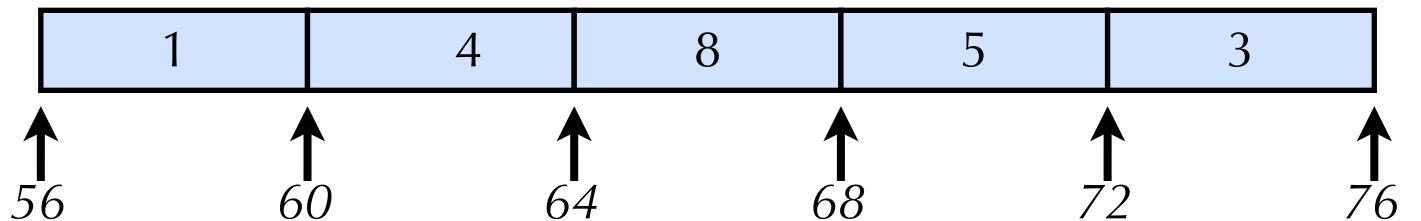
zip\_dig hvd;



zip\_dig cmu;



zip\_dig cor;



- Note: declaration `zip_dig hvd` equivalent to `int hvd[5]`
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to be true!

# Accessing arrays

```
typedef int zip_dig[5];

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

```
push    %ebp
mov     %esp,%ebp

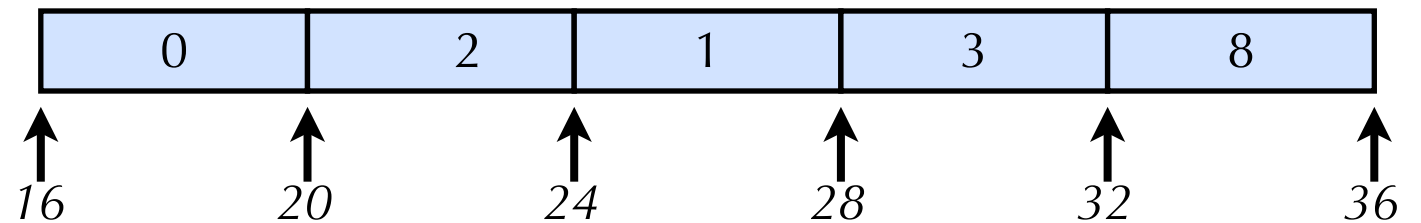
mov     0xc(%ebp),%eax      # %eax = dig
mov     0x8(%ebp),%edx      # %edx = z
mov     (%edx,%eax,4),%eax  # %eax = z[dig]

pop     %ebp
ret
```

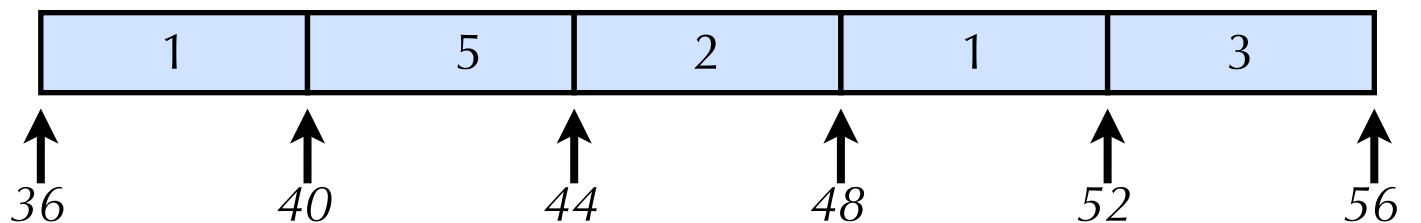
- **%edx** contains starting address of array
- **%eax** contains the array index
- Desired value at **%edx + (%eax \* 4)**

# Referencing examples

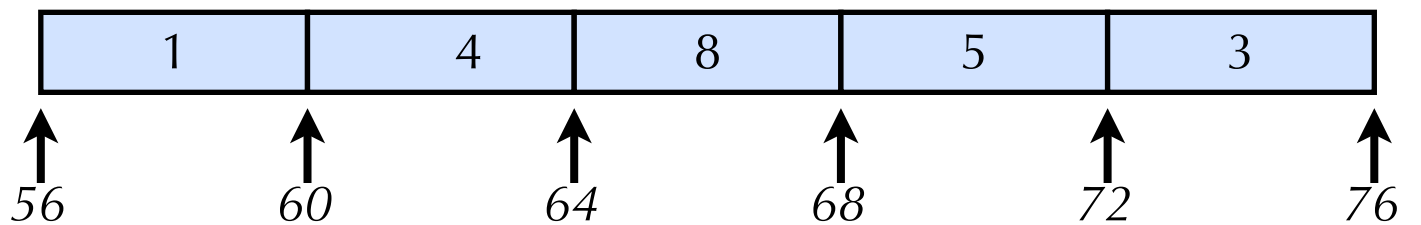
zip\_dig hvd;



zip\_dig cmu;



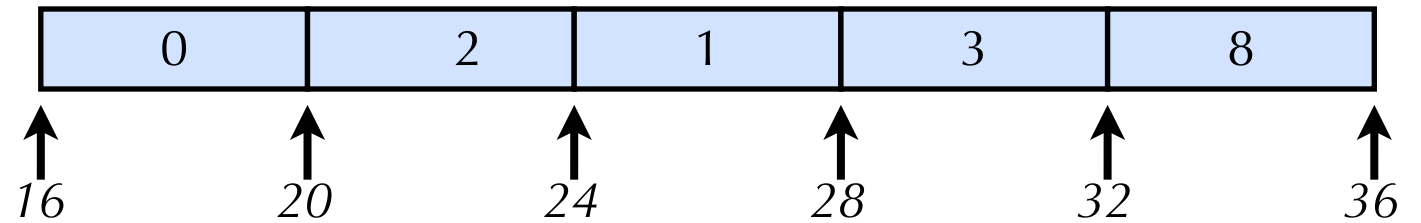
zip\_dig cor;



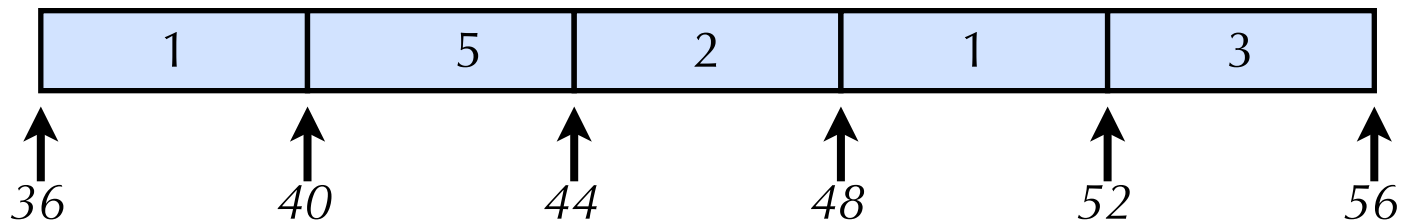
Expression	Address	Value	Guaranteed to work?
cmu[3]			
cmu[5]			
cmu[-1]			
hvd[15]			

# Referencing examples

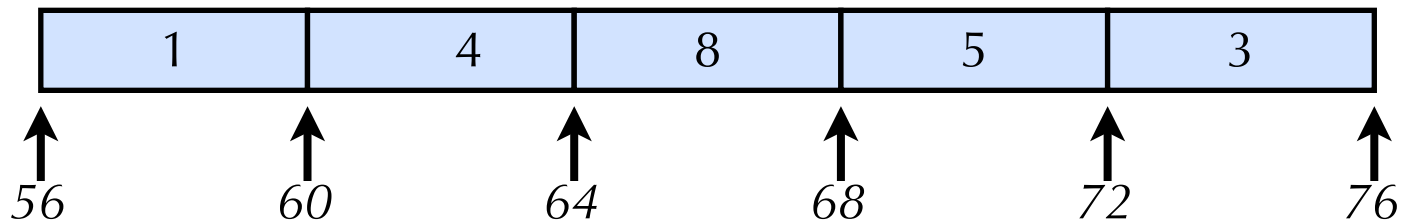
zip\_dig hvd;



zip\_dig cmu;



zip\_dig cor;



Expression	Address	Value	Guaranteed to work?
cmu[3]	$36 + 4 * 3 = 48$	1	Yes
cmu[5]	$36 + 4 * 5 = 56$	1	No
cmu[-1]	$36 + 4 * -1 = 32$	8	No
hvd[15]	$16 + 4 * 15 = 76$	??	No

# Looping over an array

- Original source

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

- What gcc produces

- (pseudocode)
- Eliminate loop variable *i*
- Convert array code to pointer code
- Rewrite as do-while loop

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

# Looping over array in machine code

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

- Notes:

- `xorl %eax,%eax` sets `%eax` to zero
- `leal` used for arithmetic
- 4 added to `%ecx` each iteration

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

# Strings

- x86 doesn't have an internal notion of a string datatype.
- In C, a string is represented as an array of char, terminated by a 0 byte
  - Each character is one ASCII encoded byte
  - E.g., "this is a string"

0x74	0x68	0x69	0x73	0x20	0x69	0x73	0x20	0x61	0x20	0x73	0x74	0x72	0x69	0x6e	0x67	0x00
------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

0x74 is ASCII encoding of 't'

0x0 terminates string

- Different languages use different string representations
  - E.g., Java stores strings in Unicode format (1-4 bytes per character)

# Topics for today

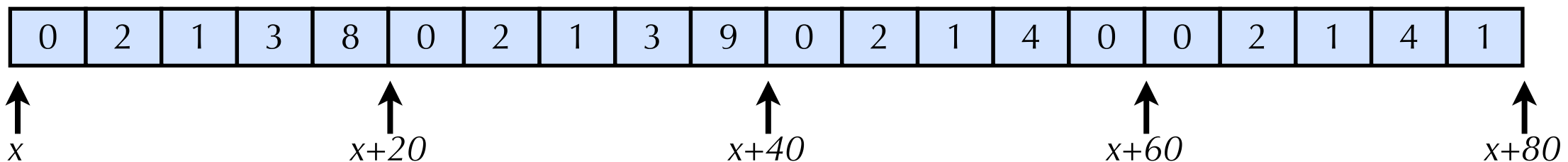
- Structured data
  - Arrays
  - Multidimensional arrays
  - Multi-level arrays
  - Structs
  - Memory alignment
  - Arrays of structs



# Multidimensional arrays

- $T\ A[R][C];$ 
  - Two dimensional array of datatype  $T$
  - $R$  rows and  $C$  columns
- Represented as contiguously allocated region of  $R \times C \times \text{sizeof}(T)$  bytes
  - Row major ordering: rows stored one after the other

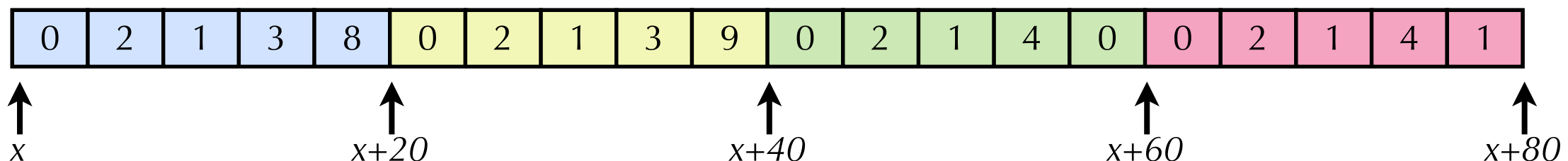
```
int cambridge_zips[4][5] =  
    {{0, 2, 1, 3, 8},  
     {0, 2, 1, 3, 9},  
     {0, 2, 1, 4, 0},  
     {0, 2, 1, 4, 1}};
```



# Multidimensional arrays

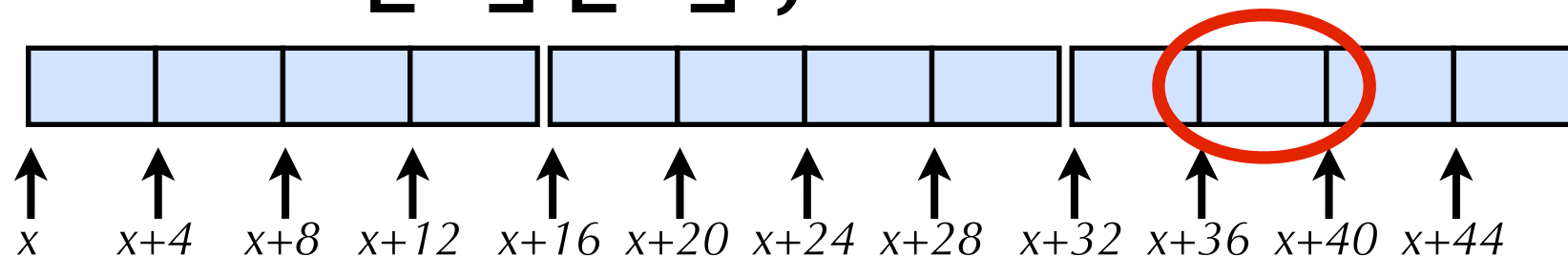
- `typedef int zip_dig[5];`
- `zip_dig cambridge_zips[4]`  
is equivalent to  
`int cambridge_zips[4][5]`
- `zip_dig cambridge_zips[4]`: array of 4 elements, allocated continuously
- Each element is an array of 5 ints, allocated continuously

```
int cambridge_zips[4][5] =  
    {{0, 2, 1, 3, 8},  
     {0, 2, 1, 3, 9},  
     {0, 2, 1, 4, 0},  
     {0, 2, 1, 4, 1}};
```



# Accessing a single array element

- $T\ A[R][C];$ 
  - $A[i][j]$  is a single element of type  $T$
  - Address is  $A + i * C * \text{sizeof}(T) + j * \text{sizeof}(T)$
- E.g., `int foo[3][4];`



- `&foo[2][1]` evaluates to
$$x + 2 * 4 * \text{sizeof}(\text{int}) + 1 * \text{sizeof}(\text{int})$$
$$= x + 32 + 4 = x + 36$$

# Machine code for array access

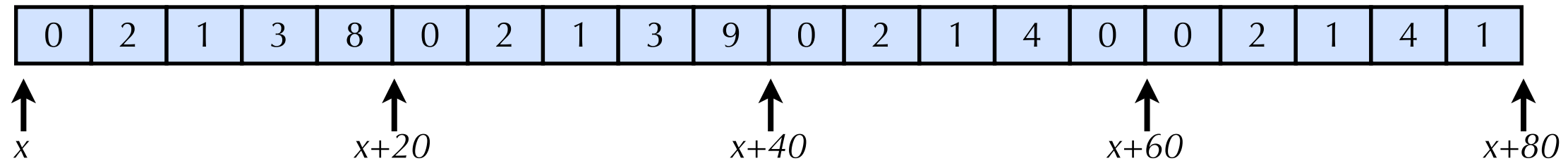
```
int cambridge_zips[4][5] = { ... };

int get_zip_digit(int index, int digit) {
    return cambridge_zips[index][digit];
}
```

```
...
# %ecx = digit
# %eax = index
leal 0(,%ecx,4),%edx          # %edx = 4*digit
leal (%eax,%eax,4),%eax       # %eax = 5*index
movl cambridge_zips(%edx,%eax,4),%eax  # return contents of
                                     # cambridge_zips + 4*digit
                                     #       + 4*5*index
...
```

# Strange referencing examples

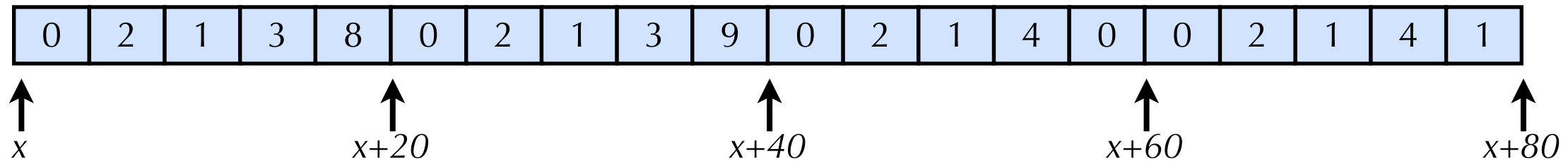
```
int zips[4][5];
```



Expression	Address	Value	Guaranteed to work?
<code>zips[3][3]</code>			
<code>zips[2][5]</code>			
<code>zips[2][-1]</code>			
<code>zips[4][-1]</code>			
<code>zips[0][19]</code>			
<code>zips[0][-1]</code>			

# Strange referencing examples

```
int zips[4][5];
```



Expression	Address	Value	Guaranteed to work?
<code>zips[3][3]</code>	$x + 20 \cdot 3 + 4 \cdot 3 = x+72$	4	Yes
<code>zips[2][5]</code>	$x + 20 \cdot 2 + 4 \cdot 5 = x+60$	0	Yes
<code>zips[2][-1]</code>	$x + 20 \cdot 2 + 4 \cdot -1 = x+36$	9	Yes
<code>zips[4][-1]</code>	$x + 20 \cdot 4 + 4 \cdot -1 = x+76$	1	Yes
<code>zips[0][19]</code>	$x + 20 \cdot 0 + 4 \cdot 19 = x+76$	1	Yes
<code>zips[0][-1]</code>	$x + 0 \cdot 2 + 4 \cdot -1 = x-4$	???	No

# Topics for today

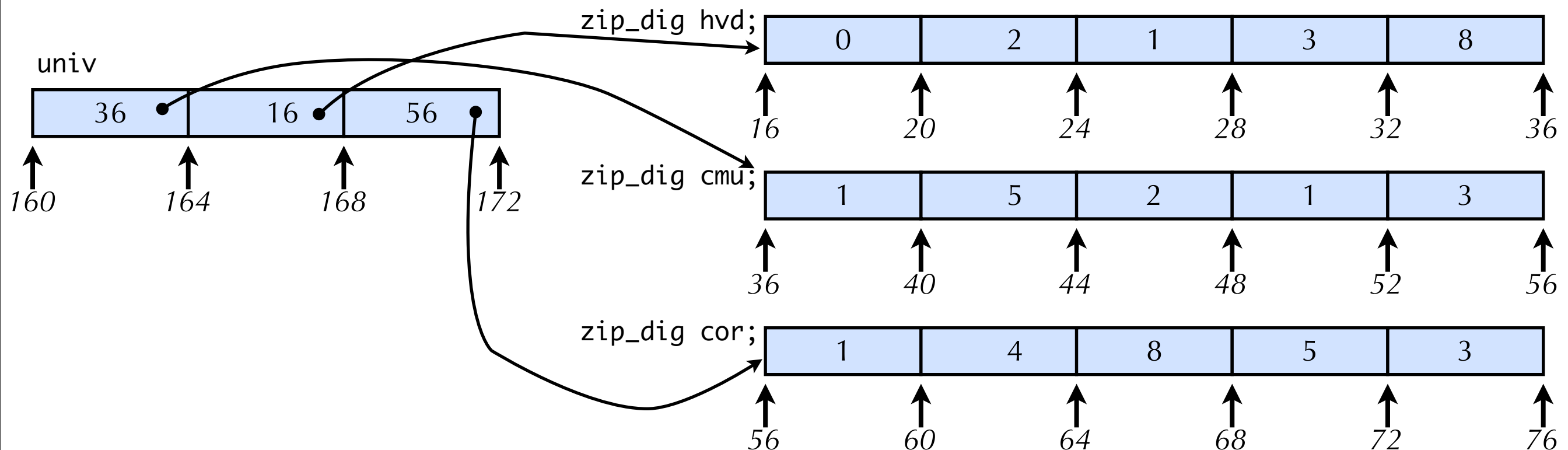
- Structured data
  - Arrays
  - Multidimensional arrays
  - Multi-level arrays
  - Structs
  - Memory alignment
  - Arrays of structs

# Multi-level array

```
typedef int zip_dig[5];  
  
zip_dig hvd = { 0, 2, 1, 3, 8 };  
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig cor = { 1, 4, 8, 5, 3 };
```

```
int *univ[3] = {cmu, hvd, cor};
```

- `univ` is array of 3 elements
- Each element is a pointer (4 bytes)
- Each pointer points to an array of `ints`



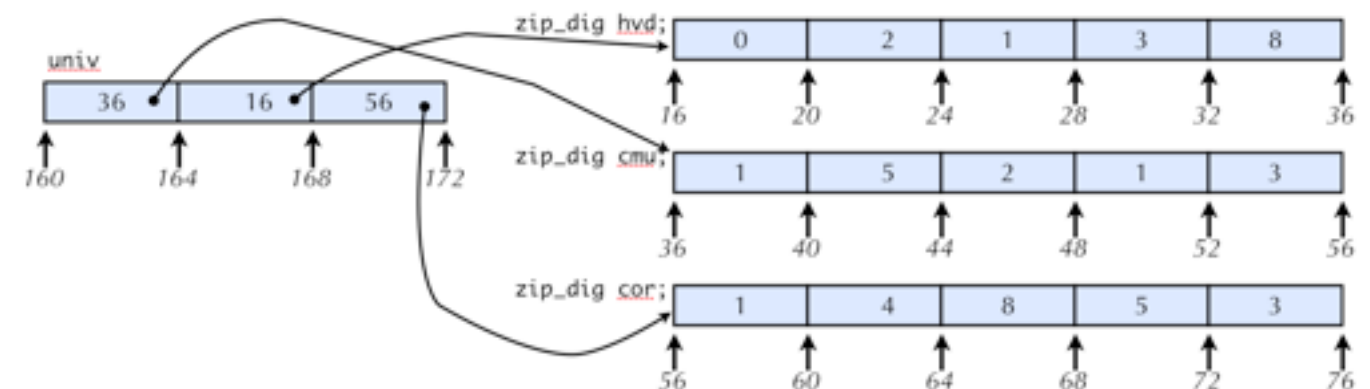
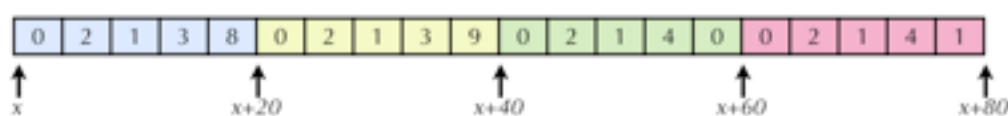


# Multilevel array element access

```
int cambridge_zips[4][5] = { ... };  
  
int get_digit(int index, int digit) {  
    return cambridge_zips[index][digit];  
}
```

```
int *univ[3] = {cmu, hvd, cor};  
  
int get_univ_digit  
    (int index, int dig)  
{  
    return univ[index][dig];  
}
```

- Access looks similar, but evaluation is quite different!



# Multilevel array element access

```
int cambridge_zips[4][5] = { ... };
```

```
int get_zip_digit(int index, int digit) {
    return cambridge_zips[index]...
```

One memory read

```
...
# %ecx = digit
# %eax = index
leal 0(,%ecx,4),%edx      # %edx = 4*digit
leal (%eax,%eax,4),%eax    # %eax = 5*index
movl cambridge_zips(%edx,%eax,4),%eax
    # return Mem[cambridge_zips + 20*index + 4*dig]
...
```

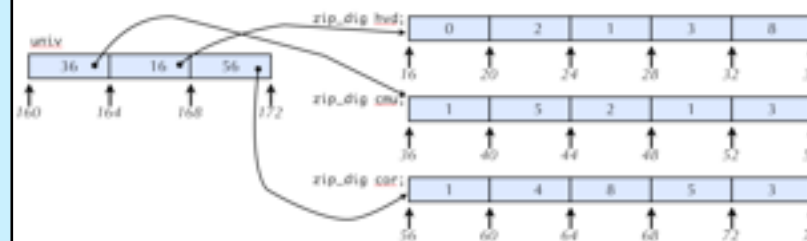


```
int *univ[3] = {cmu, hvd, cor};
```

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

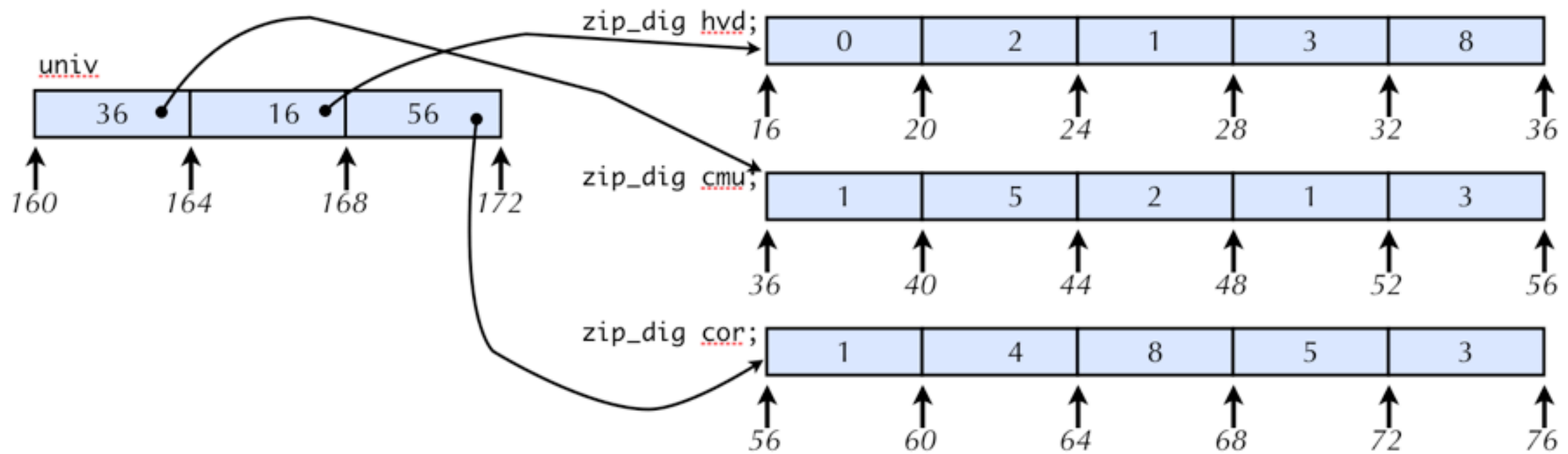
Two memory reads

- First get pointer to array
- Then access array



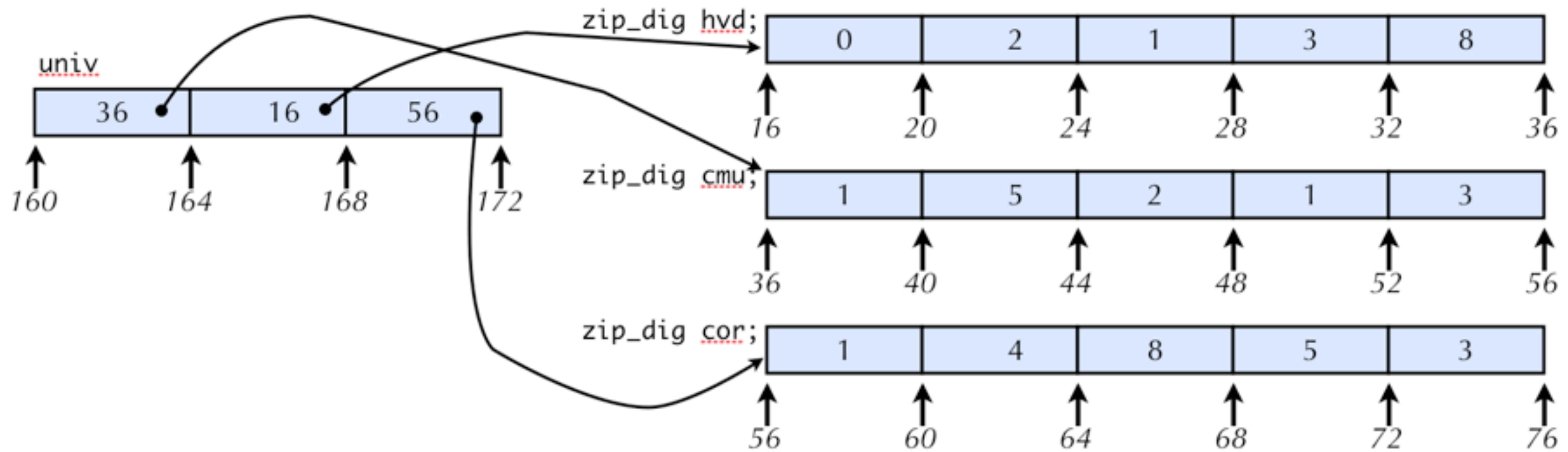
```
...
# %ecx = index
# %eax = dig
leal 0(,%ecx,4),%edx      # %edx = 4*index
movl univ(%edx),%edx      # %edx = Mem[univ + 4*index]
movl (%edx,%eax,4),%eax   # Mem[%edx + 4*dig]
...
```

# Strange referencing examples



Expression	Address	Value	Guaranteed to work?
univ[2][3]			
univ[1][5]			
univ[2][-1]			
univ[3][-1]			
univ[1][12]			

# Strange referencing examples



Expression	Address	Value	Guaranteed to work?
<code>univ[2][3]</code>	$56 + 4 * 3 = 68$	5	Yes
<code>univ[1][5]</code>	$16 + 4 * 5 = 36$	1	No
<code>univ[2][-1]</code>	$56 + 4 * -1 = 52$	3	No
<code>univ[3][-1]</code>	???	???	No
<code>univ[1][12]</code>	$16 + 4 * 12 = 64$	8	No

# Topics for today

- Structured data
  - Arrays
  - Multidimensional arrays
  - Multi-level arrays
  - Structs
  - Memory alignment
  - Arrays of structs

# Structs

- A struct groups objects into a single object
  - Each field accessed by name
  - Each field can have a different type

Declare a struct datatype

Use fields of a struct

Convenient way to initialize fields of struct

Equivalent

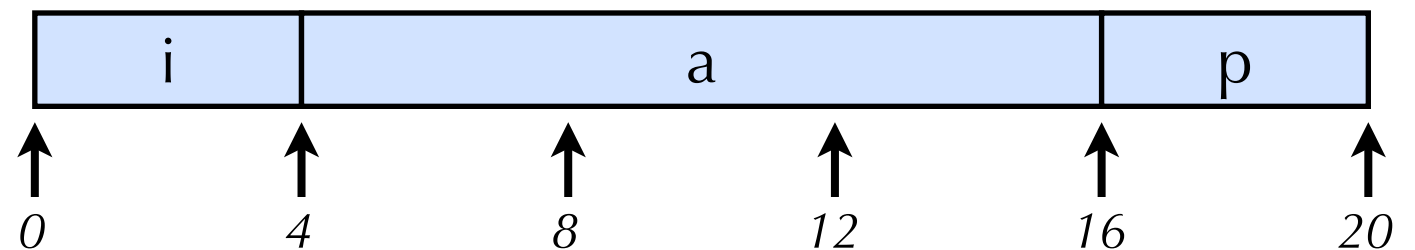
```
struct rect {  
    int llx;  
    int lly;  
    int width;  
    int height;  
    char *name;  
};  
  
struct rect r;  
r.llx = r.lly = 0;  
struct rect s =  
    {0, 0, 10, 20, "Rodney"};  
  
struct rect *rp = &s;  
rp->height = rp->width;  
(*rp).height = (*rp).width;
```

# Implementing structs

- Struct represented as contiguous region of memory

- E.g.,

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



- Compiler generates code using appropriate offsets

```
struct rec *myrec;  
int someint;  
  
void demo_struct() {  
    myrec->i = 42;  
    myrec->a[0] = 43;  
    myrec->p = &someint;  
}
```

```
demo_struct:  
    pushl %ebp  
    movl %esp, %ebp  
    movl myrec, %eax  
    movl $42, (%eax)  
    movl $43, 4(%eax)  
    movl $someint, 16(%eax)  
    popl %ebp  
    ret
```



# Memory alignment

- Many systems require data to be aligned in memory
  - E.g., Linux/x86 requires integers to be stored at memory address that is multiple of 4 bytes
- Why?
  - 32-bit machines typically read and write 4 bytes of memory at a time from **word aligned** addresses
  - If not aligned, reading an int from memory may require two memory accesses!



# Memory alignment

- Some processors **require** aligned access
  - If you try to access an “int” not stored on an aligned address, would get an “alignment fault”
- Intel x86 does **not require** memory access to be aligned
  - However, Intel strongly recommends that compilers generate code that uses aligned access, to get the best performance.
- Different OS's and compilers have different rules
  - Linux: 2-byte types aligned to 2 byte boundaries, 4 byte and larger types aligned to 4 byte boundaries
  - Windows: primitive object of  $K$  bytes must have an address that is a multiple of  $K$

# Alignment within structs

- Each field in struct must be aligned correctly
  - Compiler may have to insert **padding** within struct
- E.g., using Windows alignment rules
  - Field `i` must be aligned on 4 byte address
  - Field `v` must be aligned on 8 byte address

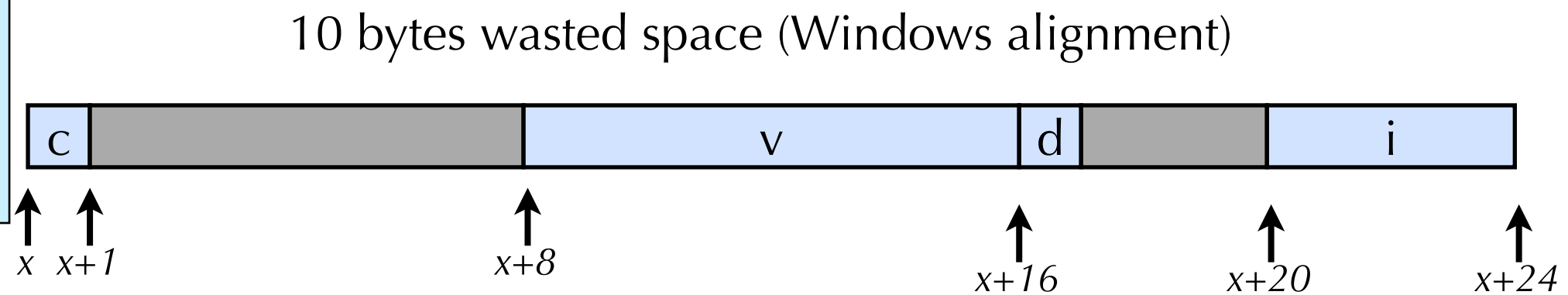
```
struct S {  
    char c;  
    int i[2];  
    double v;  
};
```



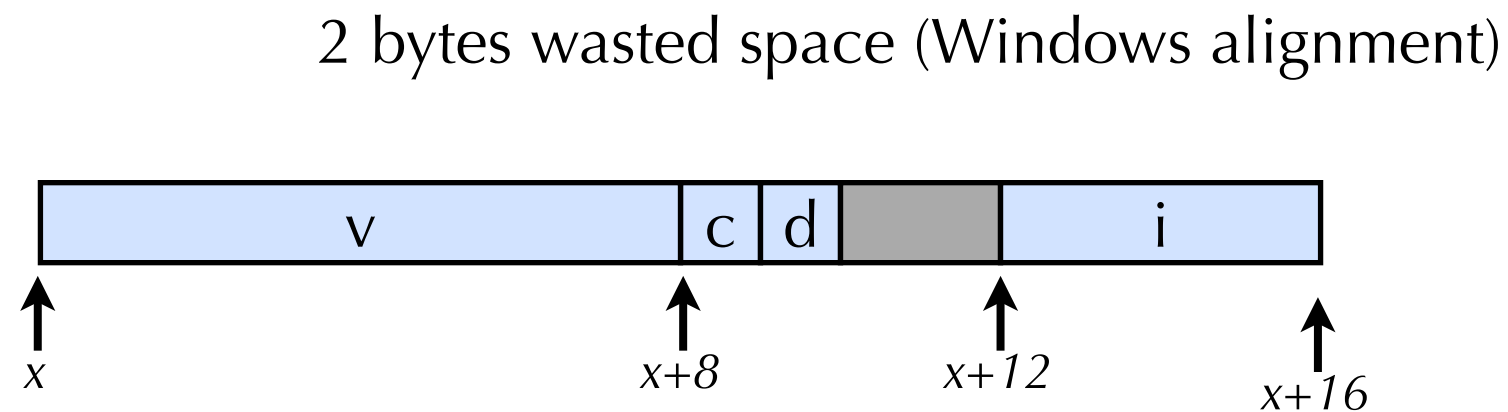
- `sizeof(struct S)` is 24, not  $1+4+4+8 = 17$

# Save memory!

```
struct T1 {  
    char c;  
    double v;  
    char d;  
    int i;  
};
```



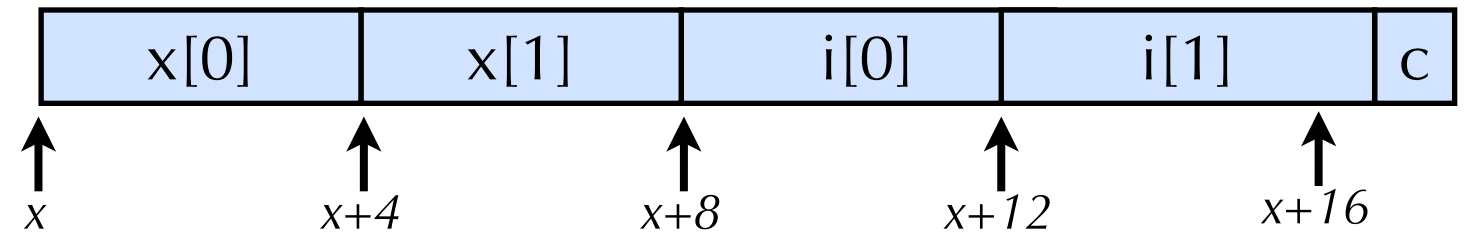
```
struct T2 {  
    double v;  
    char c;  
    char d;  
    int i;  
};
```



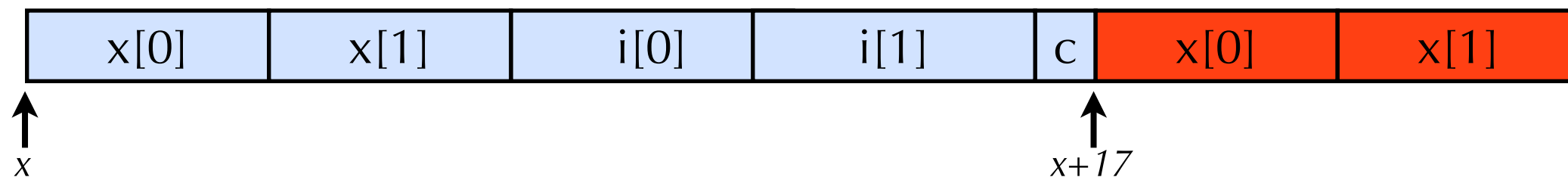
# Alignment of whole structs

• E.g.,

```
struct U {  
    float x[2];  
    int i[2];  
    char c;  
};
```



- No padding needed within struct
- But what about `struct U foo[2]` ?



Misaligned!

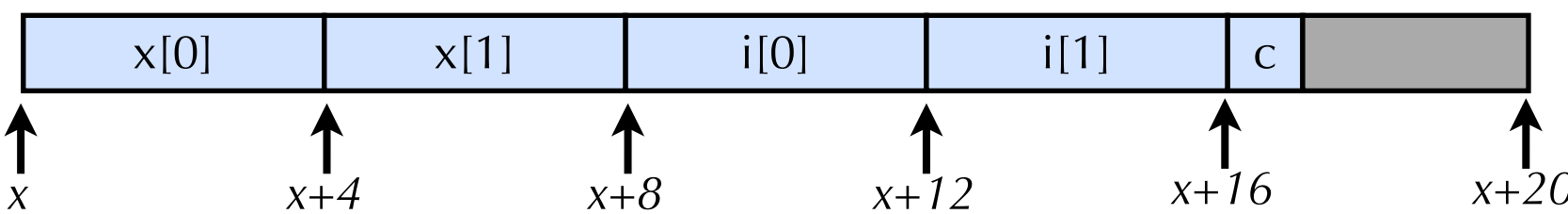
# Padding at end of struct

- To ensure that arrays of structs are correctly aligned, may need to pad at end of struct

- E.g.,

```
struct U {  
    float x[2];  
    int i[2];  
    char c;  
};
```

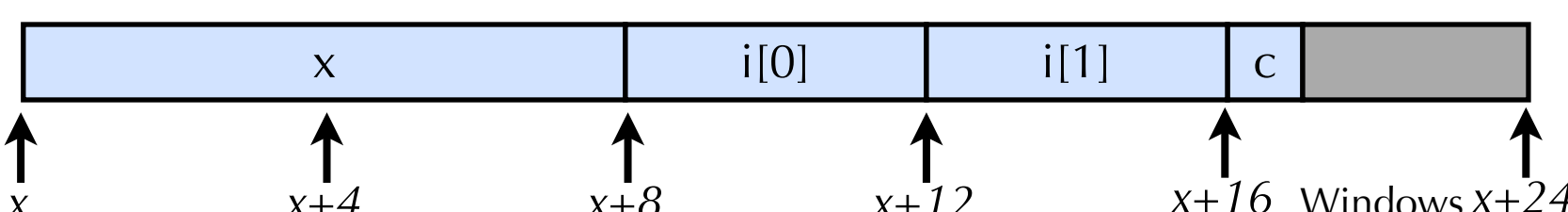
sizeof(U) must be multiple of 4 (Windows and Linux)



- E.g.,

```
struct V {  
    double x;  
    int i[2];  
    char c;  
};
```

sizeof(V) must be multiple of 8 for Windows, 4 for Linux



# Next lecture

- Buffer overruns and stack exploits
  - B3 l33t H4x0r and pwnzorz the interwebz
  - (Use your powers wisely)