

15-213

Machine-Level Programming IV: Data September 8, 2008

Structured Data

- Arrays
- Structs
- Unions

Data/Control

- Buffer overflow

Basic Data Types

Integral

- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

Floating Point

- Stored & operated on in floating point registers

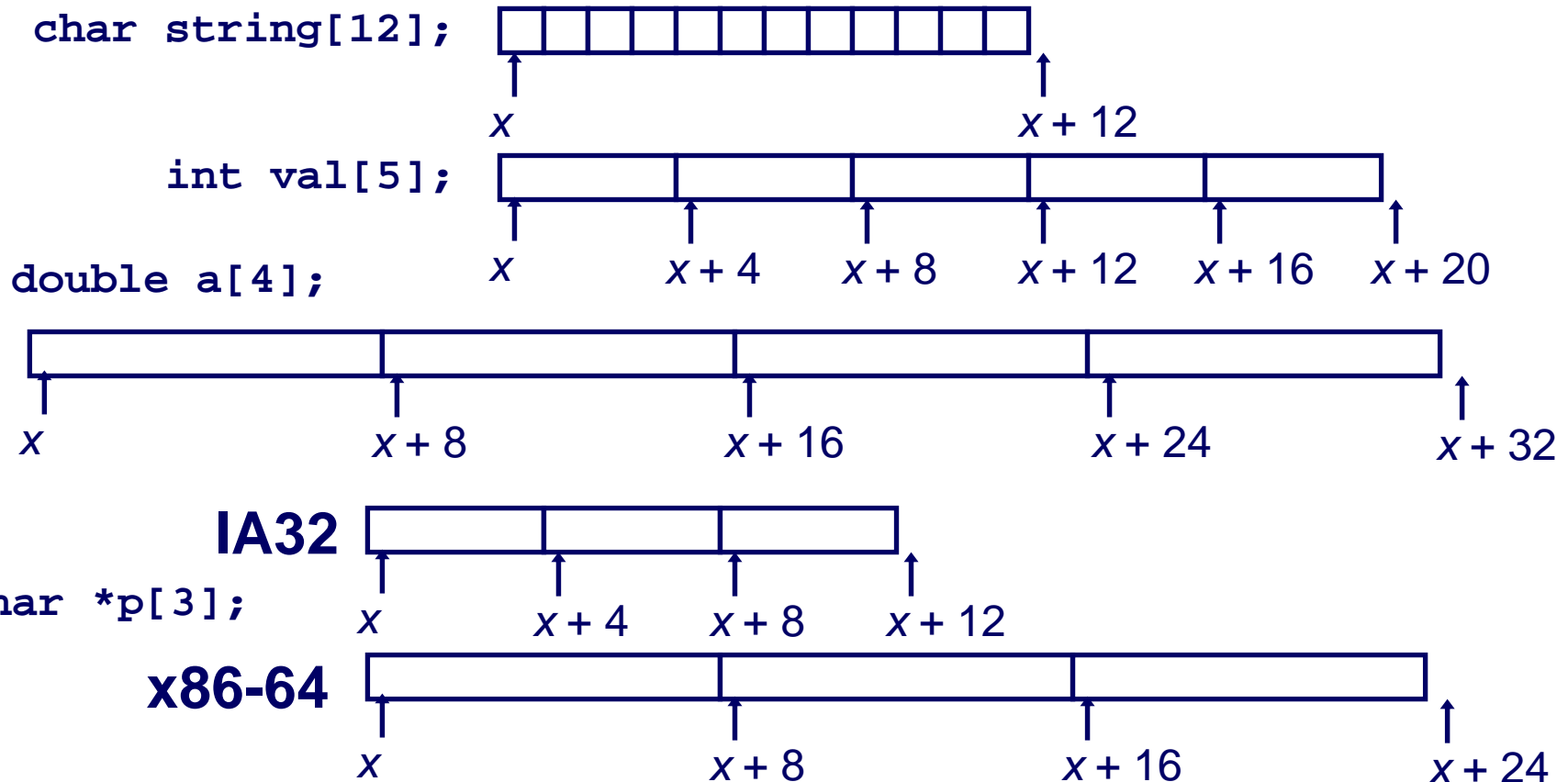
Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12/16	long double

Array Allocation

Basic Principle

`T A[L];`

- Array of data type T and length L
- Contiguously allocated region of $L * \text{sizeof}(T)$ bytes

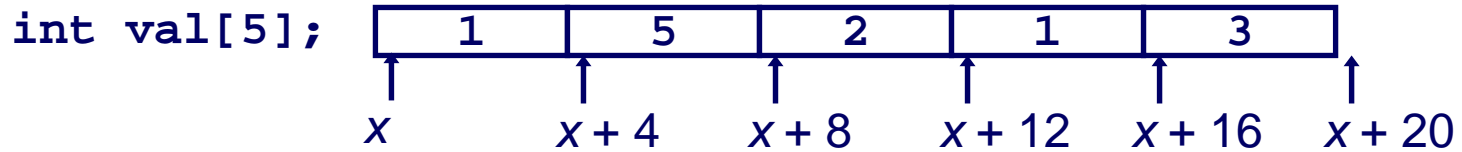


Array Access

Basic Principle

T $A[L]$;

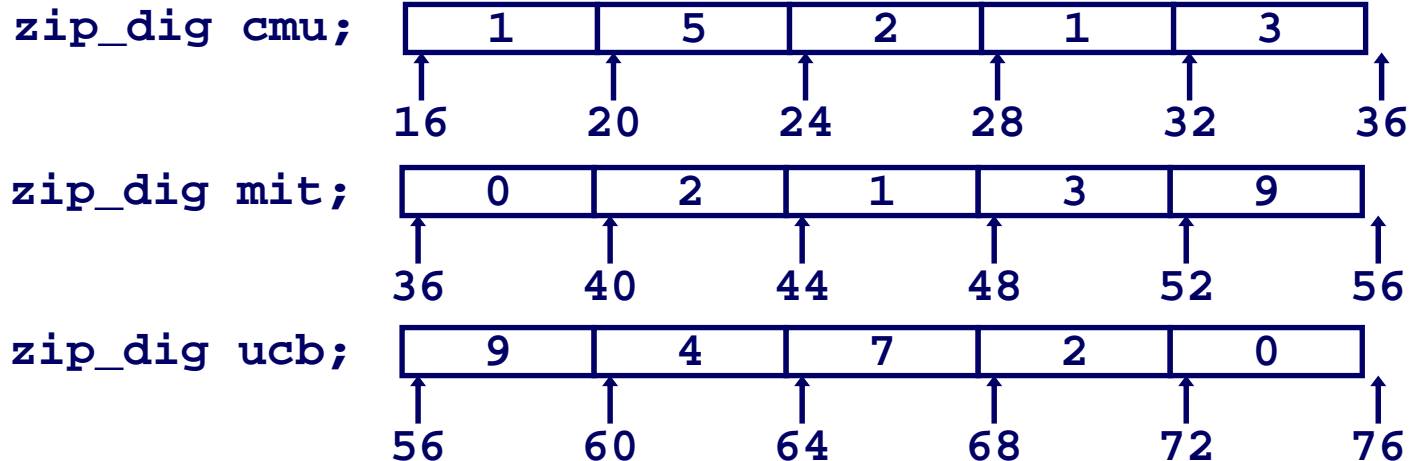
- Array of data type T and length L
- Identifier A can be used as a pointer to array element 0
 - Type T^*



Reference	Type	Value
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	x
<code>val+1</code>	<code>int *</code>	$x+4$
<code>&val[2]</code>	<code>int *</code>	$x+8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	$x+4i$

Array Example

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



Notes

- 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

Computation

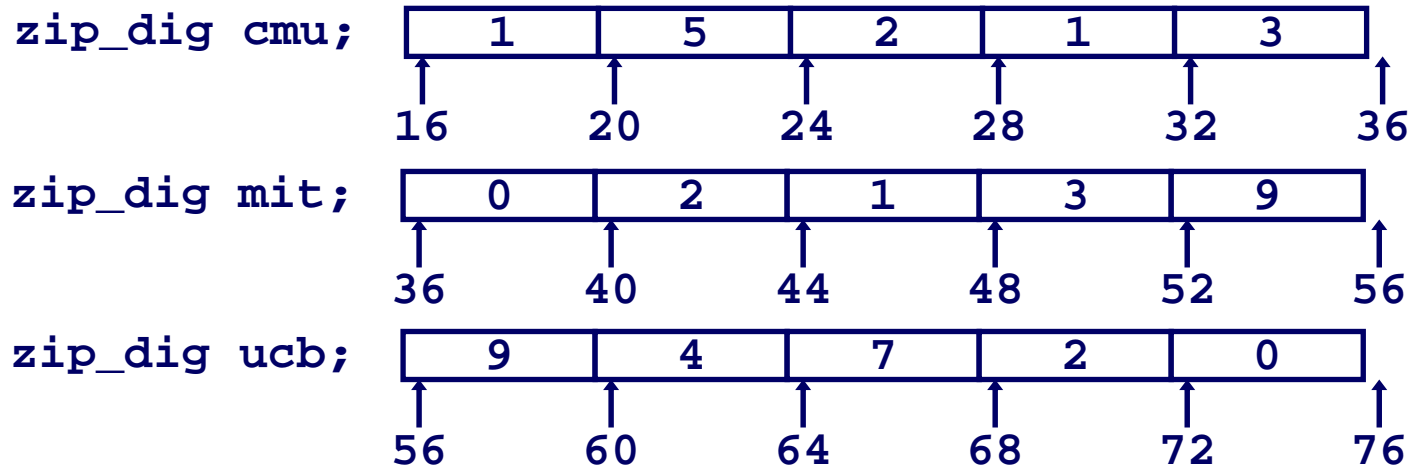
- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at $4 * \%eax + \%edx$
- Use memory reference
(`%edx, %eax, 4`)

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

IA32 Memory Reference Code

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```

Referencing Examples



Code Does Not Do Any Bounds Checking!

Reference	Address	Value	Guaranteed?
<code>mit[3]</code>	$36 + 4 * 3 = 48$	3	Yes
<code>mit[5]</code>	$36 + 4 * 5 = 56$	9	No
<code>mit[-1]</code>	$36 + 4 * -1 = 32$	3	No
<code>cmu[15]</code>	$16 + 4 * 15 = 76$??	No

- Out of range behavior implementation-dependent
 - No guaranteed relative allocation of different arrays

Array Loop Example

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;
}
```

Transformed Version

- As generated by GCC
- Eliminate loop variable *i*
- Convert array code to pointer code
- Express in do-while form
 - No need to test at entrance

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


Array Loop Implementation (IA32)

Registers

```
%ecx z
%eax zi
%ebx zend
```

Computations

- $10 * z_i + *z$ implemented as
 $*z + 2 * (z_i + 4 * z_i)$
 $z++$ increments by 4

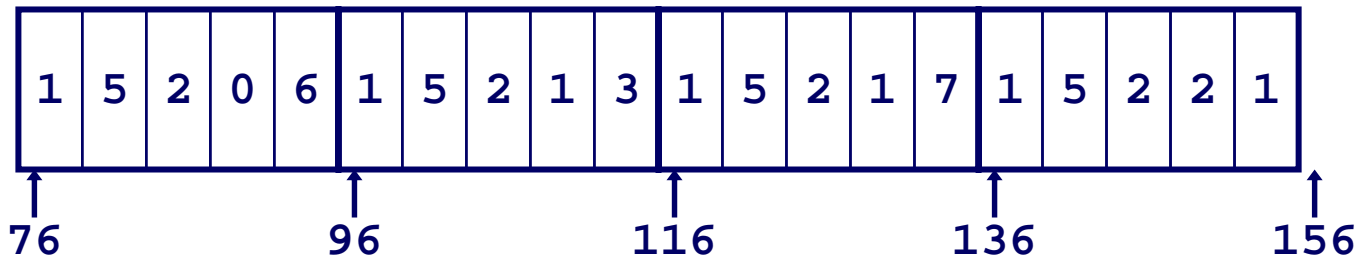
```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

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

Nested Array Example

```
#define PCOUNT 4
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];
```



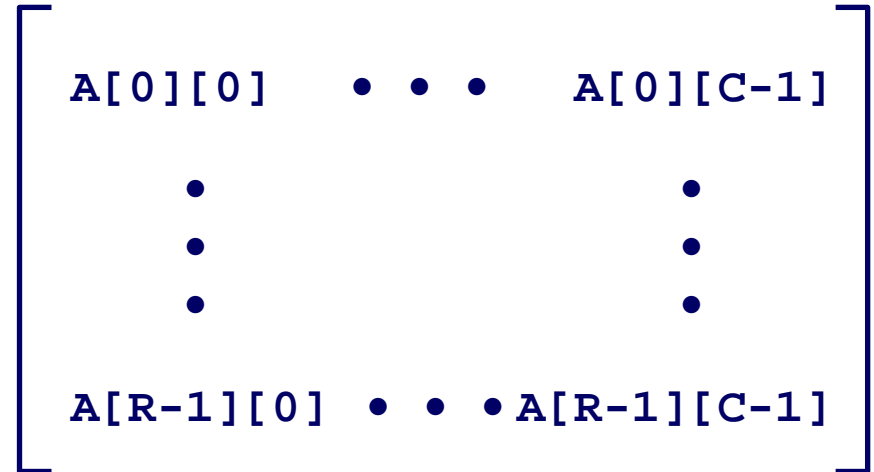
- Declaration “zip_dig pgh[4]” equivalent to “int pgh[4][5]”
 - Variable pgh denotes array of 4 elements
 - » Allocated contiguously
 - Each element is an array of 5 int’s
 - » Allocated contiguously
- “Row-Major” ordering of all elements guaranteed

Viewing as Multidimensional Array

Declaration

`T A[R][C];`

- 2D array of data type *T*
- *R* rows, *C* columns
- Type *T* element requires *K* bytes

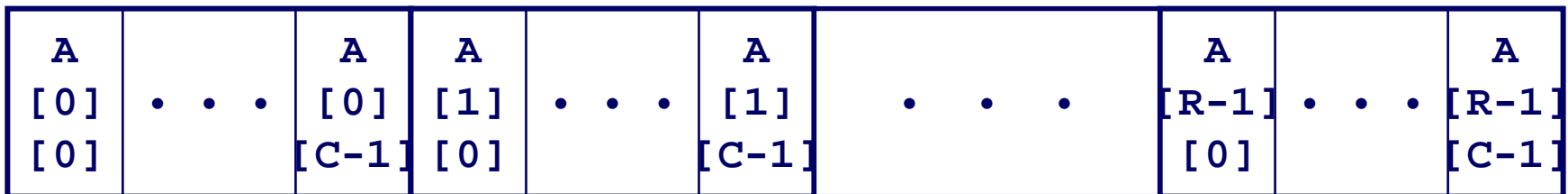


Array Size

- $R * C * K$ bytes

Arrangement

- Row-Major Ordering
- ```
int A[R][C];
```

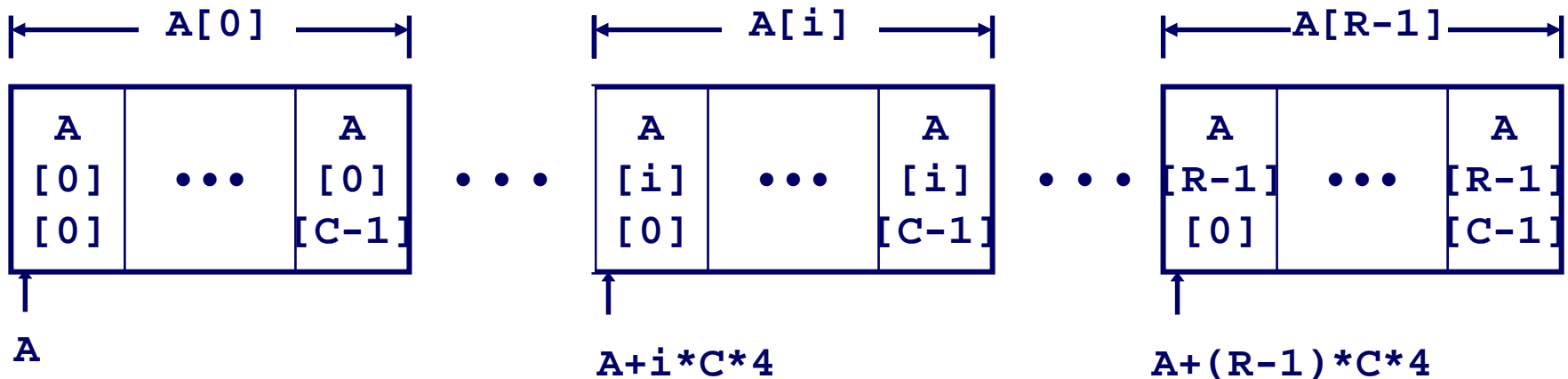


# Nested Array Row Access

## Row Vectors

- $A[i]$  is array of  $C$  elements
- Each element of type  $T$
- Starting address  $A + i * (C * K)$

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



# Nested Array Row Access Code

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

## Row Vector

- `pgh[index]` is array of 5 int's
- Starting address `pgh+20*index`

## IA32 Code

- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

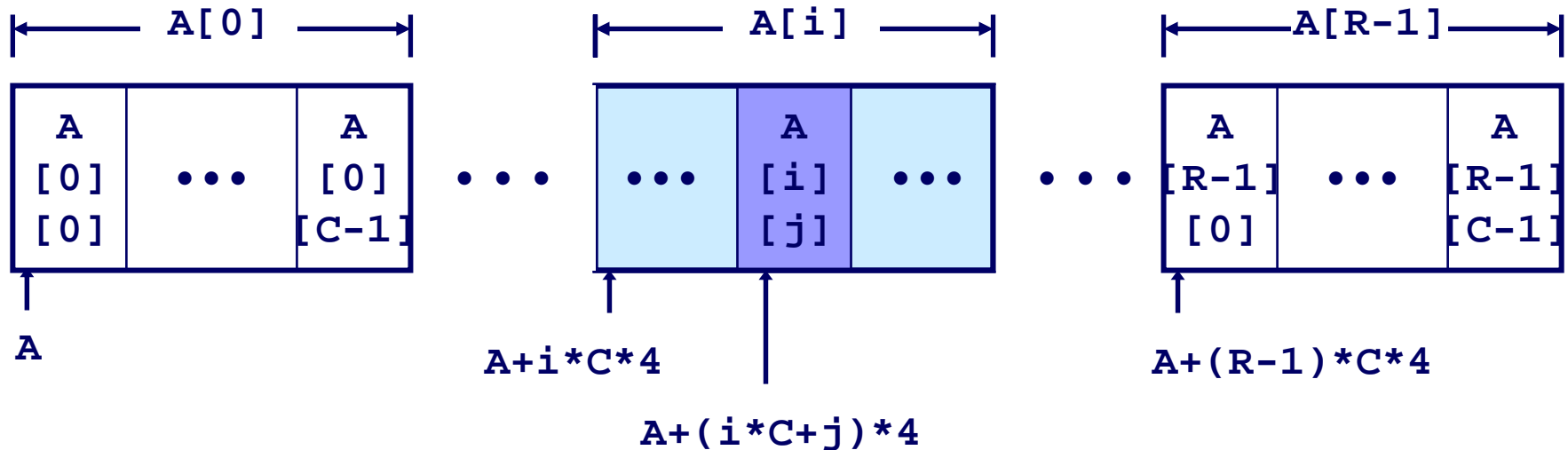
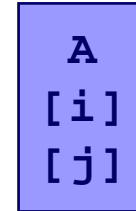
```
%eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal pgh(,%eax,4),%eax # pgh + (20 * index)
```

# Nested Array Element Access

## Array Elements

- $A[i][j]$  is element of type  $T$
- Address  $A + i * (C * K) + j * K$   
 $= A + (i * C + j) * K$

`int A[R][C];`



# Nested Array Element Access Code

## Array Elements

- `pgh[index][dig]` is `int`
- **Address:**  
`pgh + 20*index + 4*dig`

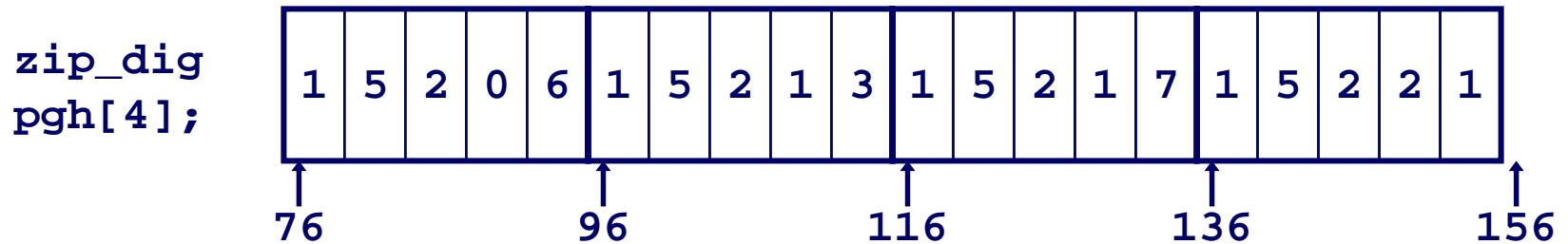
```
int get_pgh_digit
(int index, int dig)
{
 return pgh[index][dig];
}
```

## IA32 Code

- **Computes address**  
`pgh + 4*dig + 4*(index+4*index)`
- `movl` performs memory reference

```
%ecx = dig
%eax = index
leal 0(,%ecx,4),%edx # 4*dig
leal (%eax,%eax,4),%eax # 5*index
movl pgh(%edx,%eax,4),%eax # *(pgh + 4*dig + 20*index)
```

# Strange Referencing Examples



| Reference               | Address              | Value | Guaranteed? |
|-------------------------|----------------------|-------|-------------|
| <code>pgh[3][3]</code>  | $76+20*3+4*3 = 148$  | 2     | Yes         |
| <code>pgh[2][5]</code>  | $76+20*2+4*5 = 136$  | 1     | Yes         |
| <code>pgh[2][-1]</code> | $76+20*2+4*-1 = 112$ | 3     | Yes         |
| <code>pgh[4][-1]</code> | $76+20*4+4*-1 = 152$ | 1     | Yes         |
| <code>pgh[0][19]</code> | $76+20*0+4*19 = 152$ | 1     | Yes         |
| <code>pgh[0][-1]</code> | $76+20*0+4*-1 = 72$  | ??    | No          |

- Code does not do any bounds checking
- Ordering of elements within array guaranteed

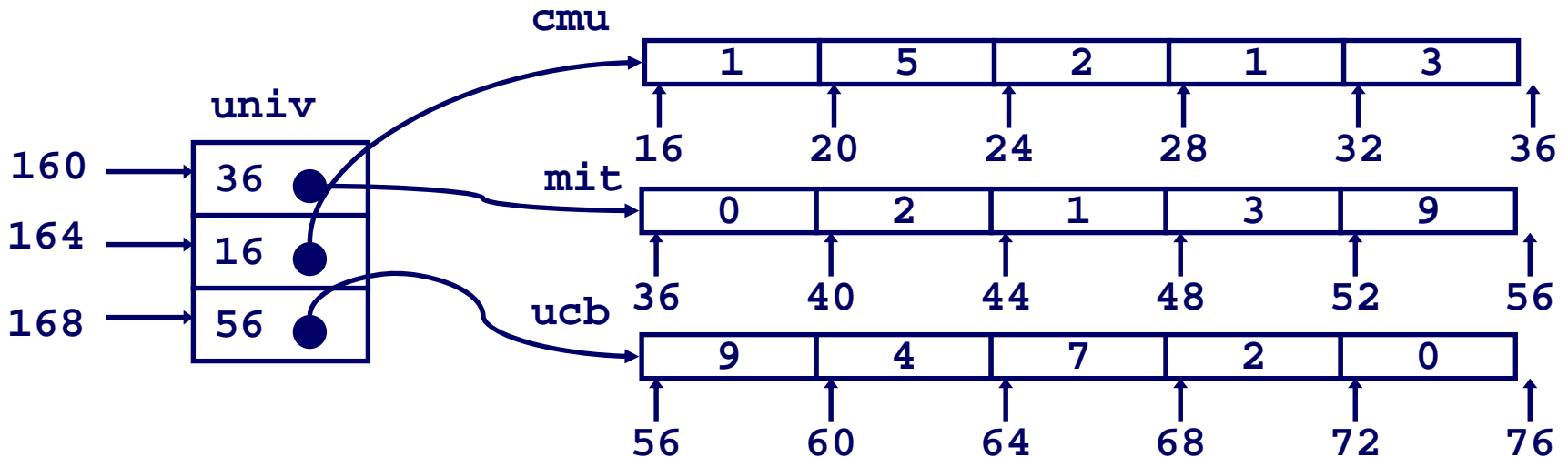


# Multi-Level Array Example

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

```
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};
```



# Element Access in Multi-Level Array

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

## Computation (IA32)

- Element access  
Mem[Mem[univ+4\*index]+4\*dig]
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

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

# Array Element Accesses

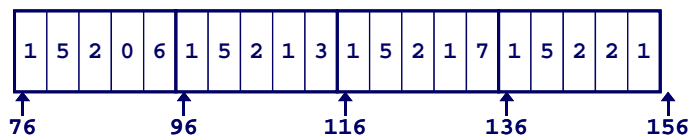
- Similar C references

## Nested Array

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

- Element at

$\text{Mem}[\text{pgh} + 20 * \text{index} + 4 * \text{dig}]$



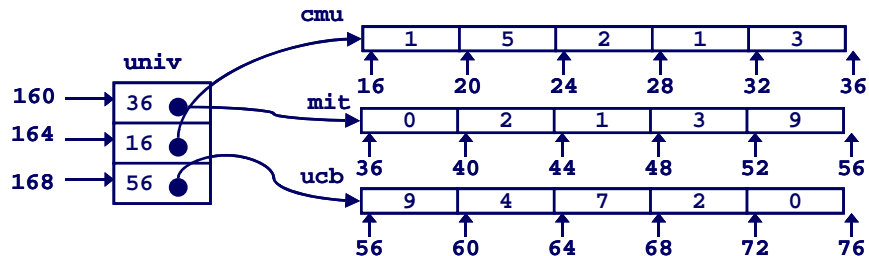
- Different address computation

## Multi-Level Array

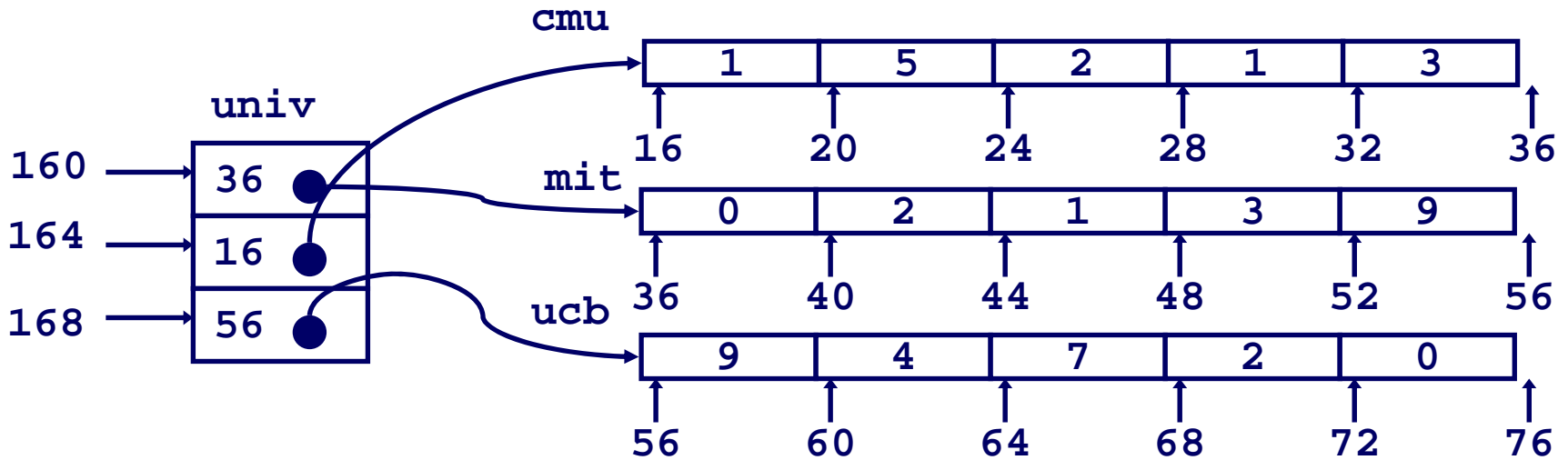
```
int get_univ_digit
(int index, int dig)
{
 return univ[index][dig];
}
```

- Element at

$\text{Mem}[\text{Mem}[\text{univ} + 4 * \text{index}] + 4 * \text{dig}]$



# Strange Referencing Examples



| Reference                | Address            | Value | Guaranteed? |
|--------------------------|--------------------|-------|-------------|
| <code>univ[2][3]</code>  | $56 + 4 * 3 = 68$  | 2     | Yes         |
| <code>univ[1][5]</code>  | $16 + 4 * 5 = 36$  | 0     | No          |
| <code>univ[2][-1]</code> | $56 + 4 * -1 = 52$ | 9     | No          |
| <code>univ[3][-1]</code> | ??                 | ??    | No          |
| <code>univ[1][12]</code> | $16 + 4 * 12 = 64$ | 7     | No          |

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

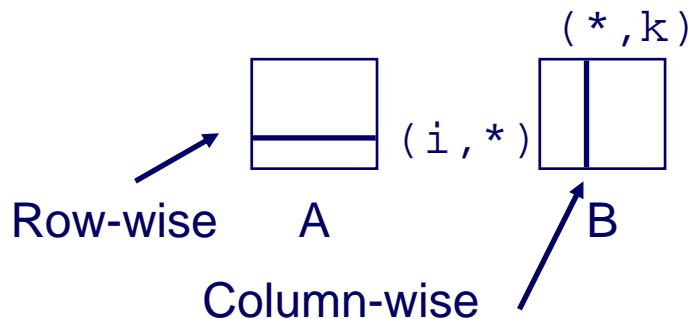
# Using Nested Arrays

## Strengths

- C compiler handles doubly subscripted arrays
- Generates very efficient code
  - Avoids multiply in index computation

## Limitation

- Only works if have fixed array size



```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Compute element i,k of
 fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
 int j;
 int result = 0;
 for (j = 0; j < N; j++)
 result += a[i][j]*b[j][k];
 return result;
}
```

# Dynamic Nested Arrays

## Strength

- Can create matrix of arbitrary size

## Programming

- Must do index computation explicitly

## Performance

- Accessing single element costly
- Must do multiplication

```
int * new_var_matrix(int n)
{
 return (int *)
 calloc(sizeof(int), n*n);
}
```

```
int var_ele
(int *a, int i,
 int j, int n)
{
 return a[i*n+j];
}
```

```
movl 12(%ebp),%eax # i
movl 8(%ebp),%edx # a
imull 20(%ebp),%eax # n*i
addl 16(%ebp),%eax # n*i+j
movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]
```

# Dynamic Array Multiplication

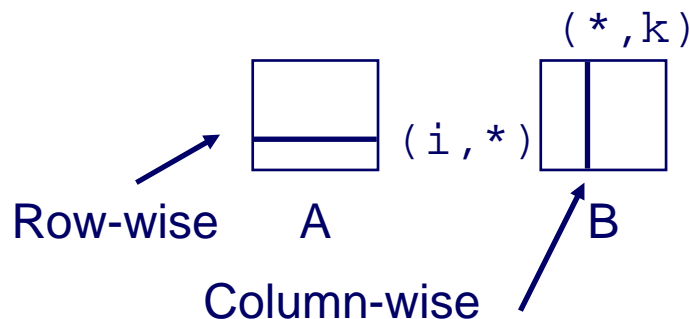
## Without Optimizations

### ■ Multiplies

- 2 for subscripts
- 1 for data

### ■ Adds

- 4 for array indexing
- 1 for loop index
- 1 for data



```
/* Compute element i,k of
 variable matrix product */
int var_prod_ele
(int *a, int *b,
 int i, int k, int n)
{
 int j;
 int result = 0;
 for (j = 0; j < n; j++)
 result +=
 a[i*n+j] * b[j*n+k];
 return result;
}
```

# Optimizing Dynamic Array Mult.

## Optimizations

- Performed when set optimization level to `-O2`

## Code Motion

- Expression `i*n` can be computed outside loop

## Strength Reduction

- Incrementing `j` has effect of incrementing `j*n+k` by `n`

## Performance

- Compiler can optimize regular access patterns

```
{
 int j;
 int result = 0;
 for (j = 0; j < n; j++)
 result +=
 a[i*n+j] * b[j*n+k];
 return result;
}
```

```
{
 int j;
 int result = 0;
 int iTn = i*n;
 int jTnPk = k;
 for (j = 0; j < n; j++) {
 result +=
 a[iTn+j] * b[jTnPk];
 jTnPk += n;
 }
 return result;
}
```



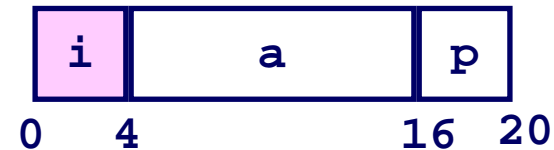
# Structures

## Concept

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

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

## Memory Layout



## Accessing Structure Member

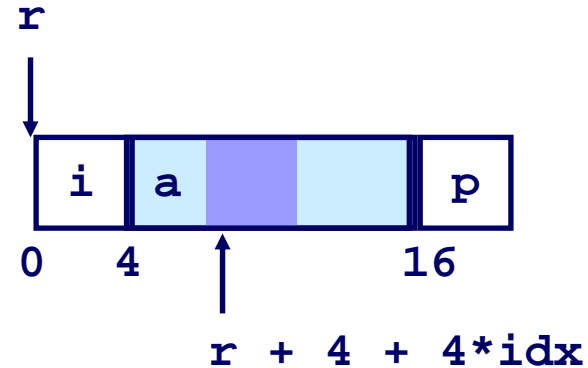
```
void
set_i(struct rec *r,
 int val)
{
 r->i = val;
}
```

## IA32 Assembly

```
%eax = val
%edx = r
movl %eax, (%edx) # Mem[r] = val
```

# Generating Pointer to Struct. Member

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



## Generating Pointer to Array Element

- Offset of each structure member determined at compile time

```
int *
find_a
(struct rec *r, int idx)
{
 return &r->a[idx];
}
```

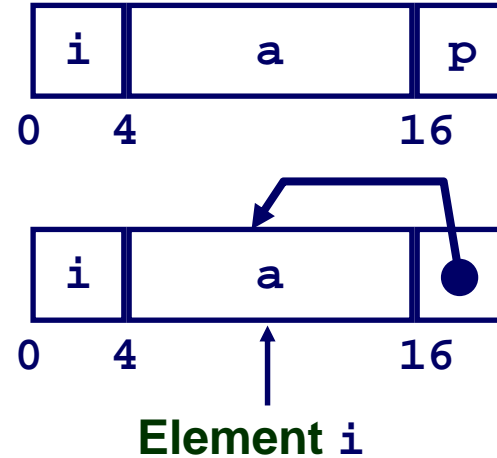
```
%ecx = idx
%edx = r
leal 0(,%ecx,4),%eax # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

# Structure Referencing (Cont.)

## C Code

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

```
void
set_p(struct rec *r)
{
 r->p =
 &r->a[r->i];
}
```



```
%edx = r
movl (%edx),%ecx # r->i
leal 0(,%ecx,4),%eax # 4*(r->i)
leal 4(%edx,%eax),%eax # r+4+4*(r->i)
movl %eax,16(%edx) # Update r->p
```

# Alignment

## Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
  - treated differently by IA32 Linux, x86-64 Linux, and Windows!

## Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory very tricky when datum spans 2 pages

## Compiler

- Inserts gaps in structure to ensure correct alignment of fields

# Specific Cases of Alignment (IA32)

## Size of Primitive Data Type:

- 1 byte (e.g., char)
  - no restrictions on address
- 2 bytes (e.g., short)
  - lowest 1 bit of address must be  $0_2$
- 4 bytes (e.g., int, float, char \*, etc.)
  - lowest 2 bits of address must be  $00_2$
- 8 bytes (e.g., double)
  - Windows (and most other OS's & instruction sets):
    - » lowest 3 bits of address must be  $000_2$
  - Linux:
    - » lowest 2 bits of address must be  $00_2$
    - » i.e., treated the same as a 4-byte primitive data type
- 12 bytes (long double)
  - Windows, Linux:
    - » lowest 2 bits of address must be  $00_2$
    - » i.e., treated the same as a 4-byte primitive data type

# Specific Cases of Alignment (x86-64)

## Size of Primitive Data Type:

- 1 byte (e.g., char)
  - no restrictions on address
- 2 bytes (e.g., short)
  - lowest 1 bit of address must be  $0_2$
- 4 bytes (e.g., int, float)
  - lowest 2 bits of address must be  $00_2$
- 8 bytes (e.g., double, char \*)
  - Windows & Linux:
    - » lowest 3 bits of address must be  $000_2$
- 16 bytes (long double)
  - Linux:
    - » lowest 3 bits of address must be  $000_2$
    - » i.e., treated the same as a 8-byte primitive data type

# Satisfying Alignment with Structures

## Offsets Within Structure

- Must satisfy element's alignment requirement

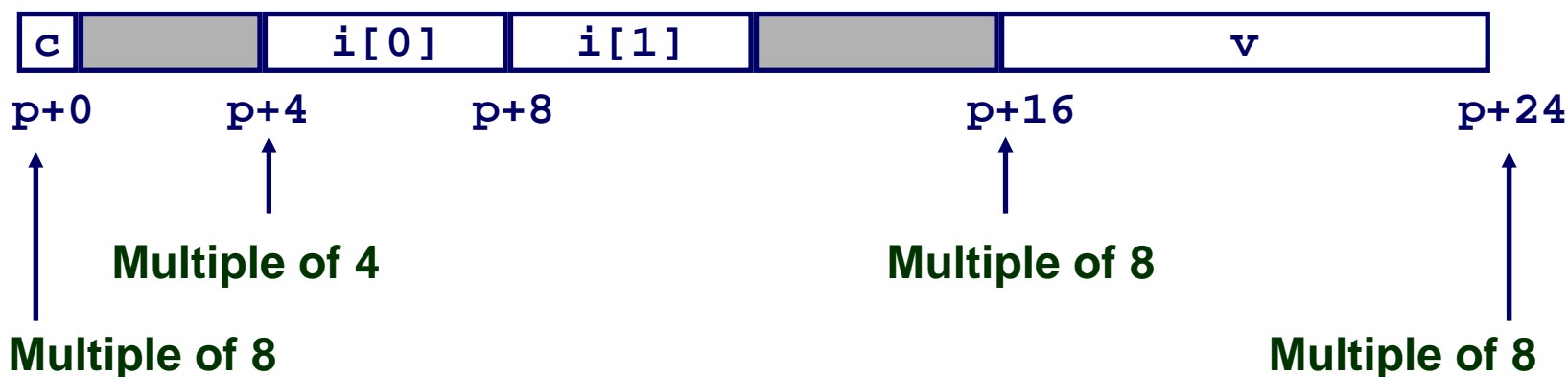
## Overall Structure Placement

- Each structure has alignment requirement K
  - Largest alignment of any element
- Initial address & structure length must be multiples of K

```
struct s1 {
 char c;
 int i[2];
 double v;
} *p;
```

## Example (under Windows or x86-64):

- $K = 8$ , due to `double` element

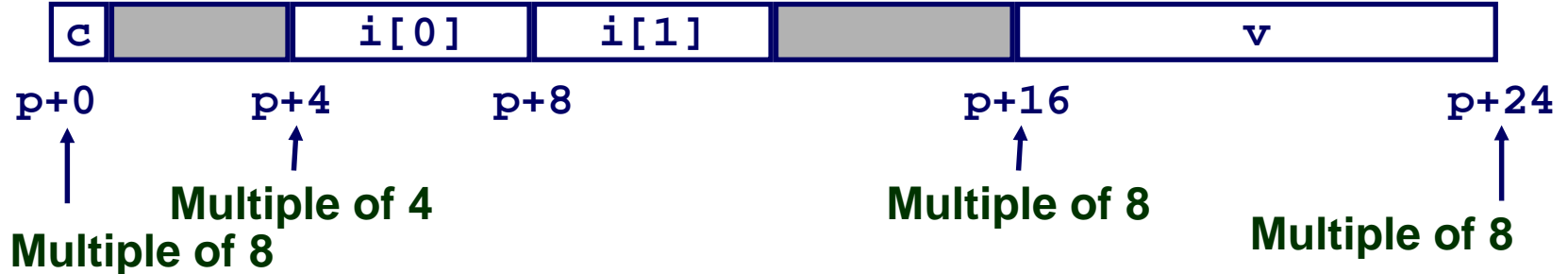


# Different Alignment Conventions

```
struct s1 {
 char c;
 int i[2];
 double v;
} *p;
```

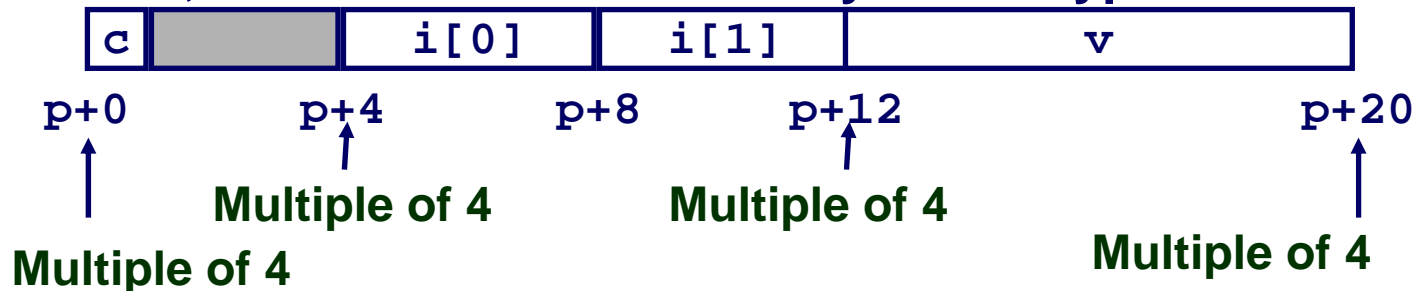
## x86-64 or IA32 Windows:

- `K = 8`, due to `double` element



## IA32 Linux

- `K = 4`; `double` treated like a 4-byte data type





# Overall Alignment Requirement

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

**p must be multiple of:**  
**8 for x86-64 or IA32 Windows**  
**4 for IA32 Linux**



p+0

p+8

p+12

p+16

**Windows: p+24**

**Linux: p+20**

```
struct S3 {
 float x[2];
 int i[2];
 char c;
} *p;
```

**p must be multiple of 4 (all cases)**



p+0

p+4

p+8

p+12

p+16

p+20

# Ordering Elements Within Structure

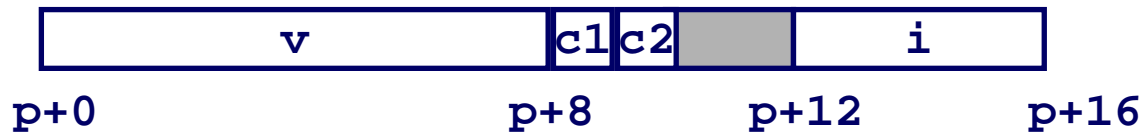
```
struct S4 {
 char c1;
 double v;
 char c2;
 int i;
} *p;
```

10 bytes wasted space in Windows  
or x86-64



```
struct S5 {
 double v;
 char c1;
 char c2;
 int i;
} *p;
```

2 bytes wasted space

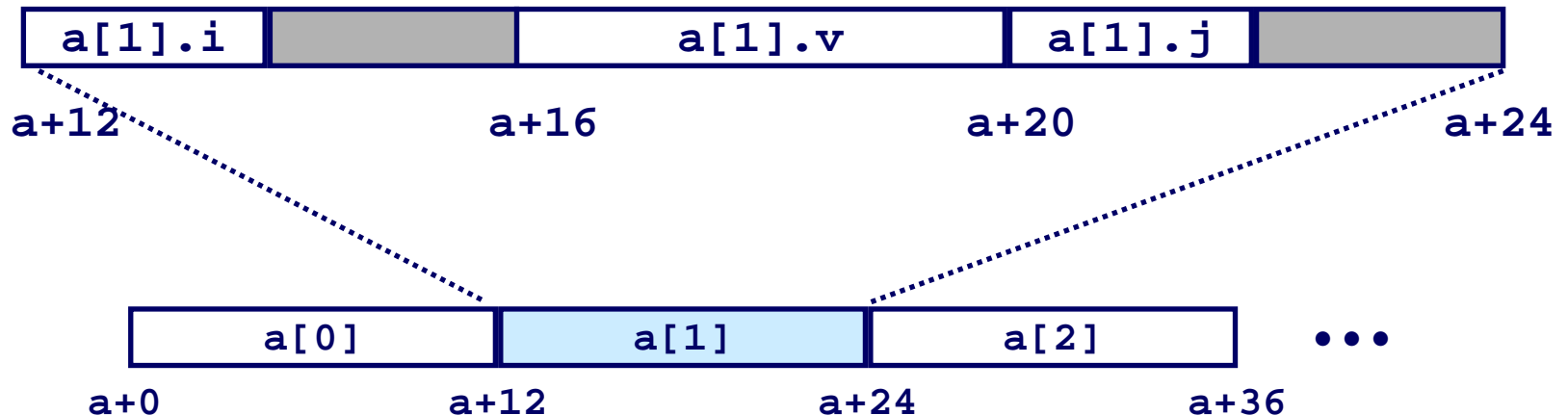


# Arrays of Structures

## Principle

- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```
struct s6 {
 short i;
 float v;
 short j;
} a[10];
```



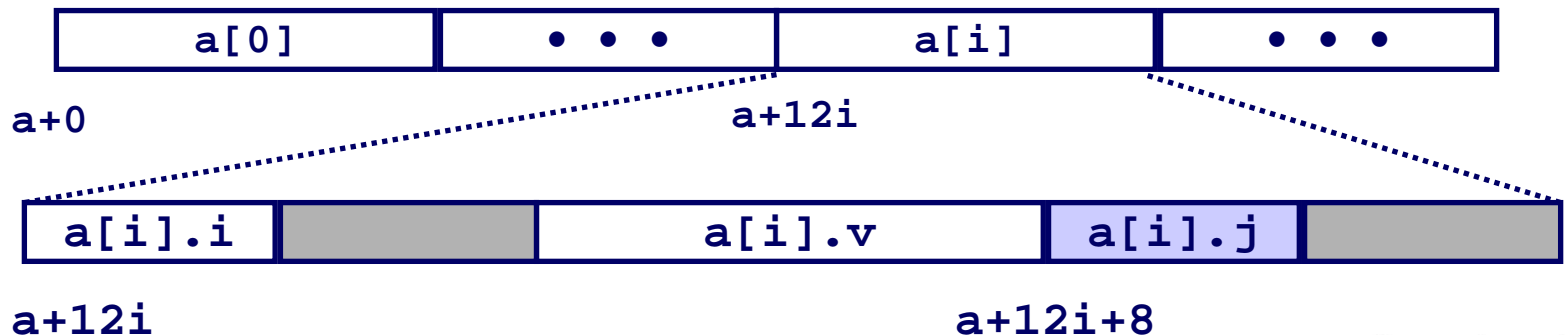
# Accessing Element within Array

- Compute offset to start of structure
  - Compute  $12*i$  as  $4*(i+2i)$
- Access element according to its offset within structure
  - Offset by 8
  - Assembler gives displacement as  $a + 8$ 
    - » Linker must set actual value

```
struct s6 {
 short i;
 float v;
 short j;
} a[10];
```

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

```
%eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

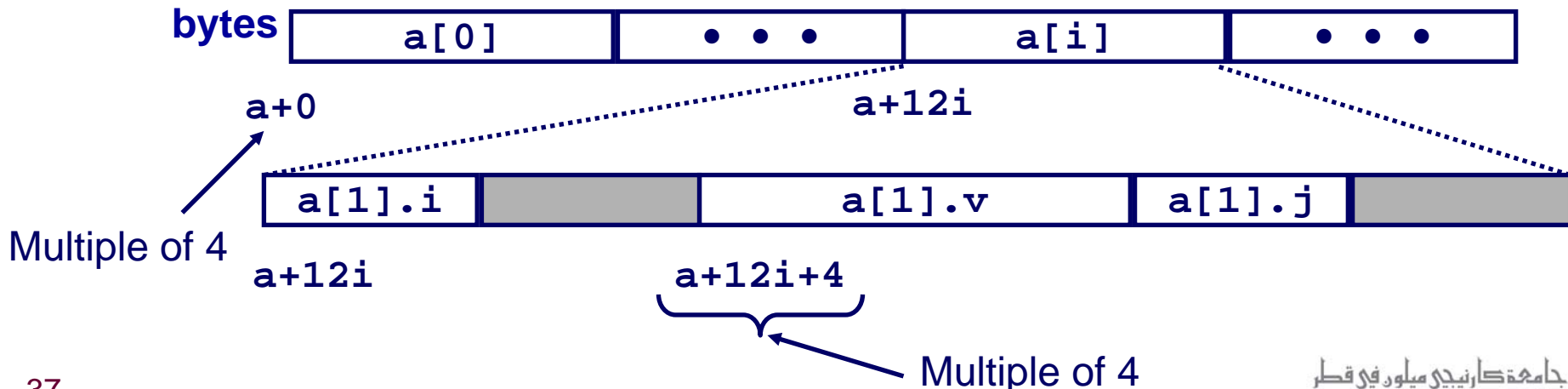


# Satisfying Alignment within Structure

## Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
  - $a$  must be multiple of 4
- Offset of element within structure must be multiple of element's alignment requirement
  - $v$ 's offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
  - Structure padded with unused space to be 12

```
struct s6 {
 short i;
 float v;
 short j;
} a[10];
```



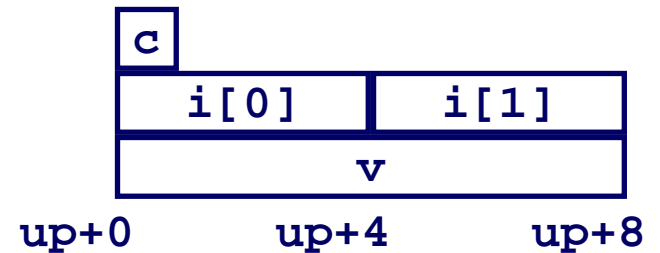
# Union Allocation

## Principles

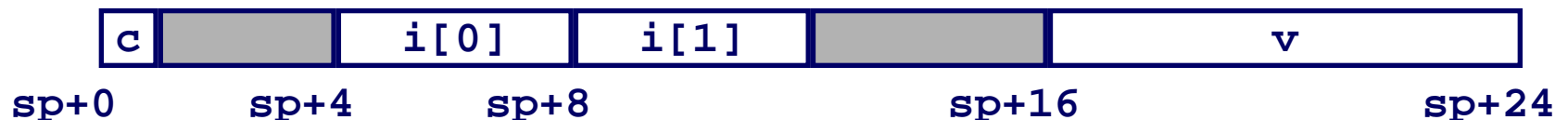
- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time

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

```
union U1 {
 char c;
 int i[2];
 double v;
} *up;
```

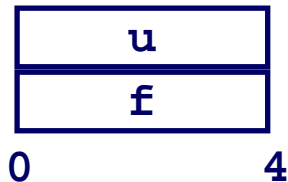


*(Windows alignment)*



# Using Union to Access Bit Patterns

```
typedef union {
 float f;
 unsigned u;
} bit_float_t;
```



```
float bit2float(unsigned u)
{
 bit_float_t arg;
 arg.u = u;
 return arg.f;
}
```

```
unsigned float2bit(float f)
{
 bit_float_t arg;
 arg.f = f;
 return arg.u;
}
```

- Get direct access to bit representation of float
- `bit2float` generates float with given bit pattern
  - NOT the same as `(float) u`
- `float2bit` generates bit pattern from float
  - NOT the same as `(unsigned) f`

# Byte Ordering Revisited

## Idea

- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

## Big Endian

- Most significant byte has lowest address
- PowerPC, Sparc

## Little Endian

- Least significant byte has lowest address
- Intel x86



# Byte Ordering Example

```
union {
 unsigned char c[8];
 unsigned short s[4];
 unsigned int i[2];
 unsigned long l[1];
} dw;
```

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| c[0] | c[1] | c[2] | c[3] | c[4] | c[5] | c[6] | c[7] |
| s[0] |      | s[1] |      | s[2] |      | s[3] |      |
| i[0] |      |      |      | i[1] |      |      |      |
| l[0] |      |      |      |      |      |      |      |

# Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
 dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
 dw.c[0], dw.c[1], dw.c[2], dw.c[3],
 dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

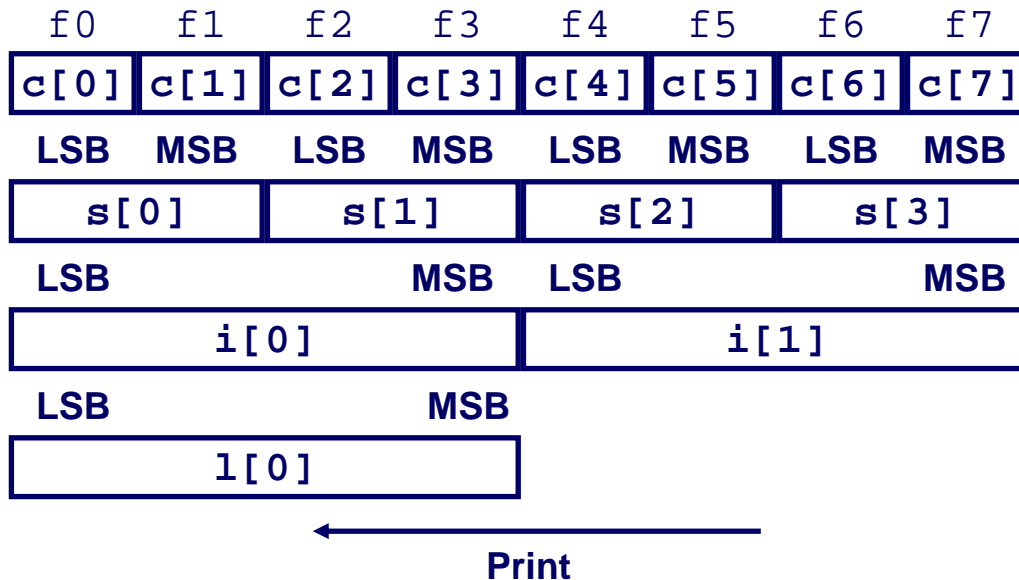
printf("Shorts 0-3 ==
[0x%x,0x%x,0x%x,0x%x]\n",
 dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
 dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
 dw.l[0]);
```

# Byte Ordering on IA32

## Little Endian

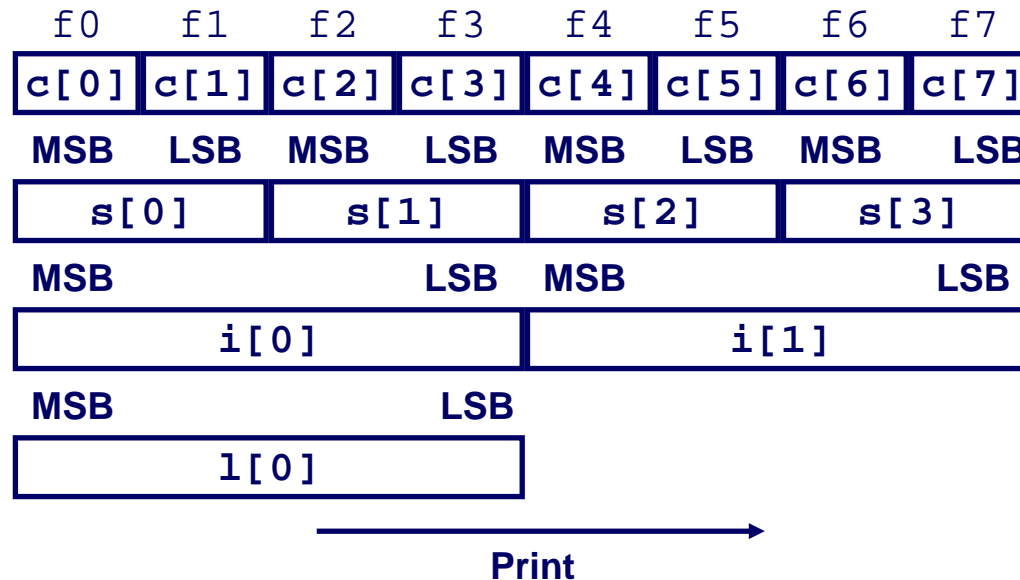


## Output on IA32:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]  
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]  
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]  
Long 0 == [0xf3f2f1f0]

# Byte Ordering on Sun

## Big Endian

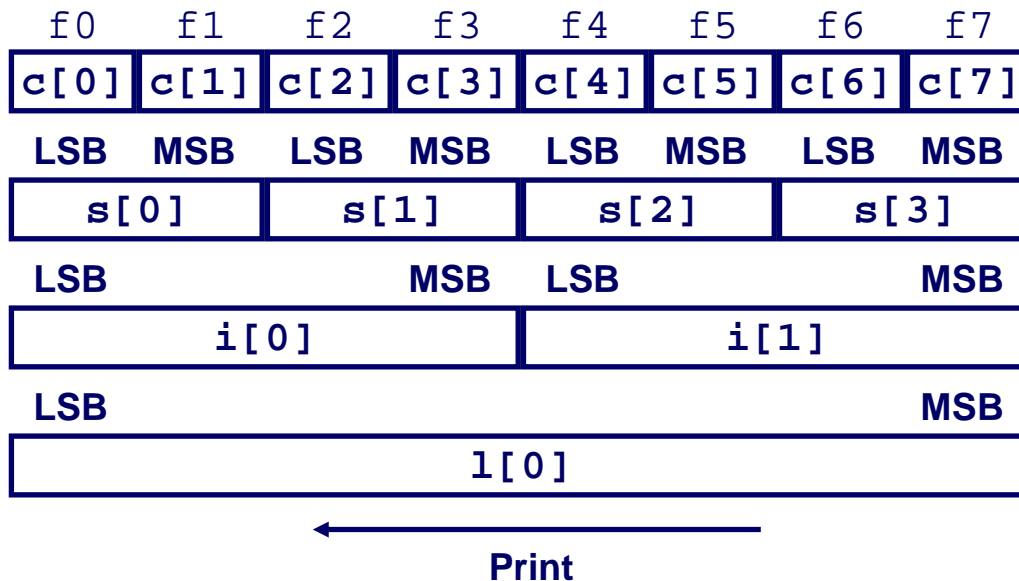


## Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints 0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long 0 == [0xf0f1f2f3]
```

# Byte Ordering on x86-64

## Little Endian



## Output on x86-64:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]  
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]  
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]  
Long 0 == [0xf7f6f5f4f3f2f1f0]

# Buffer Overflow Attacks

## November, 1988

- First Internet Worm spread over then-new Internet
- Many university machines compromised
- No malicious effect

## Today

- Buffer overflow is still the initial entry for over 50% of network-based attacks

# String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read

```
/* Get string from stdin */
char *gets(char *dest)
{
 int c = getc();
 char *p = dest;
 while (c != EOF && c != '\n') {
 *p++ = c;
 c = getc();
 }
 *p = '\0';
 return dest;
}
```

- Similar problems with other Unix functions
  - `strcpy`: Copies string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

# Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
 char buf[4]; /* Way too small! */
 gets(buf);
 puts(buf);
}
```

```
int main()
{
 printf("Type a string:");
 echo();
 return 0;
}
```



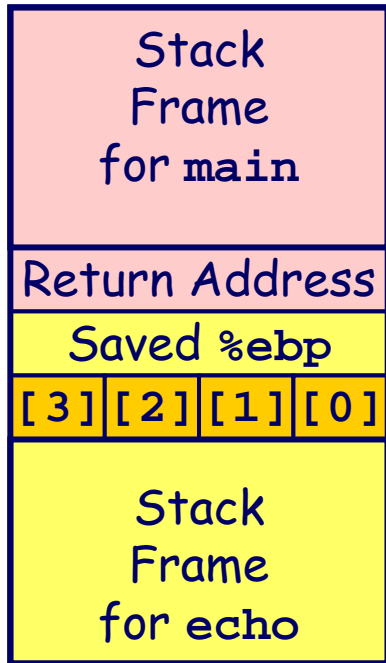
# Buffer Overflow Executions

```
unix> ./bufdemo
Type a string:123
123
```

```
unix> ./bufdemo
Type a string:12345
Segmentation Fault
```

```
unix> ./bufdemo
Type a string:12345678
Segmentation Fault
```

# Buffer Overflow Stack (IA32)



```
/* Echo Line */
void echo()
{
 char buf[4]; /* Way too small! */
 gets(buf);
 puts(buf);
}
```

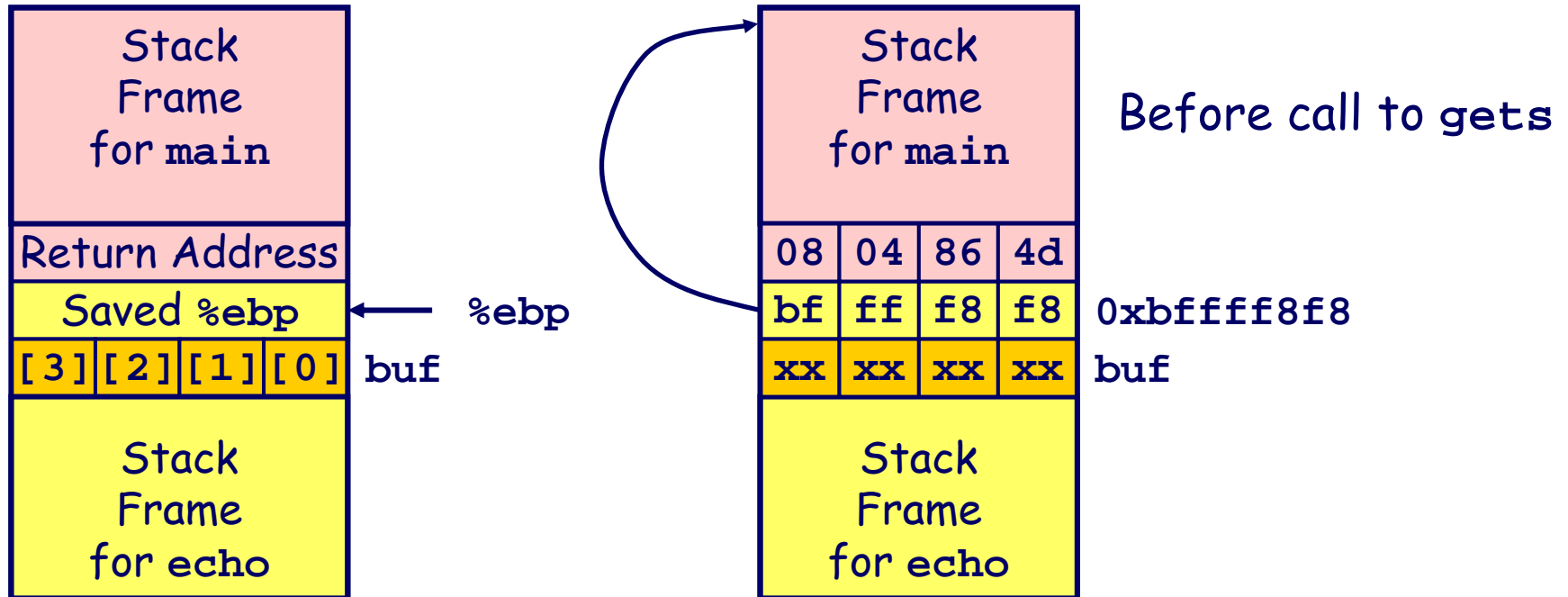
```
echo:
 pushl %ebp # Save %ebp on stack
 movl %esp,%ebp
 subl $20,%esp # Allocate stack space
 pushl %ebx # Save %ebx
 addl $-12,%esp # Allocate stack space
 leal -4(%ebp),%ebx # Compute buf as %ebp-4
 pushl %ebx # Push buf on stack
 call gets # Call gets
 . . .
```

# Buffer Overflow Stack Example

```

unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d

```



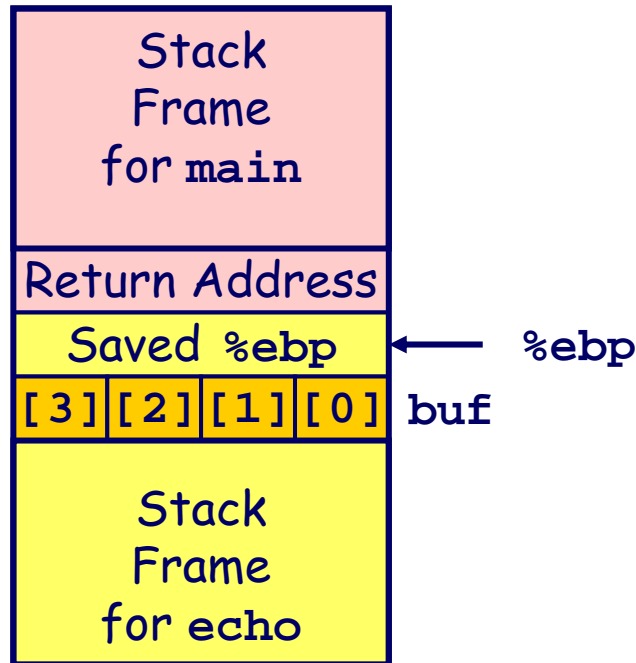
```

8048648: call 804857c <echo>
804864d: mov 0xffffffe8(%ebp),%ebx # Return Point

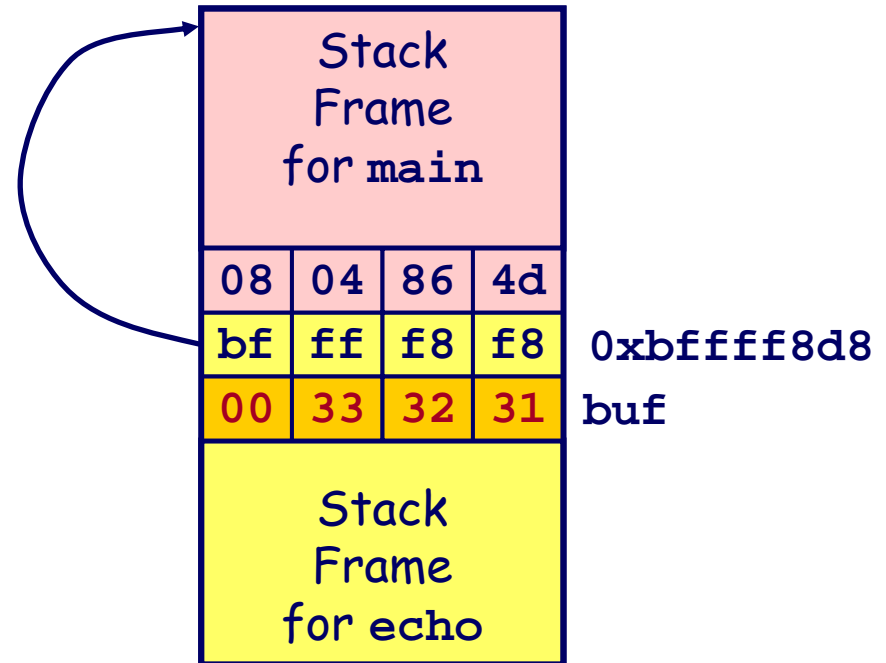
```

# Buffer Overflow Example #1

Before Call to gets

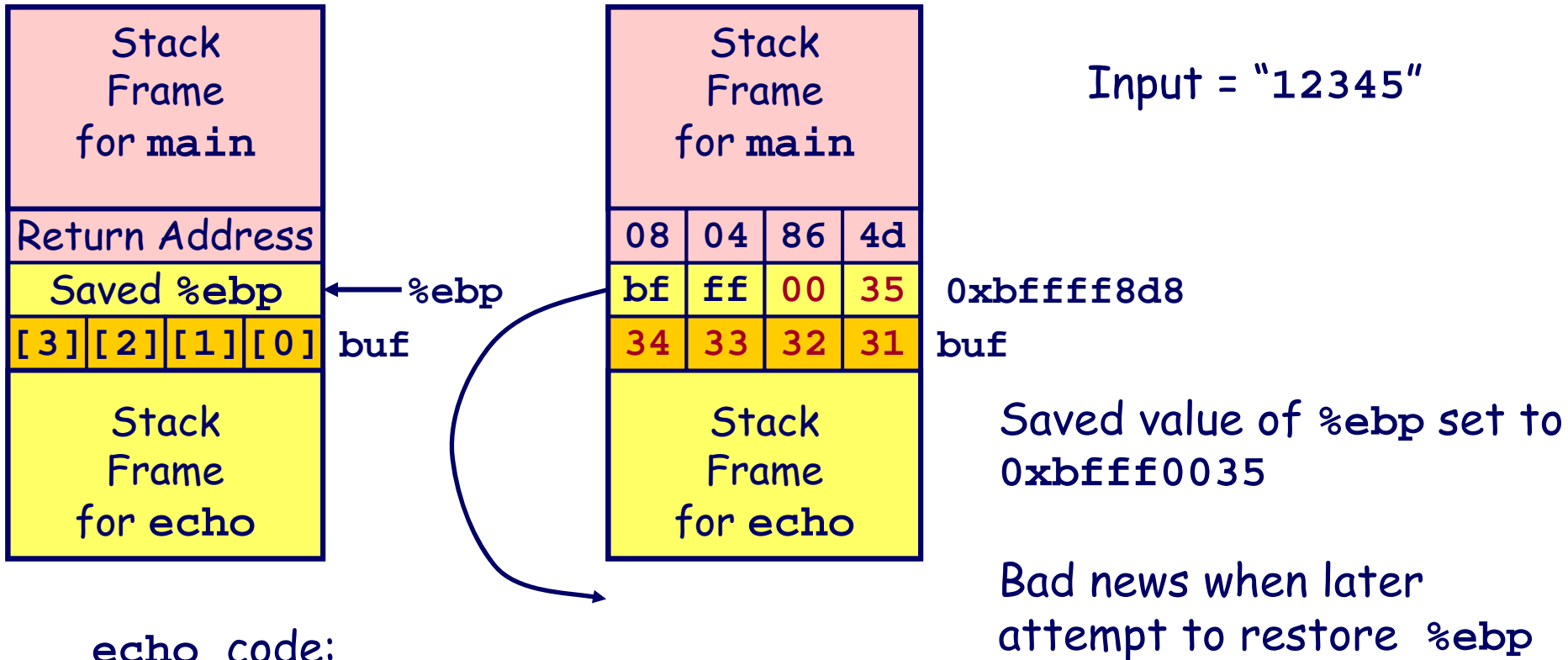


Input = "123"



No Problem

# Buffer Overflow Stack Example #2



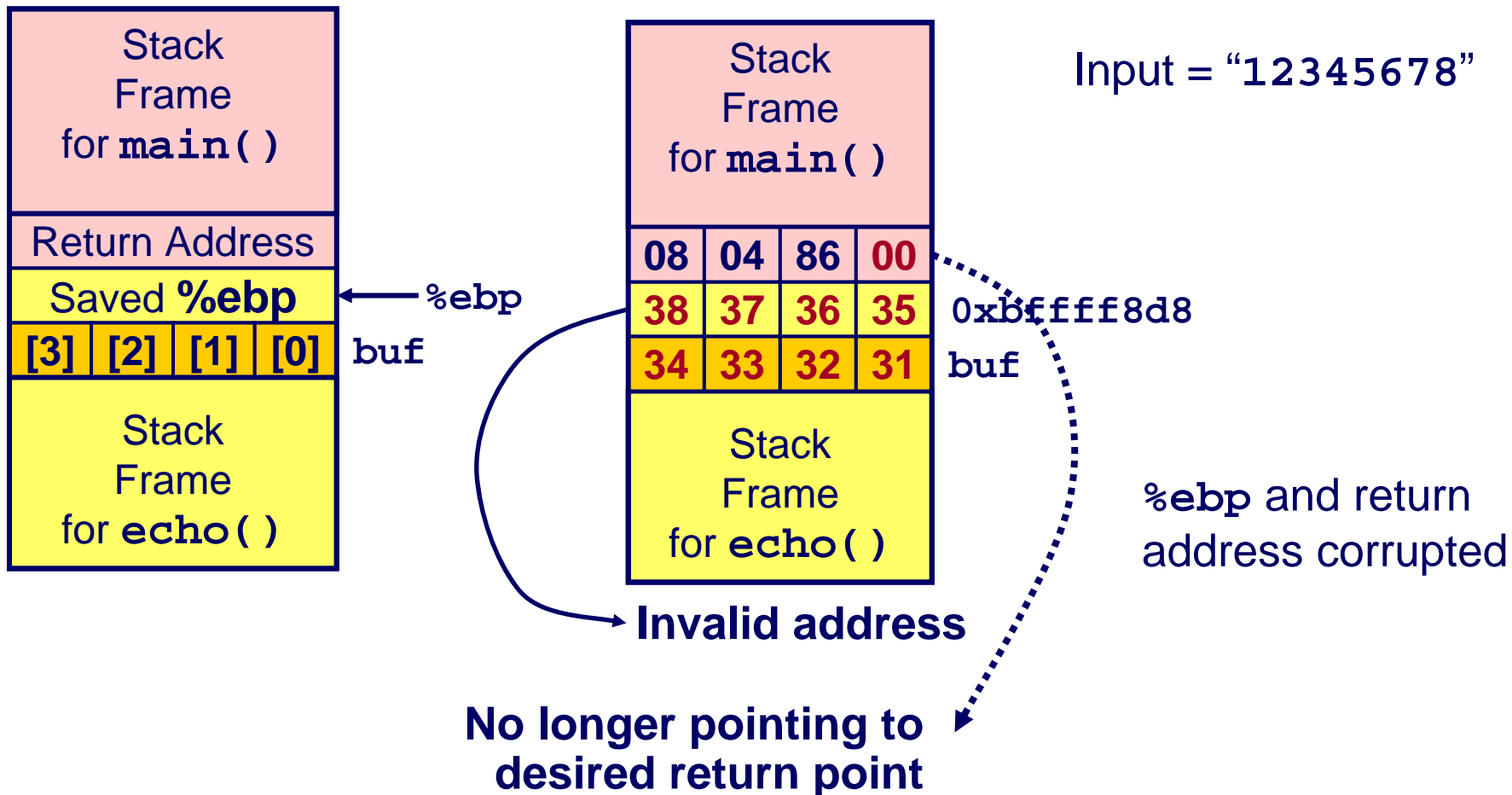
echo code:

```

8048592: push %ebx
8048593: call 80483e4 <_init+0x50> # gets
8048598: mov 0xffffffffe8(%ebp),%ebx
804859b: mov %ebp,%esp
804859d: pop %ebp # %ebp gets set to invalid value
804859e: ret

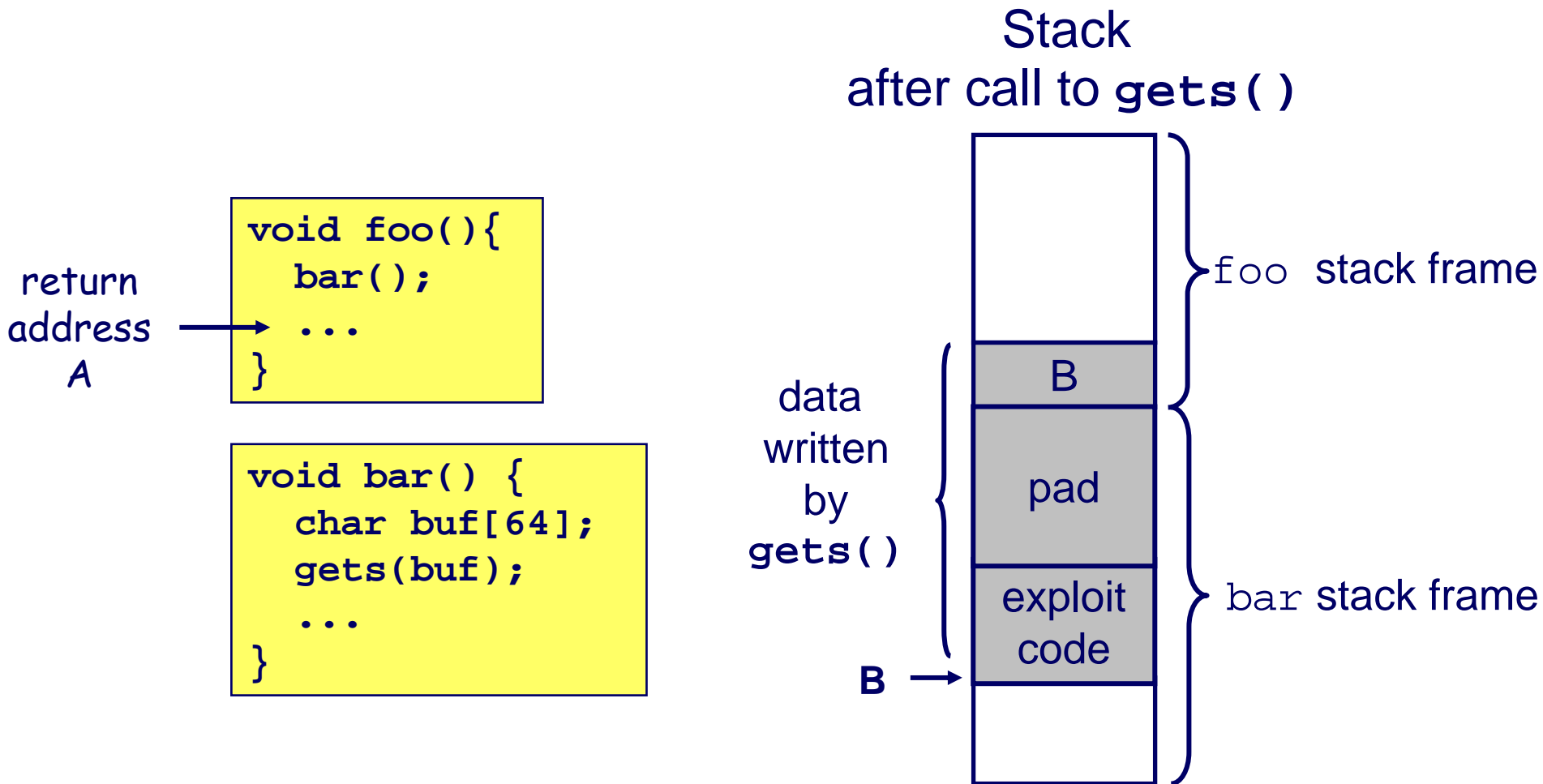
```

# Buffer Overflow Stack Example #3



```
8048648: call 804857c <echo>
804864d: mov 0xffffffe8(%ebp),%ebx # Return Point
```

# Malicious Use of Buffer Overflow



- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When `bar()` executes `ret`, will jump to exploit code

# Exploits Based on Buffer Overflows

*Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.*

## Internet worm

- Early versions of the finger server (fingerd) used `gets ( )` to read the argument sent by the client:
  - `finger droh@cs.cmu.edu`
- Worm attacked fingerd server by sending phony argument:
  - `finger "exploit-code padding new-return-address"`
  - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.



# Summary

## Arrays in C

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

## Structures

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

## Unions

- Overlay declarations
- Way to circumvent type system

## Buffer Overflow

- Overrun stack state with externally supplied data
- Potentially contains executable code