

Machine-Level Programming: Basic Data Types

Fall 2012

Instructors:

Aykut & Erkut Erdem

Acknowledgement: The course slides are adapted from the slides prepared by R.E. Bryant, D.R. O'Hallaron, G. Kesden and Markus Püschel of Carnegie-Mellon Univ.

Last time

- **Control: Condition codes**
- **Conditional branches**
- **Loops**
- **Switch statements**
- **IA 32 Procedures**
 - Stack Structure
 - Calling Conventions
 - Illustrations of Recursion & Pointers

Today

- **Arrays**
 - One-dimensional
 - Multi-dimensional (nested)
 - Multi-level
- Structures
- Unions
- Memory Layout
- Buffer Overflow
- Floating Point

Basic Data Types

■ Integral

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

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

■ Floating Point

- Stored & operated on in floating point registers

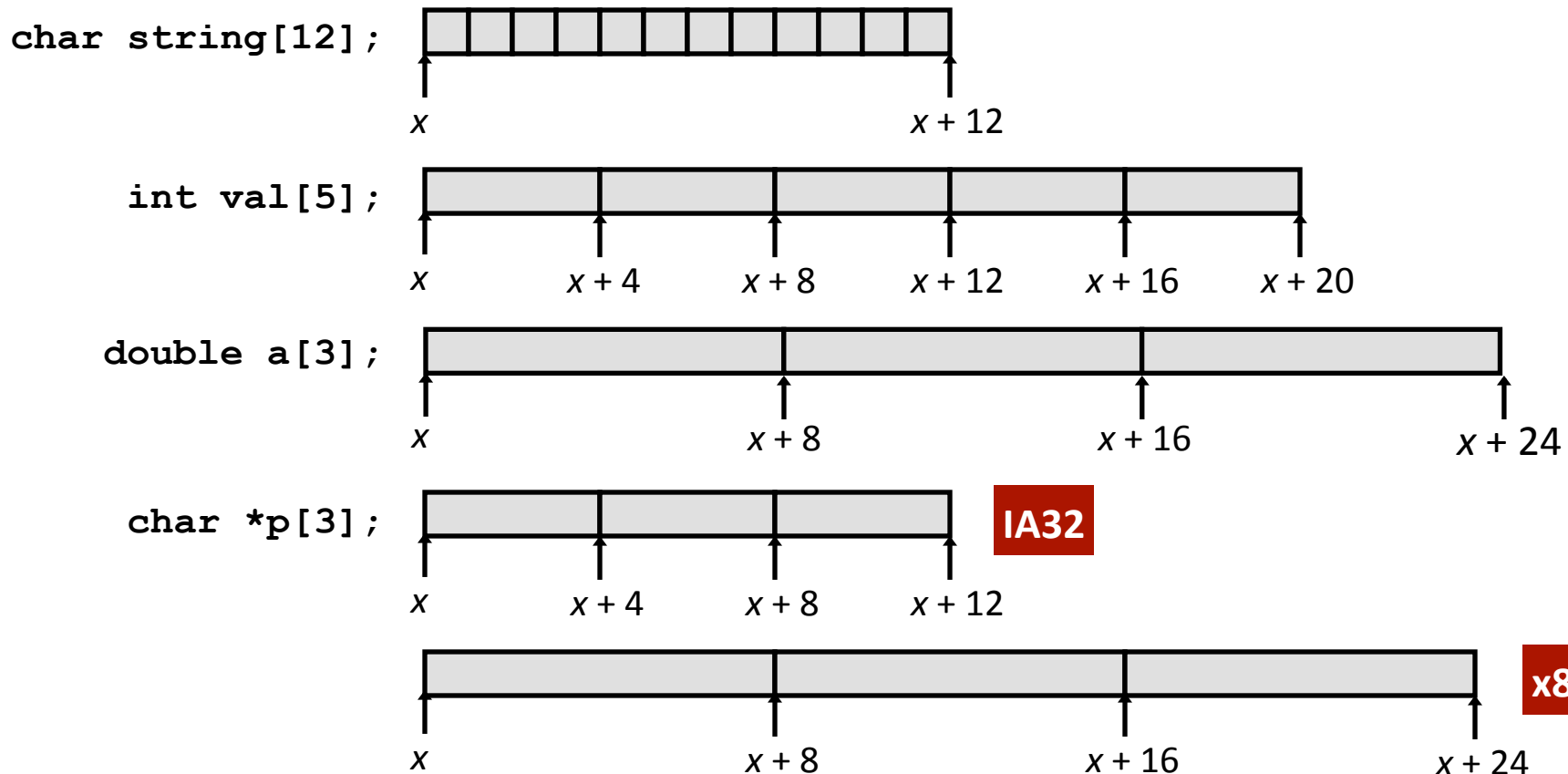
Intel	ASM	Bytes	C
Single	s	4	float
Double	d	8	double
Extended	t	10/12/16	long double

Array Allocation

■ Basic Principle

$T \ A[L];$

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

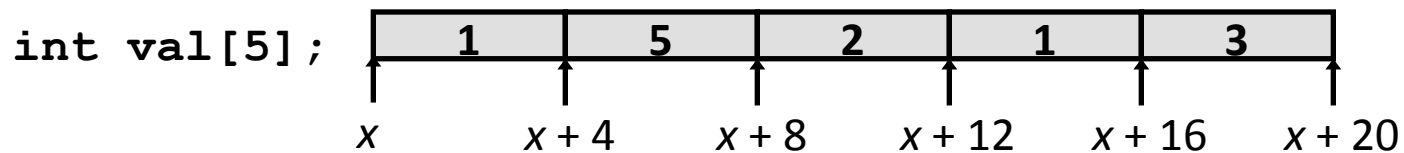


Array Access

■ Basic Principle

T $\mathbf{A}[L]$;

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

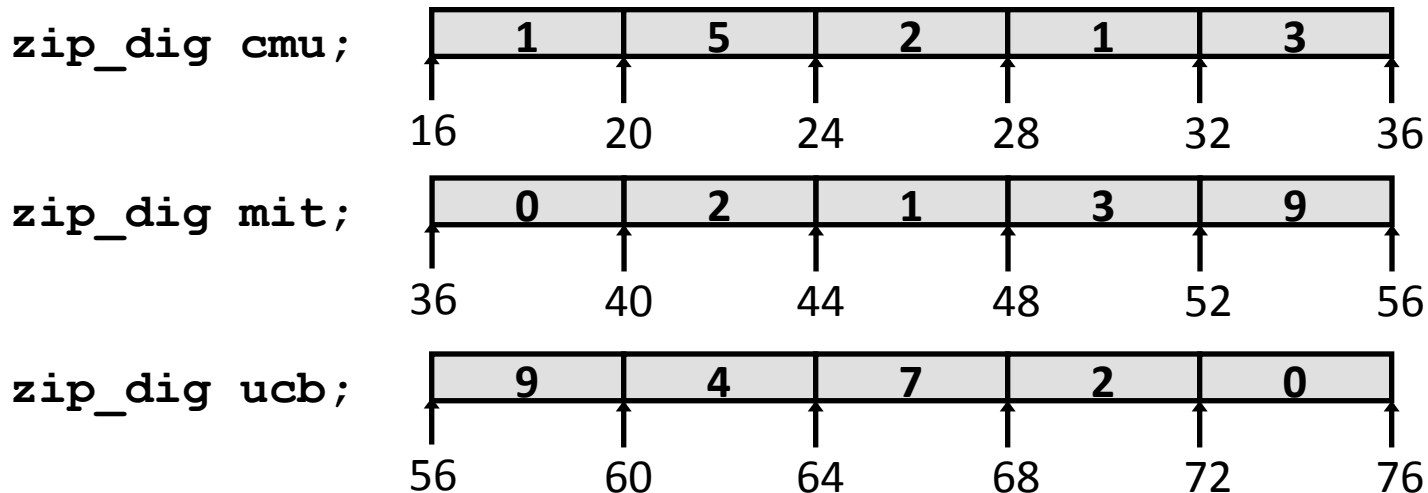


Reference	Type	Value
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	x
<code>val+1</code>	<code>int *</code>	$x+4$
<code>&val[2]</code>	<code>int *</code>	$x+8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *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

Array Example

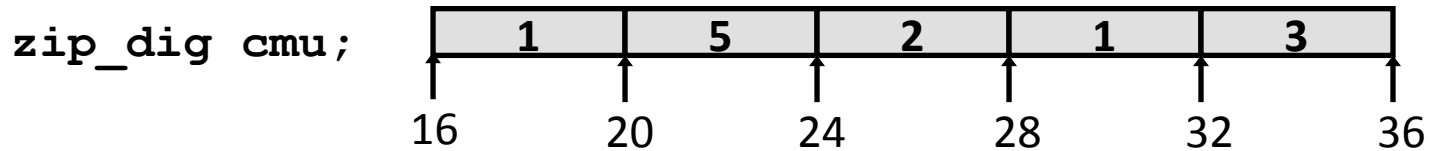
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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



- 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



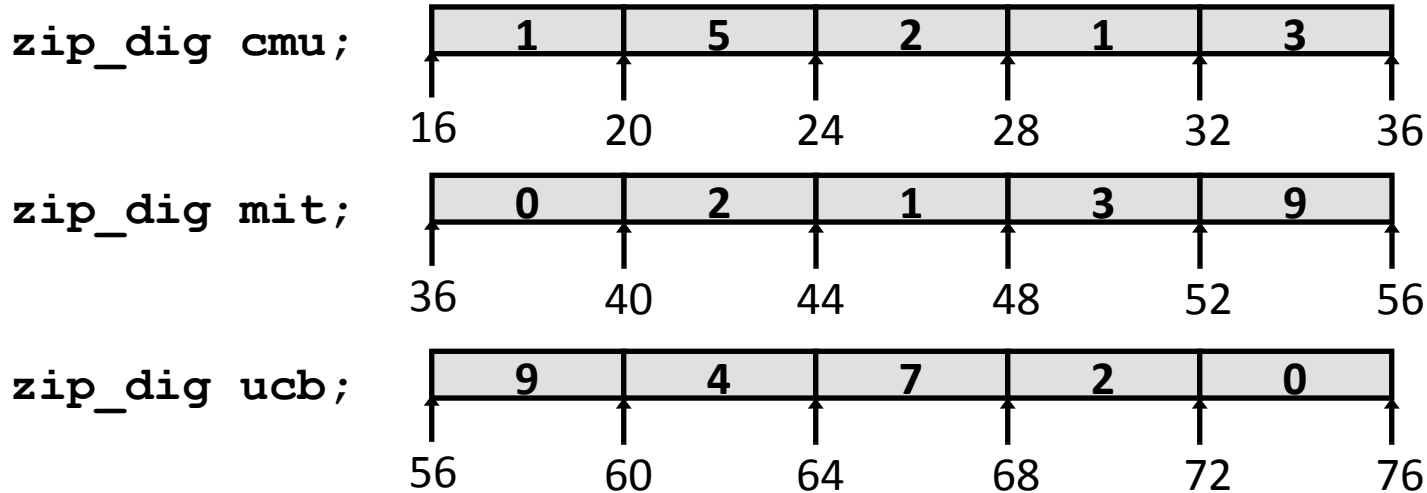
```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

IA32

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

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

Referencing Examples



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

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

Array Loop Example (IA32)

```
void zincr(zip_dig z) {  
    int i;  
    for (i = 0; i < ZLEN; i++)  
        z[i]++;  
}
```

```
# edx = z  
movl    $0, %eax           # %eax = i  
.L4:      # loop:  
addl    $1, (%edx,%eax,4)  # z[i]++  
addl    $1, %eax          # i++  
cmpl    $5, %eax         # i:5  
jne     .L4              # if !=, goto loop
```

Pointer Loop Example (IA32)

```
void zincr_p(zip_dig z) {
    int *zend = z+ZLEN;
    do {
        (*z)++;
        z++;
    } while (z != zend);
}
```

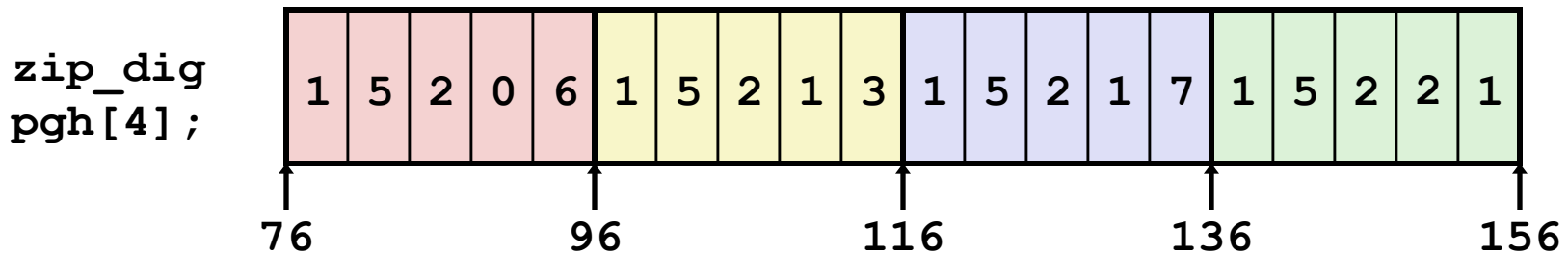


```
void zincr_v(zip_dig z) {
    void *vz = z;
    int i = 0;
    do {
        (*((int *) (vz+i)))++;
        i += ISIZE;
    } while (i != ISIZE*ZLEN);
}
```

```
# edx = z = vz
movl  $0, %eax
.L8:
addl  $1, (%edx,%eax)
addl  $4, %eax
cmpl  $20, %eax
jne   .L8
# i = 0
# loop:
# Increment vz+i
# i += 4
# Compare i:20
# 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]” equivalent to “int pgh[4][5]”**
 - Variable `pgh`: array of 4 elements, allocated contiguously
 - Each element is an array of 5 `int`'s, allocated contiguously
- **“Row-Major” ordering of all elements guaranteed**

Multidimensional (Nested) Arrays

■ Declaration

T $A[R][C];$

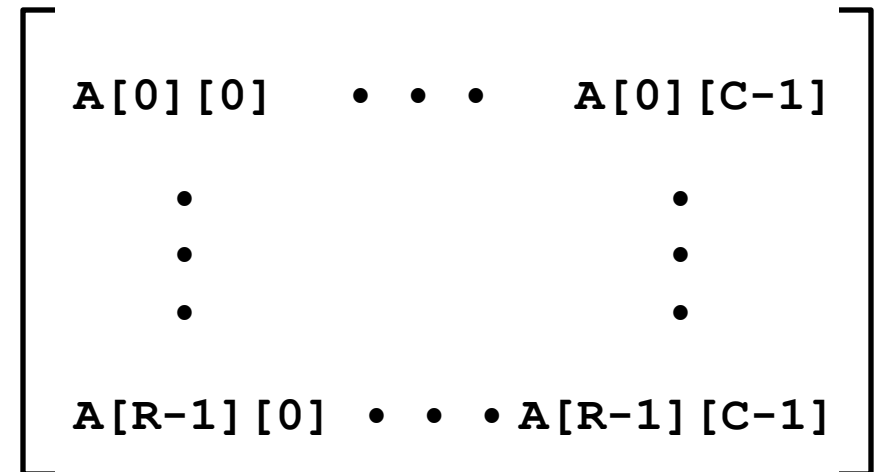
- 2D array of data type T
- R rows, C columns
- Type T element requires K bytes

■ Array Size

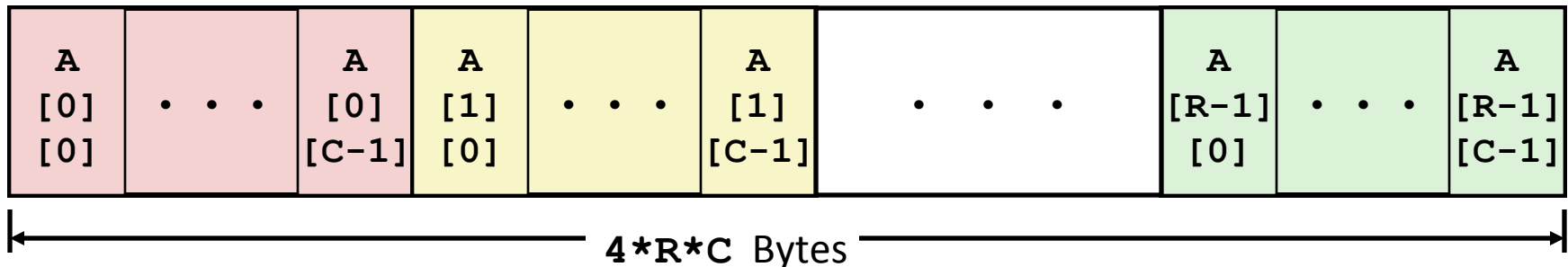
- $R * C * K$ bytes

■ Arrangement

- Row-Major Ordering



`int A[R][C];`

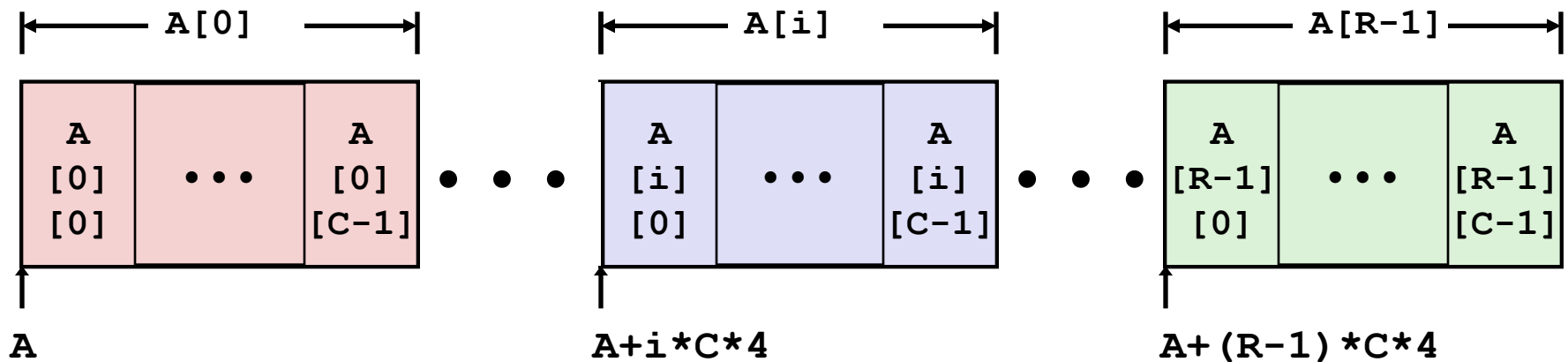


Nested Array Row Access

■ Row Vectors

- $\mathbf{A}[i]$ is array of C elements
- Each element of type T requires K bytes
- Starting address $\mathbf{A} + i * (C * K)$

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



Nested Array Row Access Code

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

```
#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 }};
```

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

■ Row Vector

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

■ IA32 Code

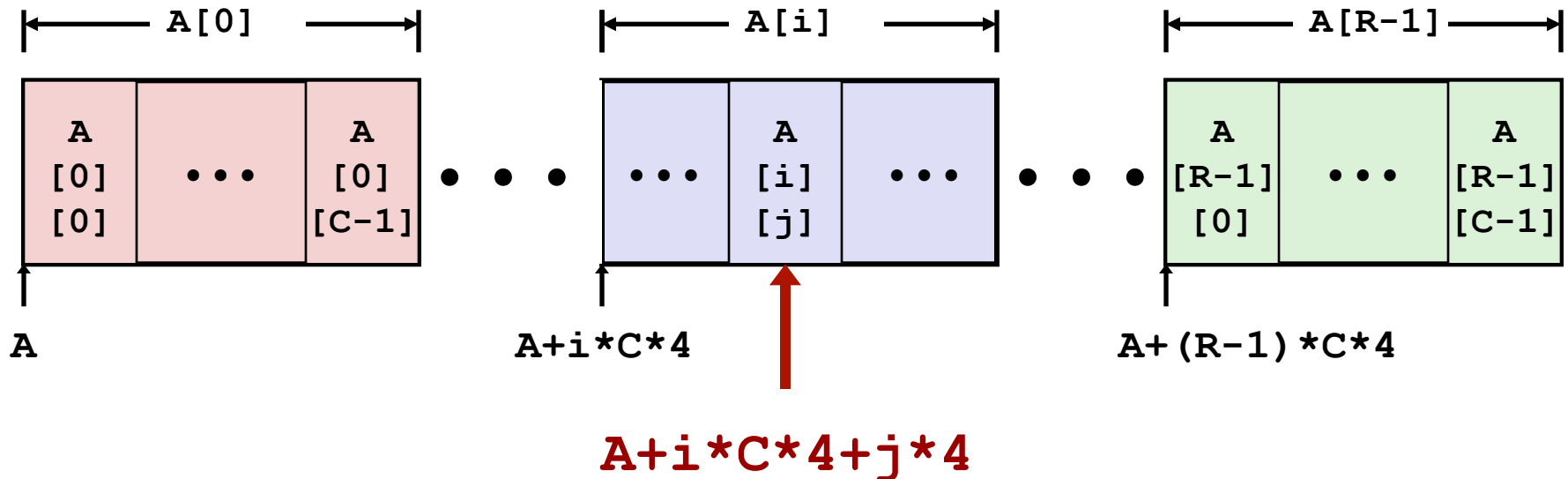
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

Nested Array Row Access

■ Array Elements

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

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



Nested Array Element Access Code

```

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

```

```

movl   8(%ebp), %eax      # index
leal   (%eax,%eax,4), %eax # 5*index
addl   12(%ebp), %eax     # 5*index+dig
movl   pgh(,%eax,4), %eax # offset 4*(5*index+dig)

```

■ Array Elements

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

■ IA32 Code

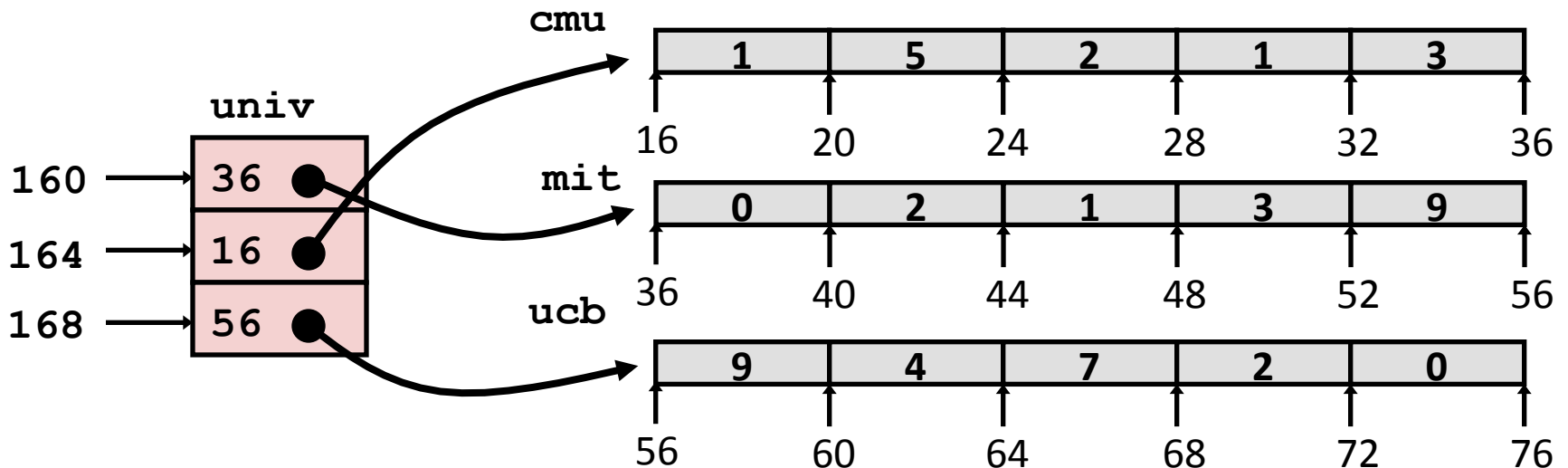
- Computes address `pgh + 4*((index+4*index)+dig)`

Multi-Level Array Example

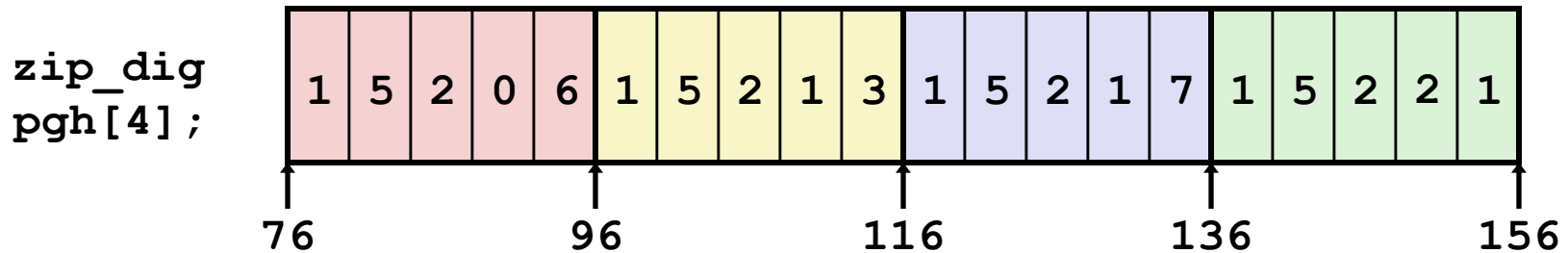
```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

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



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

Element Access in Multi-Level Array

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

```
movl    8(%ebp), %eax           # index
movl    univ(,%eax,4), %edx     # p = univ[index]
movl    12(%ebp), %eax         # dig
movl    (%edx,%eax,4), %eax     # p[dig]
```

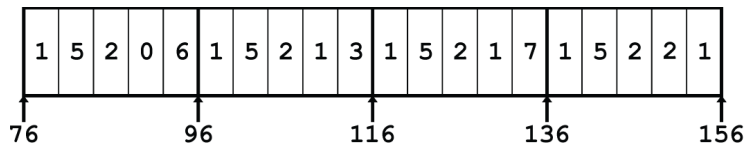
■ Computation (IA32)

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

Array Element Accesses

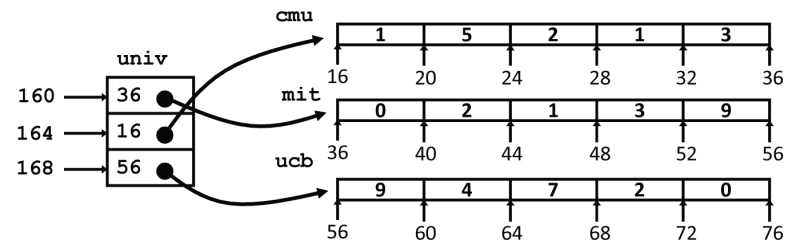
Nested array

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



Multi-level array

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

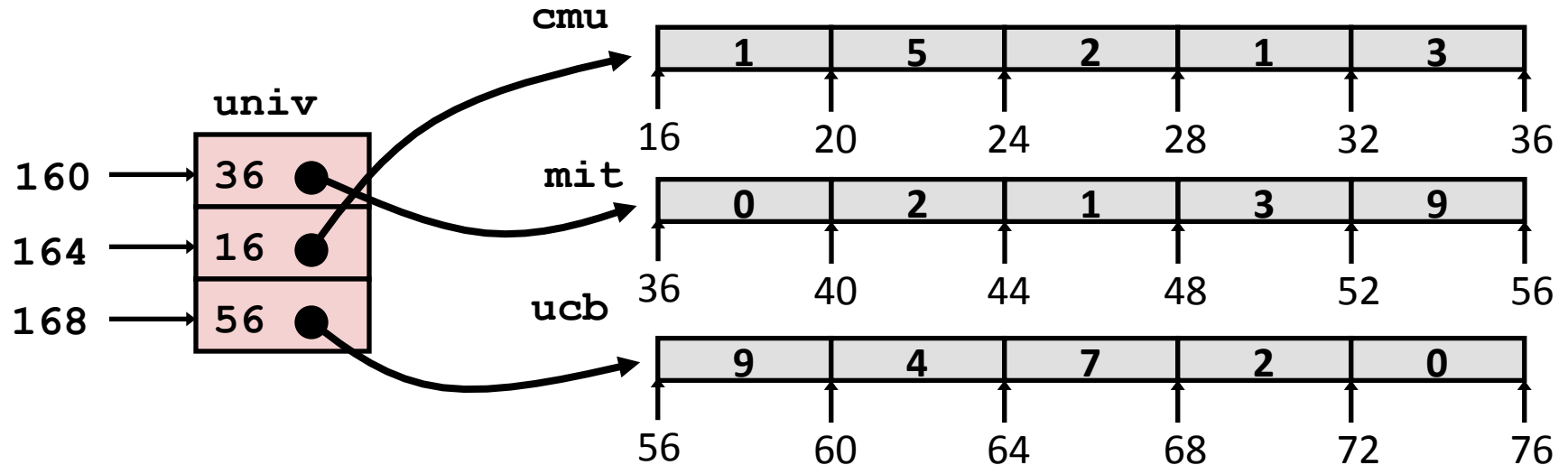


Accesses looks similar in C, but addresses very different:

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

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

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

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

■ Variable dimensions, implicit indexing

- Now supported by gcc

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

16 X 16 Matrix Access

■ Array Elements

- Address $\mathbf{A} + i * (\mathbf{C} * \mathbf{K}) + j * \mathbf{K}$
- $\mathbf{C} = 16, \mathbf{K} = 4$

```

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

```

```

movl    12(%ebp), %edx    # i
sall    $6, %edx         # i*64
movl    16(%ebp), %eax    # j
sall    $2, %eax         # j*4
addl    8(%ebp), %eax     # a + j*4
movl    (%eax,%edx), %eax # *(a + j*4 + i*64)

```

n X n Matrix Access

■ Array Elements

- Address $\mathbf{A} + i * (\mathbf{C} * \mathbf{K}) + j * \mathbf{K}$
- $\mathbf{C} = \mathbf{n}, \mathbf{K} = 4$

```

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

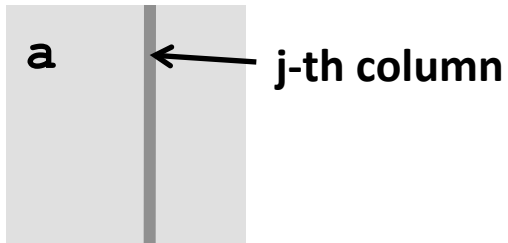
```

```

movl    8(%ebp), %eax    # n
sall    $2, %eax        # n*4
movl    %eax, %edx      # n*4
imull   16(%ebp), %edx   # i*n*4
movl    20(%ebp), %eax   # j
sall    $2, %eax        # j*4
addl    12(%ebp), %eax   # a + j*4
movl    (%eax,%edx), %eax # *(a + j*4 + i*n*4)

```

Optimizing Fixed Array Access



■ Computation

- Step through all elements in column j

■ Optimization

- Retrieving successive elements from single column

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

```
/* Retrieve column j from array */
void fix_column
    (fix_matrix a, int j, int *dest)
{
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
```

Optimizing Fixed Array Access

■ Optimization

- Compute $ajp = \&a[i][j]$
 - Initially $= a + 4*j$
 - Increment by $4*N$

Register	Value
<code>%ecx</code>	<code>ajp</code>
<code>%ebx</code>	<code>dest</code>
<code>%edx</code>	<code>i</code>

```

/* Retrieve column j from array */
void fix_column
    (fix_matrix a, int j, int *dest)
{
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
  
```

```

.L8:                                # loop:
    movl    (%ecx), %eax             #   Read *ajp
    movl    %eax, (%ebx,%edx,4)     #   Save in dest[i]
    addl    $1, %edx                #   i++
    addl    $64, %ecx               #   ajp += 4*N
    cmpl    $16, %edx              #   i:N
    jne     .L8                    #   if !=, goto loop
  
```

Optimizing Variable Array Access

- Compute $ajp = \&a[i][j]$
 - Initially $= a + 4*j$
 - Increment by $4*n$

Register	Value
%ecx	ajp
%edi	dest
%edx	i
%ebx	4*n
%esi	n

```

/* Retrieve column j from array */
void var_column
(int n, int a[n][n],
 int j, int *dest)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = a[i][j];
}
  
```

```

.L18:                                # loop:
    movl    (%ecx), %eax              #   Read *ajp
    movl    %eax, (%edi,%edx,4)      #   Save in dest[i]
    addl    $1, %edx                 #   i++
    addl    $ebx, %ecx               #   ajp += 4*n
    cmpl    $edx, %esi              #   n:i
    jg     .L18                      #   if >, goto loop
  
```

Today

■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

- Allocation
- Access
- Alignment

■ Unions

■ Memory Layout

■ Buffer Overflow

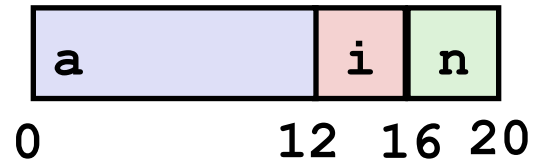
■ Floating Point

Structure Allocation

```

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

Memory Layout



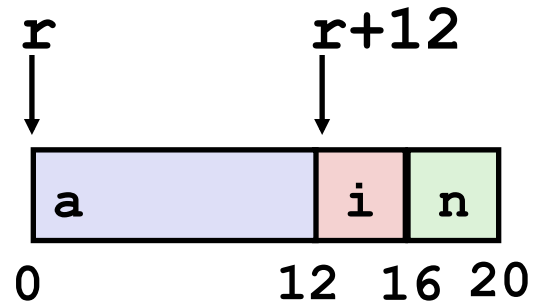
■ Concept

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

Structure Access

```

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



■ Accessing Structure Member

- Pointer indicates first byte of structure
- Access elements with offsets

```

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

IA32 Assembly

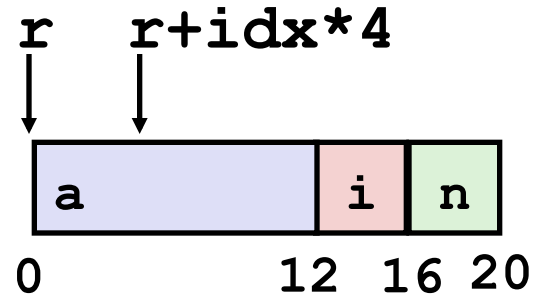
```

# %edx = val
# %eax = r
movl %edx, 12(%eax) # Mem[r+12] = val
    
```

Generating Pointer to Structure Member

```

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



■ Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Arguments
 - Mem[%ebp+8]: **r**
 - Mem[%ebp+12]: **idx**

```

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

```

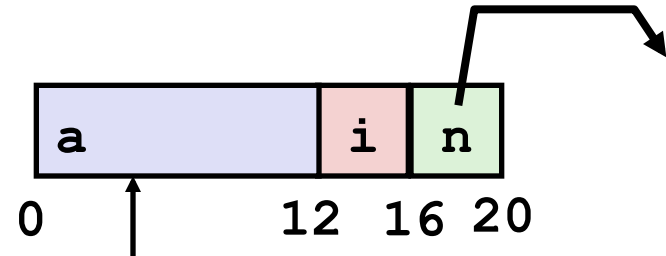
movl    12(%ebp), %eax    # Get idx
sall    $2, %eax         # idx*4
addl    8(%ebp), %eax     # r+idx*4
  
```

Following Linked List

■ C Code

```
void set_val
(struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->n;
    }
}
```

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



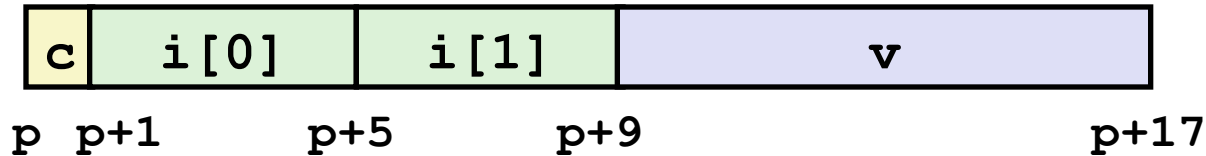
Element i

Register	Value
%edx	r
%ecx	val

```
.L17:                # loop:
    movl    12(%edx), %eax    # r->i
    movl    %ecx, (%edx,%eax,4) # r->a[i] = val
    movl    16(%edx), %edx    # r = r->n
    testl   %edx, %edx       # Test r
    jne     .L17              # If != 0 goto loop
```

Alignment

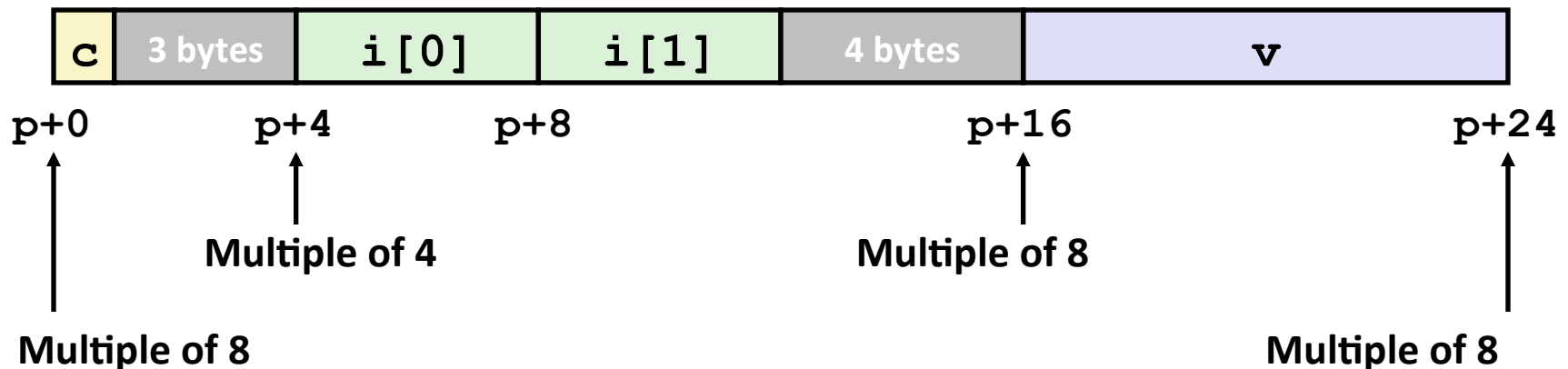
■ Unaligned Data



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

■ Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K



Alignment Principles

■ 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)

- **1 byte: char, ...**
 - no restrictions on address
- **2 bytes: short, ...**
 - lowest 1 bit of address must be 0_2
- **4 bytes: int, float, char *, ...**
 - lowest 2 bits of address must be 00_2
- **8 bytes: 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)

- **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, 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

■ Within structure:

- Must satisfy each element's alignment requirement

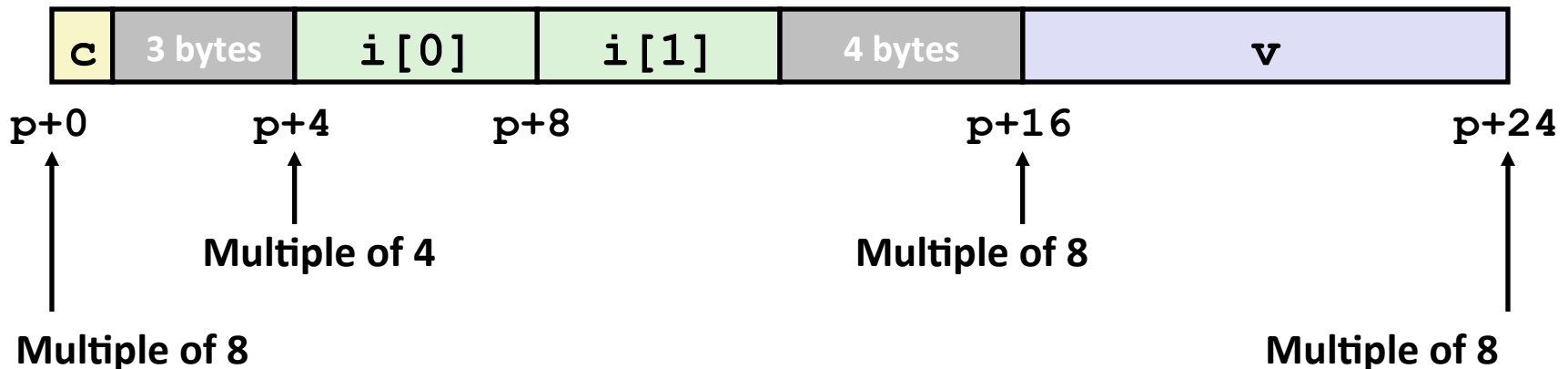
■ Overall structure placement

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

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

■ Example (under Windows or x86-64):

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

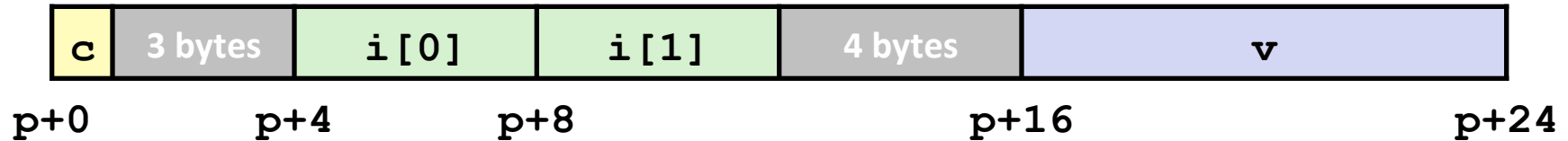


Different Alignment Conventions

■ x86-64 or IA32 Windows:

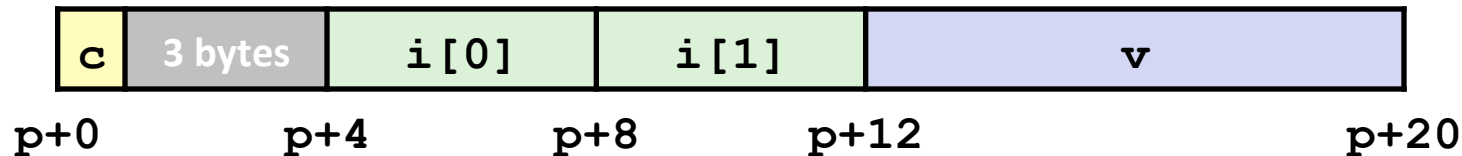
- $K = 8$, due to `double` element

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



■ IA32 Linux

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

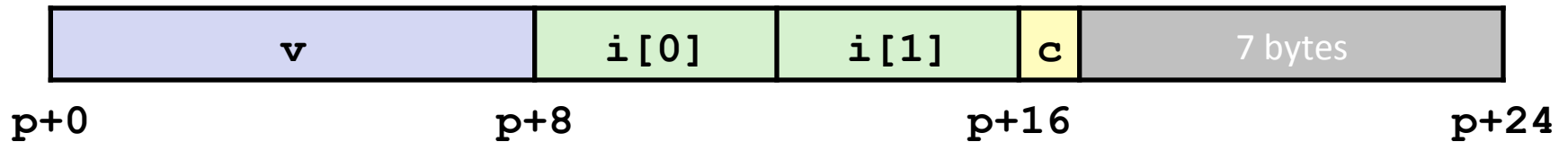


Meeting Overall Alignment Requirement

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

```

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

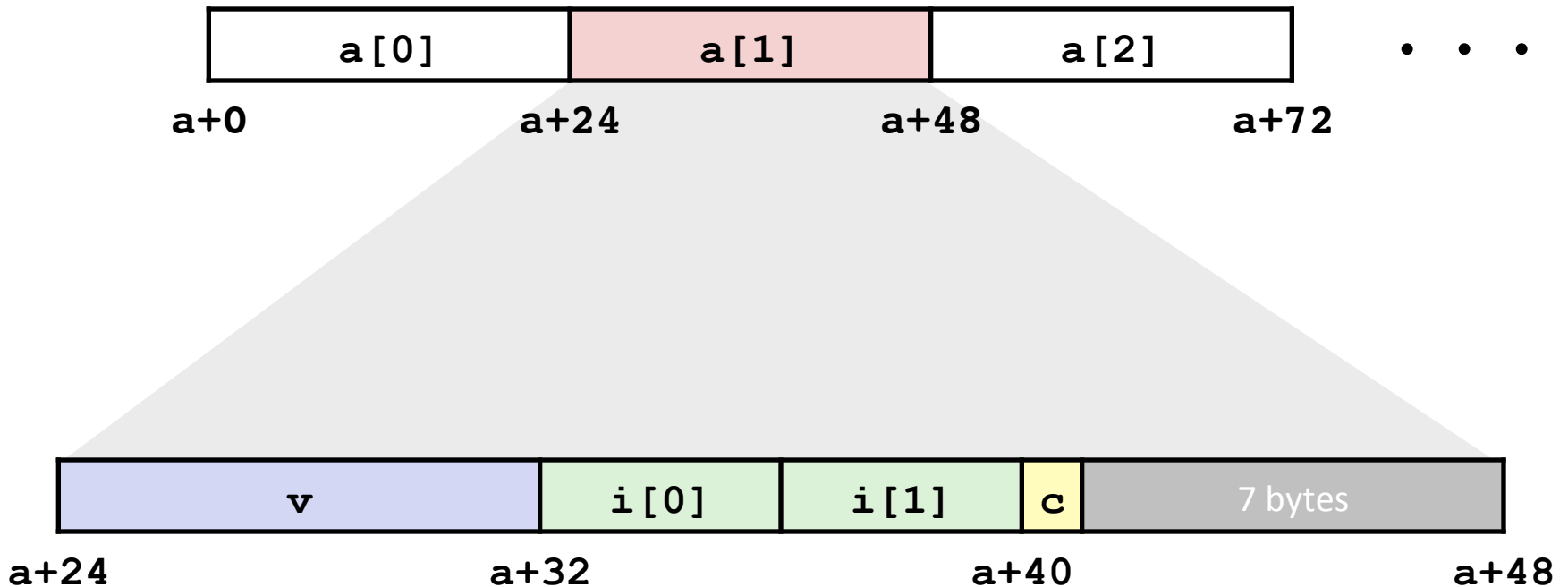


Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```

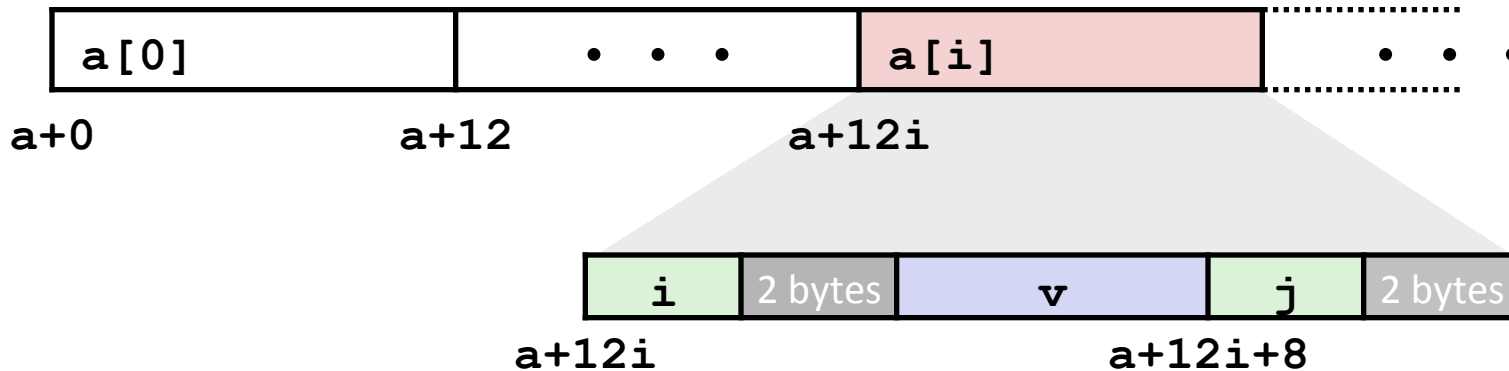
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
  
```



Accessing Array Elements

- **Compute array offset $12i$**
 - `sizeof(S3)`, including alignment spacers
- **Element j is at offset 8 within structure**
- **Assembler gives offset $a+8$**
 - Resolved during linking

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



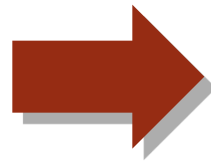
```
short get_j(int idx)
{
    return a[idx].j;
}
```

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

Saving Space

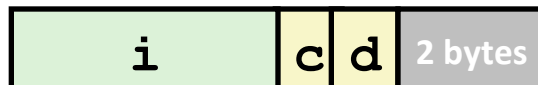
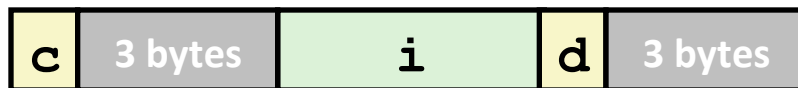
- Put large data types first

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



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

- Effect (K=4)



Today

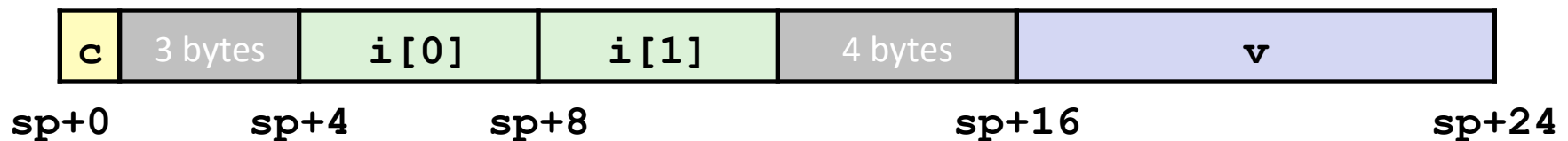
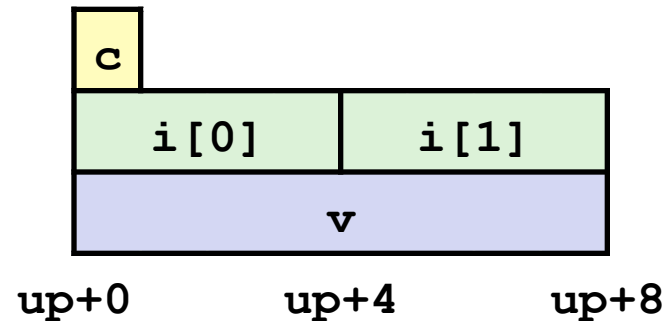
- Arrays
- Structures
- **Unions**
- Memory Layout
- **Buffer Overflow**
 - Vulnerability
 - Protection
- **Floating Point**

Union Allocation

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

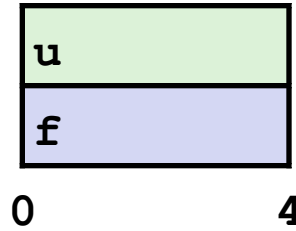
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

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



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;
}
```

Same as (float) u?

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

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
- 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;
```

32-bit

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]							

64-bit

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]);

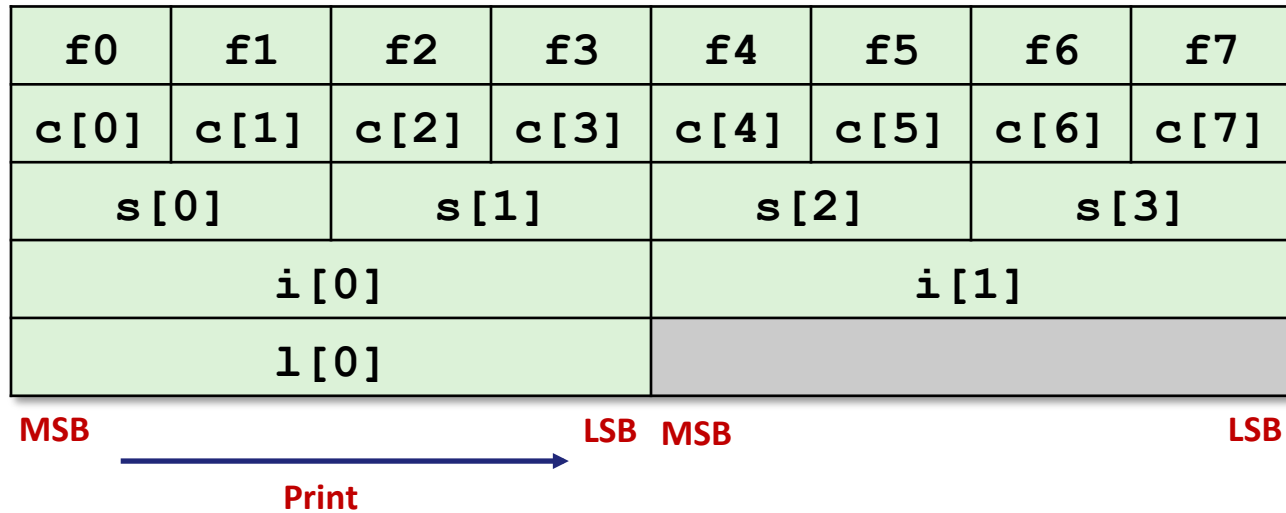
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x] \n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

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 Sun

Big Endian



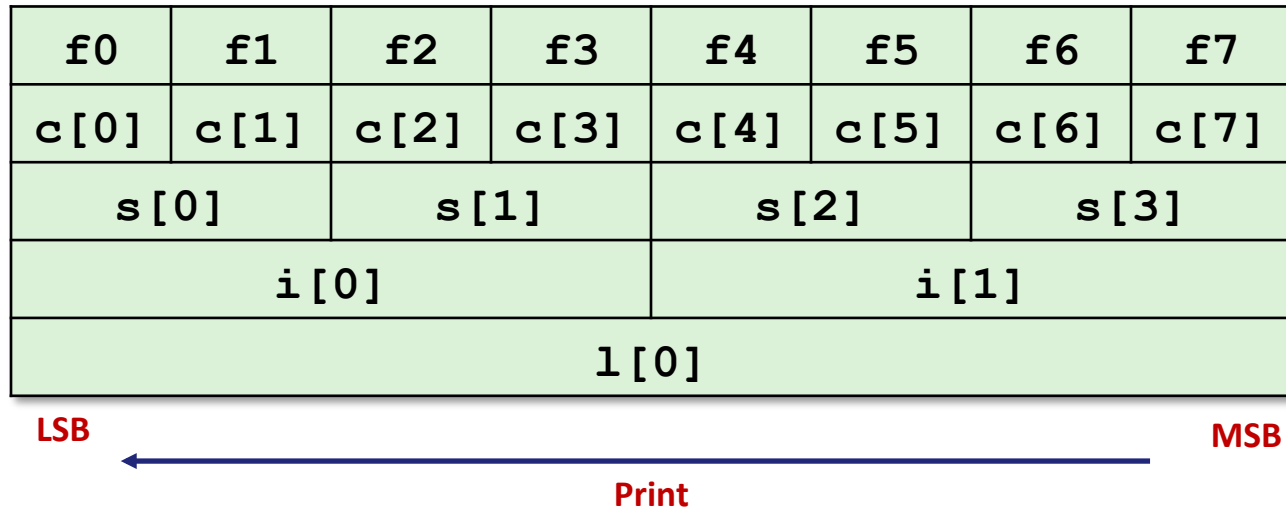
Output on Sun:

```

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]
  
```

Summary

■ Arrays in C

- Contiguous allocation of memory
- Aligned to satisfy every element's alignment requirement
- 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

Today

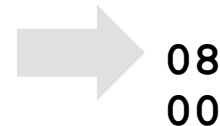
- Arrays
- Structures
- Unions
- **Memory Layout**
- **Buffer Overflow**
 - Vulnerability
 - Protection
- **Floating Point**

IA32 Linux Memory Layout

- **Stack**
 - Runtime stack (8MB limit)
 - E. g., local variables
- **Heap**
 - Dynamically allocated storage
 - When call `malloc()`, `calloc()`, `new()`
- **Data**
 - Statically allocated data
 - E.g., arrays & strings declared in code
- **Text**
 - Executable machine instructions
 - Read-only

not drawn to scale

Upper 2 hex digits
= 8 bits of address



not drawn to scale

Memory Allocation Example

```

char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */

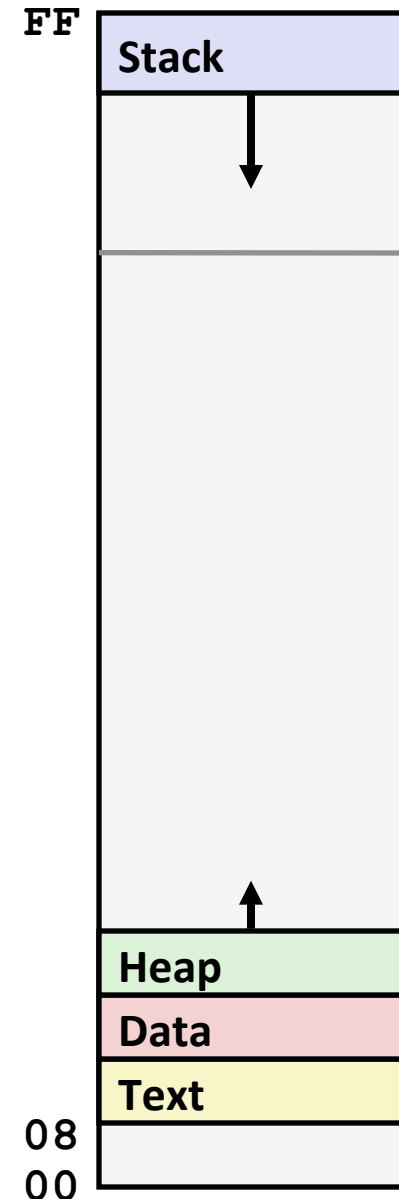
int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main()
{
    p1 = malloc(1 <<28); /* 256 MB */
    p2 = malloc(1 << 8); /* 256 B */
    p3 = malloc(1 <<28); /* 256 MB */
    p4 = malloc(1 << 8); /* 256 B */
    /* Some print statements ... */
}

```

Where does everything go?



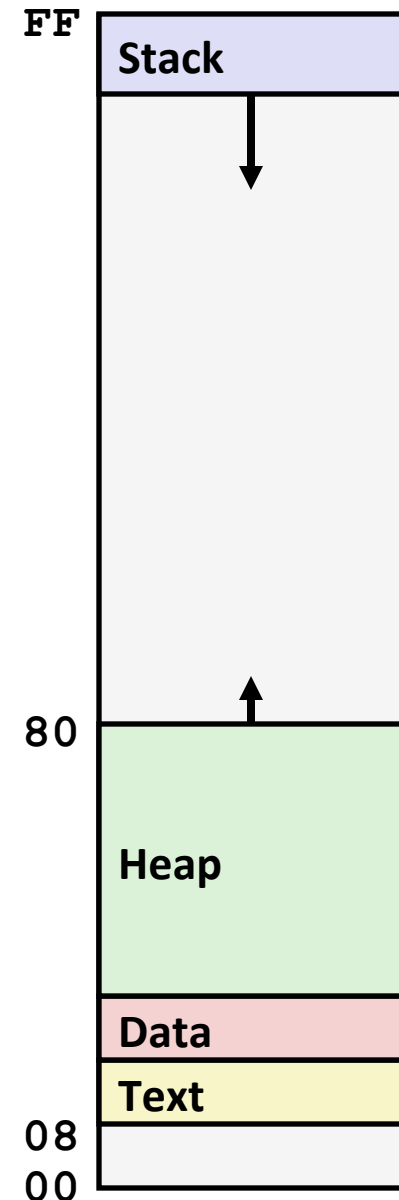
not drawn to scale

IA32 Example Addresses

address range $\sim 2^{32}$

<code>\$esp</code>	<code>0xffffbcd0</code>
<code>p3</code>	<code>0x65586008</code>
<code>p1</code>	<code>0x55585008</code>
<code>p4</code>	<code>0x1904a110</code>
<code>p2</code>	<code>0x1904a008</code>
<code>&p2</code>	<code>0x18049760</code>
<code>&beyond</code>	<code>0x08049744</code>
<code>big_array</code>	<code>0x18049780</code>
<code>huge_array</code>	<code>0x08049760</code>
<code>main()</code>	<code>0x080483c6</code>
<code>useless()</code>	<code>0x08049744</code>
<code>final malloc()</code>	<code>0x006be166</code>

`malloc()` is dynamically linked
address determined at runtime



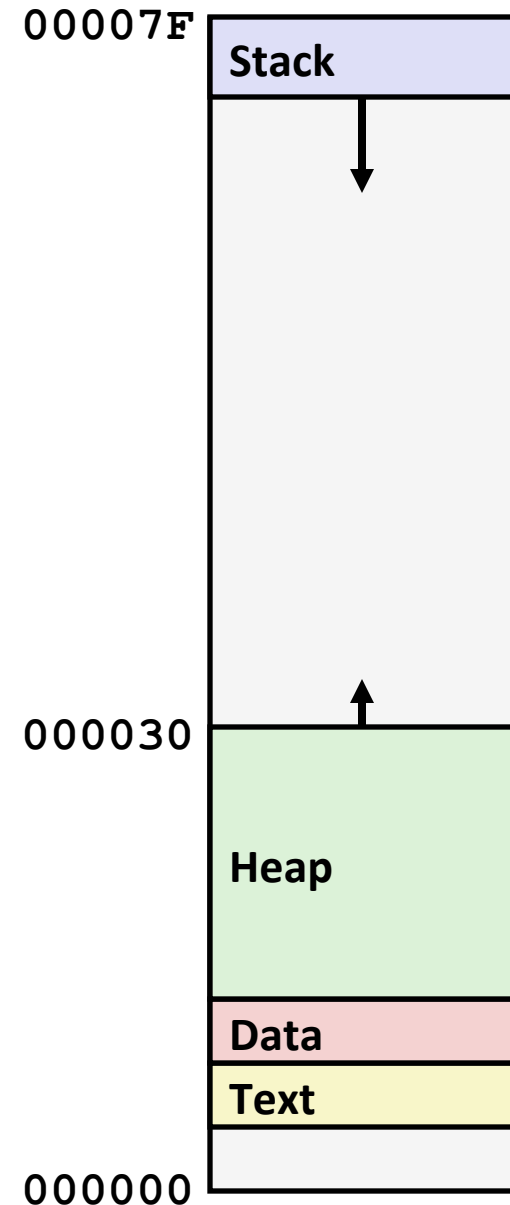
x86-64 Example Addresses

address range $\sim 2^{47}$

<code>\$rsp</code>	<code>0x00007fffffff8d1f8</code>
<code>p3</code>	<code>0x00002aaabaadd010</code>
<code>p1</code>	<code>0x00002aaaaadc010</code>
<code>p4</code>	<code>0x0000000011501120</code>
<code>p2</code>	<code>0x0000000011501010</code>
<code>&p2</code>	<code>0x0000000010500a60</code>
<code>&beyond</code>	<code>0x0000000000500a44</code>
<code>big_array</code>	<code>0x0000000010500a80</code>
<code>huge_array</code>	<code>0x0000000000500a50</code>
<code>main()</code>	<code>0x0000000000400510</code>
<code>useless()</code>	<code>0x0000000000400500</code>
<code>final malloc()</code>	<code>0x000000386ae6a170</code>

`malloc()` is dynamically linked
address determined at runtime

not drawn to scale



Today

- Arrays
- Structures
- Structures
- Unions
- Memory Layout
- **Buffer Overflow**
 - Vulnerability
 - Protection
- Floating Point

Internet Worm and IM War

■ November, 1988

- Internet Worm attacks thousands of Internet hosts.
- How did it happen?

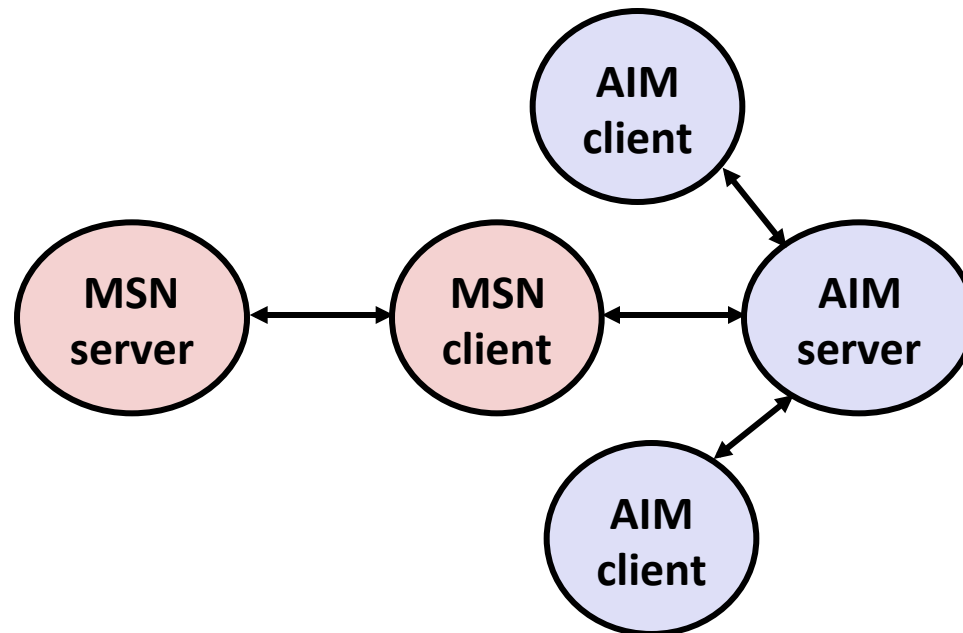
Internet Worm and IM War

■ November, 1988

- Internet Worm attacks thousands of Internet hosts.
- How did it happen?

■ July, 1999

- Microsoft launches MSN Messenger (instant messaging system).
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



Internet Worm and IM War (cont.)

■ August 1999

- Mysteriously, Messenger clients can no longer access AIM servers.
- Microsoft and AOL begin the IM war:
 - AOL changes server to disallow Messenger clients
 - Microsoft makes changes to clients to defeat AOL changes.
 - At least 13 such skirmishes.
- How did it happen?

■ The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!

- many library functions do not check argument sizes.
- allows target buffers to overflow.

String Library Code

■ Implementation of Unix function gets ()

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

- No way to specify limit on number of characters to read
- **Similar problems with other library functions**
 - **strcpy, strcat**: Copy strings 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);  
}
```

```
void call_echo() {  
    echo();  
}
```

```
unix>./bufdemo  
Type a string:123  
1234567
```

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

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

Buffer Overflow Disassembly

echo:

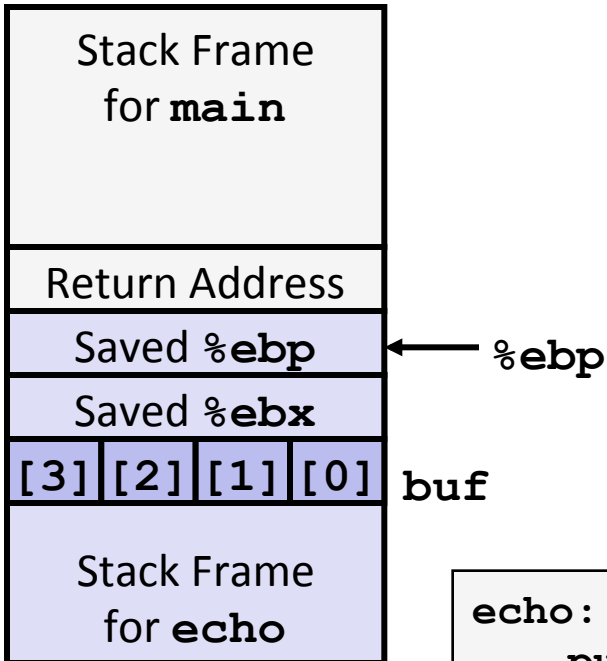
80485c5:	55	push	%ebp
80485c6:	89 e5	mov	%esp, %ebp
80485c8:	53	push	%ebx
80485c9:	83 ec 14	sub	\$0x14, %esp
80485cc:	8d 5d f8	lea	0xffffffff8(%ebp), %ebx
80485cf:	89 1c 24	mov	%ebx, (%esp)
80485d2:	e8 9e ff ff ff	call	8048575 <gets>
80485d7:	89 1c 24	mov	%ebx, (%esp)
80485da:	e8 05 fe ff ff	call	80483e4 <puts@plt>
80485df:	83 c4 14	add	\$0x14, %esp
80485e2:	5b	pop	%ebx
80485e3:	5d	pop	%ebp
80485e4:	c3	ret	

call_echo:

80485eb:	e8 d5 ff ff ff	call	80485c5 <echo>
80485f0:	c9	leave	
80485f1:	c3	ret	

Buffer Overflow Stack

Before call to gets



```

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

```

```

echo:
    pushl %ebp           # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx          # Save %ebx
    subl  $20, %esp     # Allocate stack space
    leal  -8(%ebp), %ebx # Compute buf as %ebp-8
    movl  %ebx, (%esp)  # Push buf on stack
    call  gets          # Call gets
    . . .

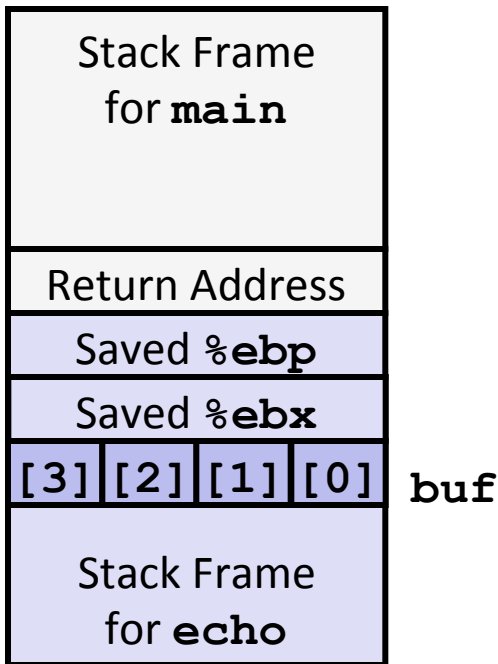
```

Buffer Overflow Stack Example

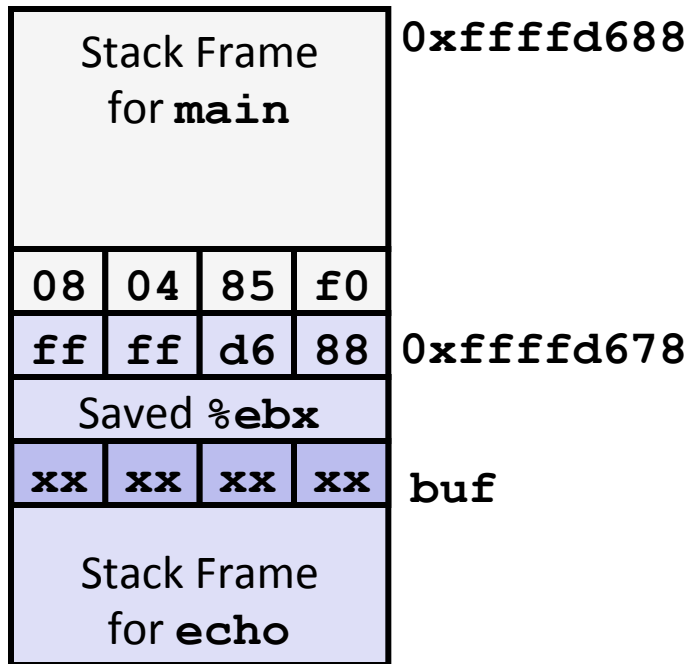
```

unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffd688
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0
    
```

Before call to gets



Before call to gets



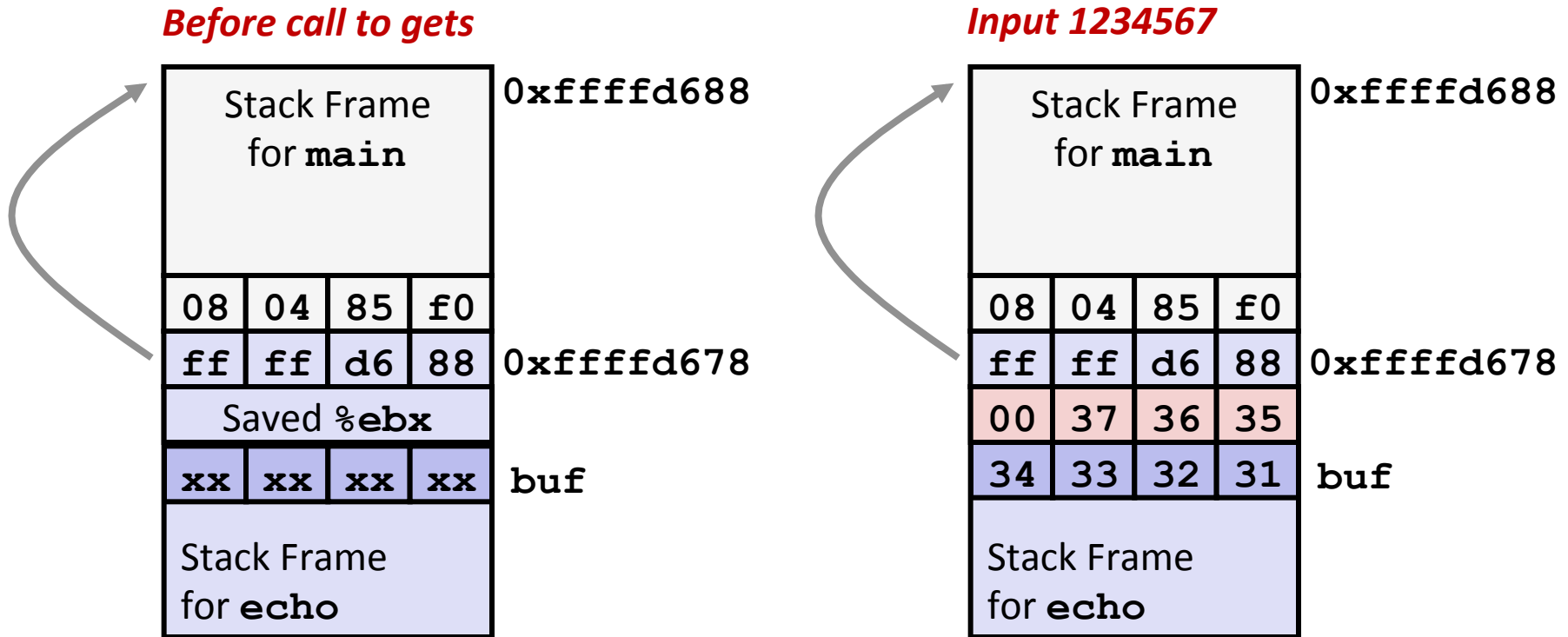
```

80485eb: e8 d5 ff ff ff
80485f0: c9
    
```

```

call 80485c5 <echo>
leave
    
```

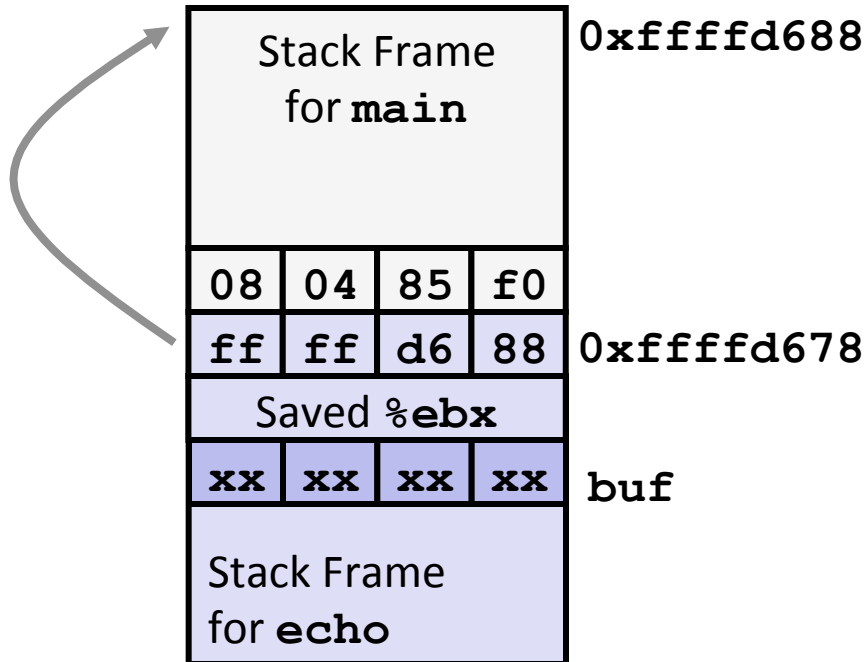
Buffer Overflow Example #1



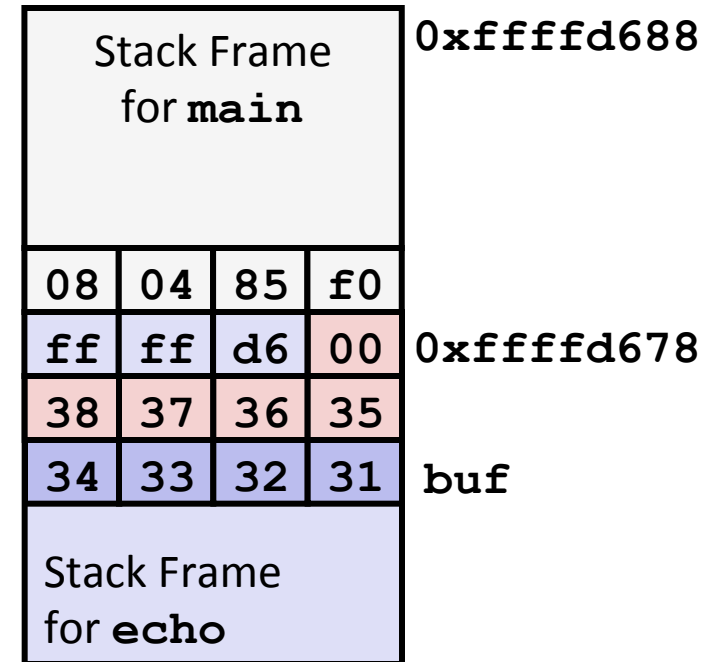
**Overflow buf, and corrupt `%ebx`,
but no problem**

Buffer Overflow Example #2

Before call to gets



Input 12345678



Base pointer corrupted

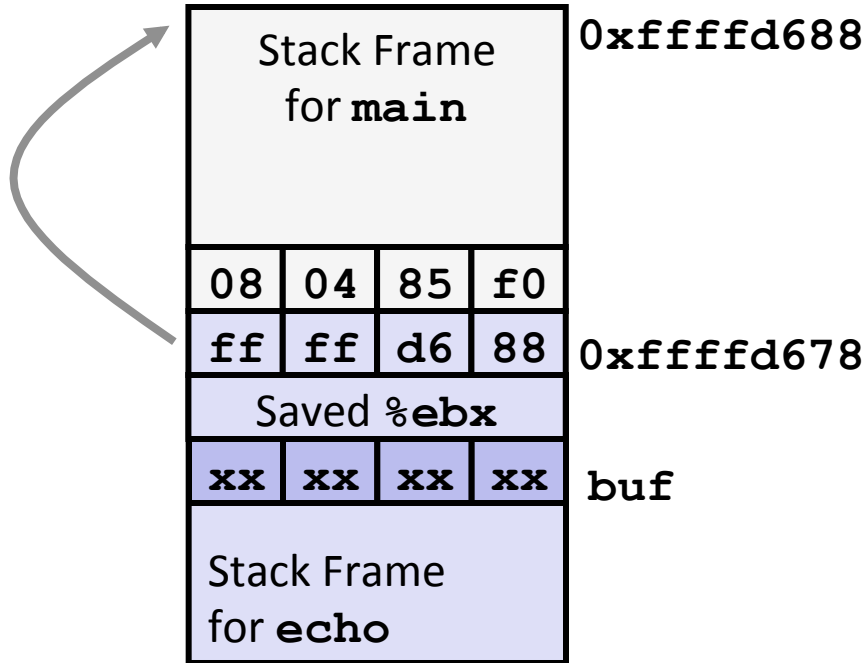
```

. . .
80485eb:  e8 d5 ff ff ff  call  80485c5 <echo>
80485f0:  c9                leave # Set %ebp to corrupted value
80485f1:  c3                ret

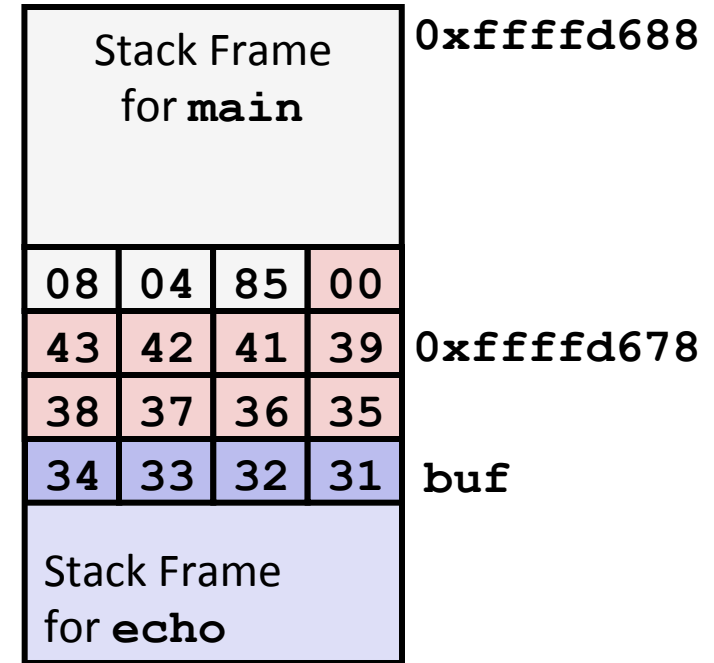
```

Buffer Overflow Example #3

Before call to gets



Input 123456789



Return address corrupted

```

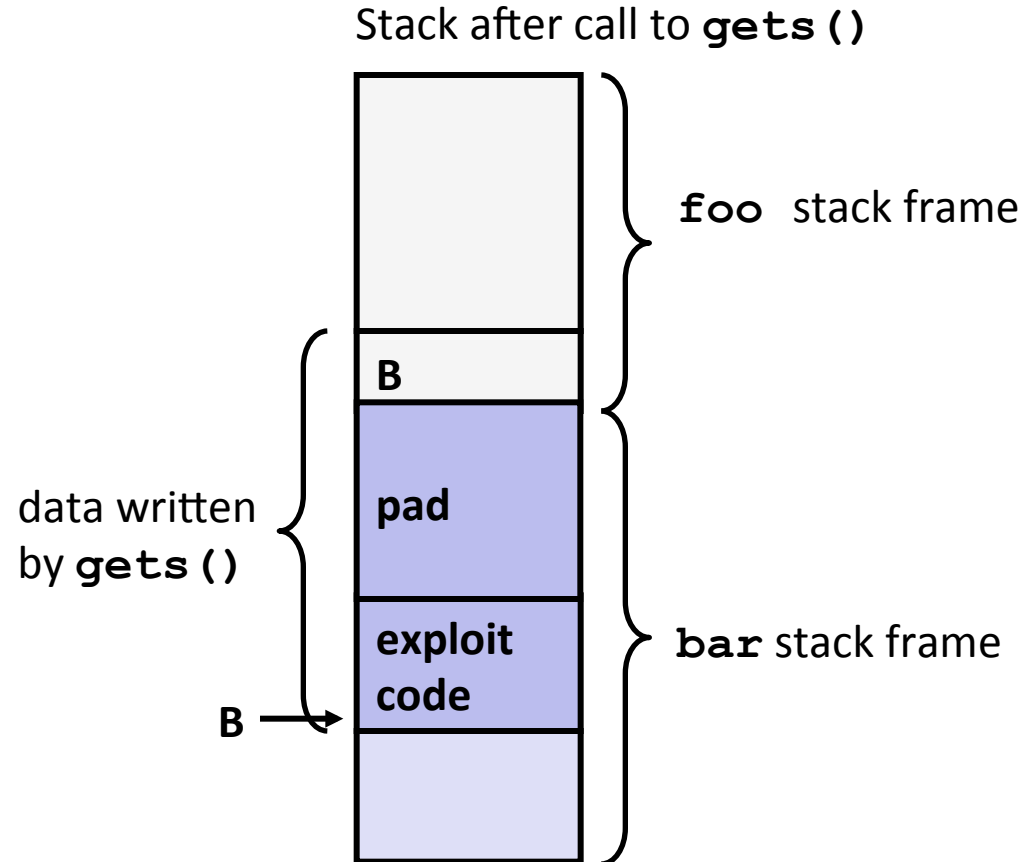
80485eb:  e8 d5 ff ff ff  call  80485c5 <echo>
80485f0:  c9                leave # Desired return point
    
```

Malicious Use of Buffer Overflow

```
void foo() {
    bar();
    ...
}
```

return
address
A

```
int bar() {
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```



- Input string contains byte representation of executable code
- Overwrite return address `A` with address of buffer `B`
- 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.

Exploits Based on Buffer Overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- **IM War**
 - AOL exploited existing buffer overflow bug in AIM clients
 - exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
 - When Microsoft changed code to match signature, AOL changed signature location.

Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.

...

It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.

....

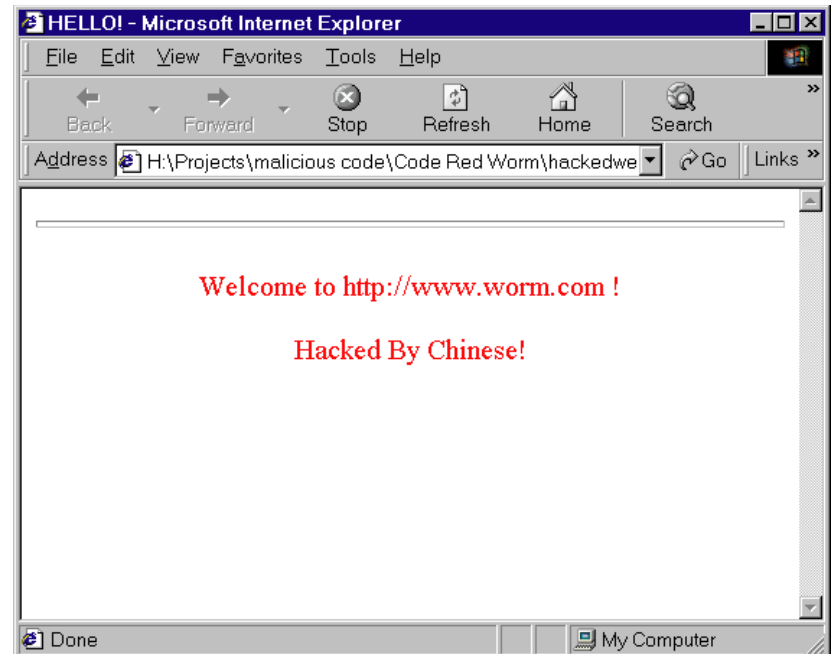
Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com

***It was later determined that this
email originated from within
Microsoft!***

Code Red Exploit Code

- Starts 100 threads running
- Spread self
 - Generate random IP addresses & send attack string
 - Between 1st & 19th of month
- Attack **www.whitehouse.gov**
 - Send 98,304 packets; sleep for 4-1/2 hours; repeat
 - Denial of service attack
 - Between 21st & 27th of month
- Deface server's home page
 - After waiting 2 hours



Avoiding Overflow Vulnerability

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

- **Use library routines that limit string lengths**
 - **fgets** instead of **gets**
 - **strncpy** instead of **strcpy**
 - Don't use **scanf** with **%s** conversion specification
 - Use **fgets** to read the string
 - Or use **%ns** where **n** is a suitable integer

System-Level Protections

■ Randomized stack offsets

- At start of program, allocate random amount of space on stack
- Makes it difficult for hacker to predict beginning of inserted code

■ Nonexecutable code segments

- In traditional x86, can mark region of memory as either “read-only” or “writeable”
 - Can execute anything readable
- X86-64 added explicit “execute” permission

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

Stack Canaries

■ Idea

- Place special value (“canary”) on stack just beyond buffer
- Check for corruption before exiting function

■ GCC Implementation

- `-fstack-protector`
- `-fstack-protector-all`

```
unix> ./bufdemo-protected
Type a string:1234
1234
```

```
unix> ./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

Protected Buffer Disassembly

echo:

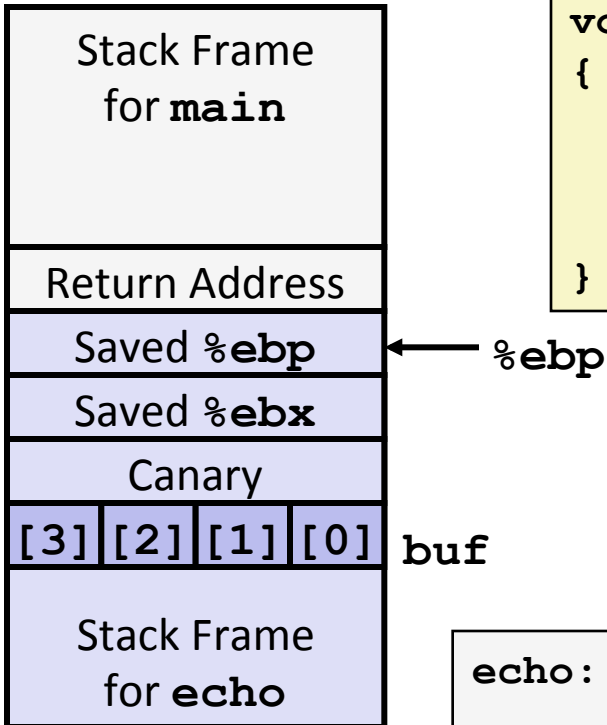
```

804864d:  55          push   %ebp
804864e:  89 e5      mov    %esp,%ebp
8048650:  53        push   %ebx
8048651:  83 ec 14   sub    $0x14,%esp
8048654:  65 a1 14 00 00 00  mov   %gs:0x14,%eax
804865a:  89 45 f8   mov    %eax,0xffffffff8(%ebp)
804865d:  31 c0     xor    %eax,%eax
804865f:  8d 5d f4   lea   0xffffffff4(%ebp),%ebx
8048662:  89 1c 24   mov    %ebx,(%esp)
8048665:  e8 77 ff ff ff  call  80485e1 <gets>
804866a:  89 1c 24   mov    %ebx,(%esp)
804866d:  e8 ca fd ff ff  call  804843c <puts@plt>
8048672:  8b 45 f8   mov    0xffffffff8(%ebp),%eax
8048675:  65 33 05 14 00 00 00  xor   %gs:0x14,%eax
804867c:  74 05     je     8048683 <echo+0x36>
804867e:  e8 a9 fd ff ff  call  804842c <FAIL>
8048683:  83 c4 14   add    $0x14,%esp
8048686:  5b        pop    %ebx
8048687:  5d        pop    %ebp
8048688:  c3        ret

```

Setting Up Canary

Before call to gets



```

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

```

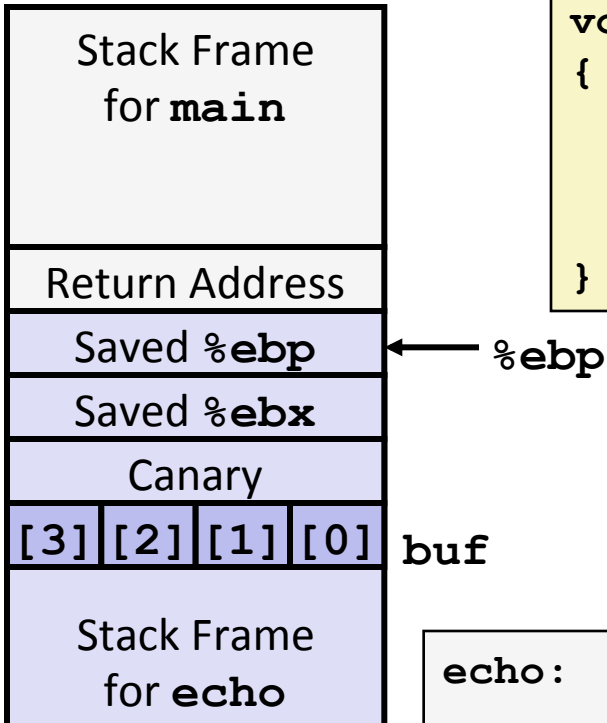
```

echo:
    . . .
    movl    %gs:20, %eax    # Get canary
    movl    %eax, -8(%ebp)  # Put on stack
    xorl    %eax, %eax     # Erase canary
    . . .

```

Checking Canary

Before call to gets



```

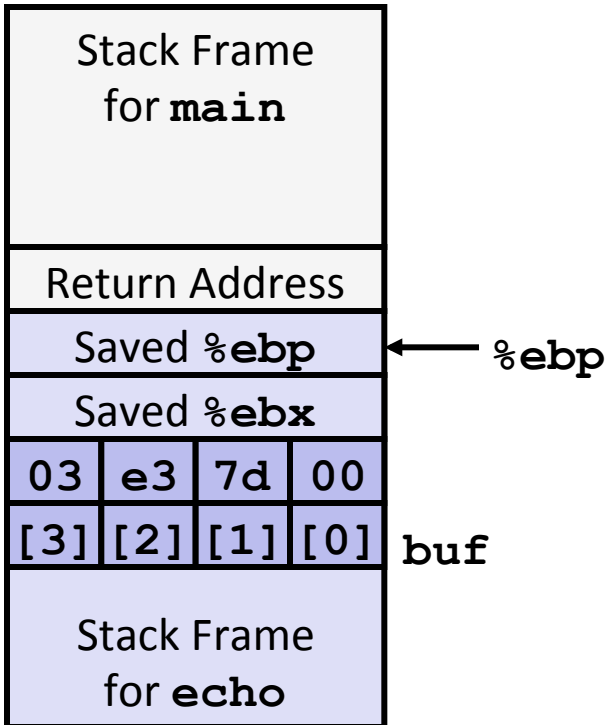
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
  
```

```

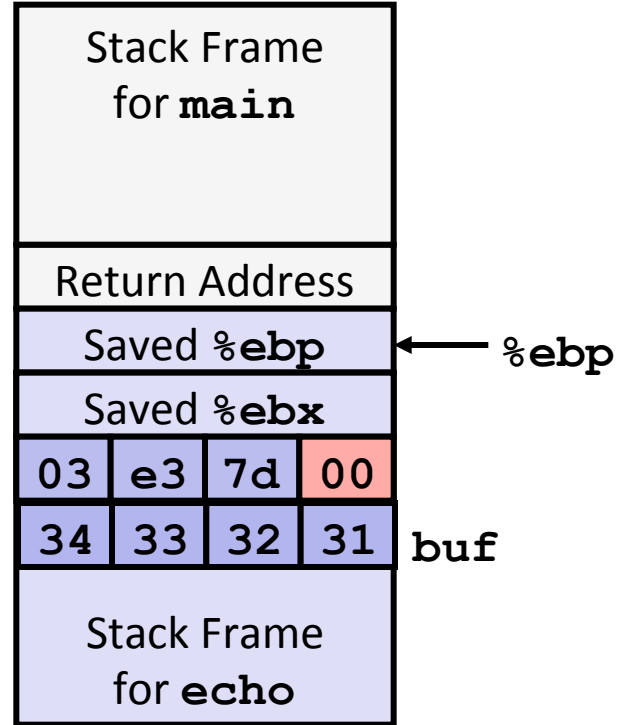
echo:
    . . .
    movl    -8(%ebp), %eax    # Retrieve from stack
    xorl    %gs:20, %eax     # Compare with Canary
    je     .L24              # Same: skip ahead
    call   __stack_chk_fail # ERROR
.L24:
    . . .
  
```

Canary Example

Before call to gets



Input 1234



```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

Benign corruption!
 (allows programmers to make
 silent off-by-one errors)

Worms and Viruses

- **Worm: A program that**
 - Can run by itself
 - Can propagate a fully working version of itself to other computers

- **Virus: Code that**
 - Add itself to other programs
 - Cannot run independently

- **Both are (usually) designed to spread among computers and to wreak havoc**

Reading Assignment

[Smashing the Stack for Fun and Profit, Aleph One](#)

<http://www.phrack.com/issues.html?issue=49&id=14>