Machine-Level Programming V: Advanced Topics

Lecture 5, March 17, 2011
Alexandre David

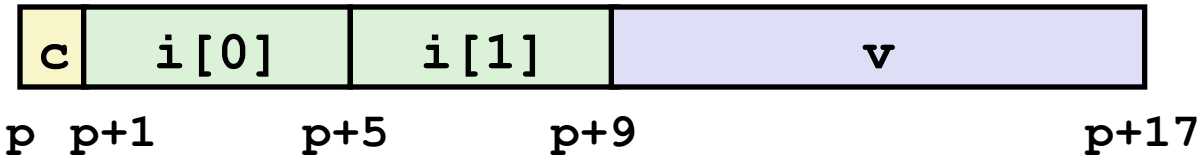
Credits to Randy Bryant & Dave O'Hallaron from Carnegie Mellon

Today

- **Structures**
  - Alignment
- **Unions – self reading if you can**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

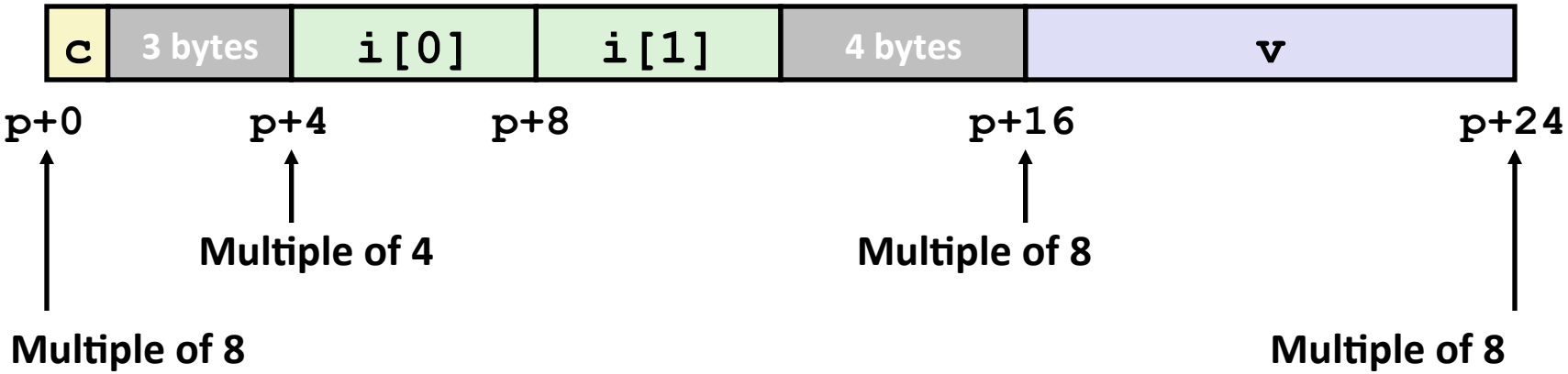
Structures & Alignment

- **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

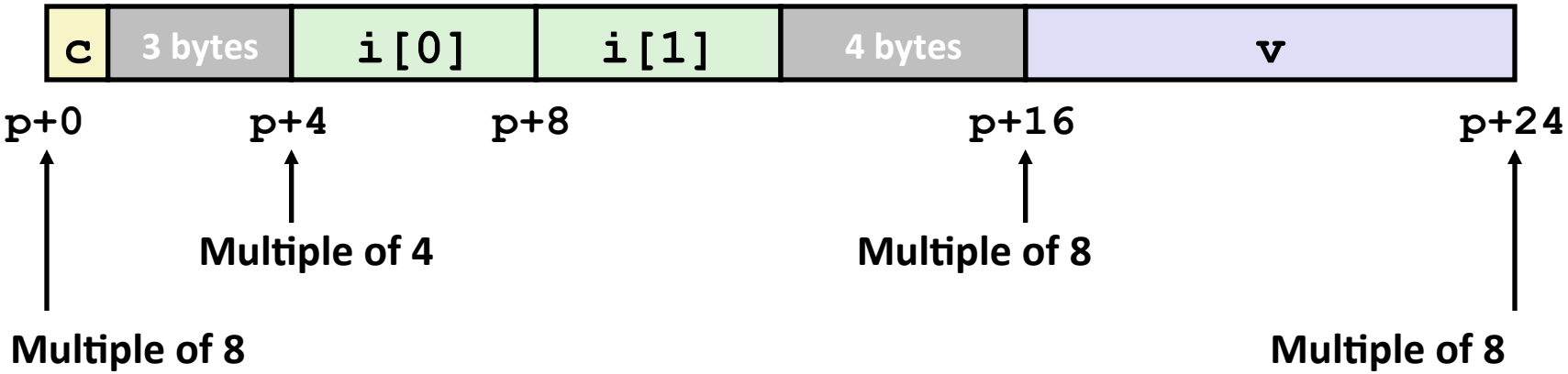
Specific Cases of Alignment (x86-64)

- **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
- **Example (under Windows or x86-64):**
  - K = 8, due to `double` element

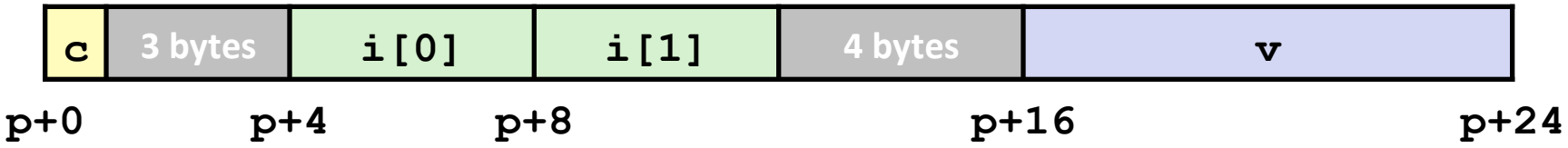
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



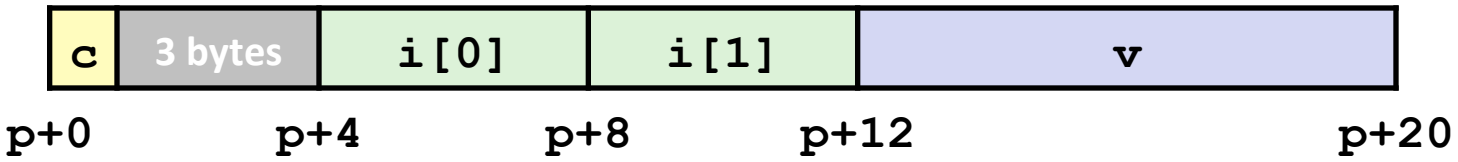
Different Alignment Conventions

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

- **x86-64 or IA32 Windows:**
  - K = 8, due to **`double`** element



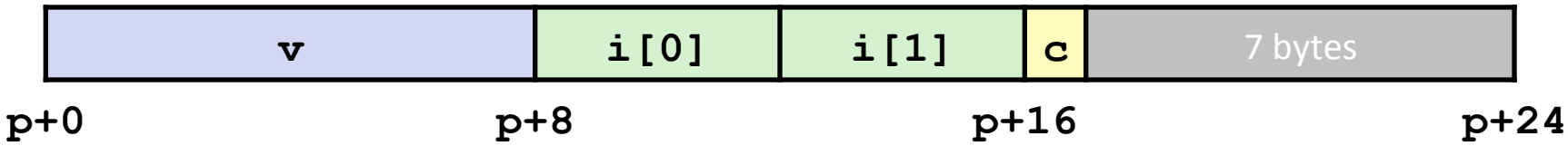
- **IA32 Linux**
  - K = 4; `double` treated like a 4-byte data type



Meeting Overall Alignment Requirement

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

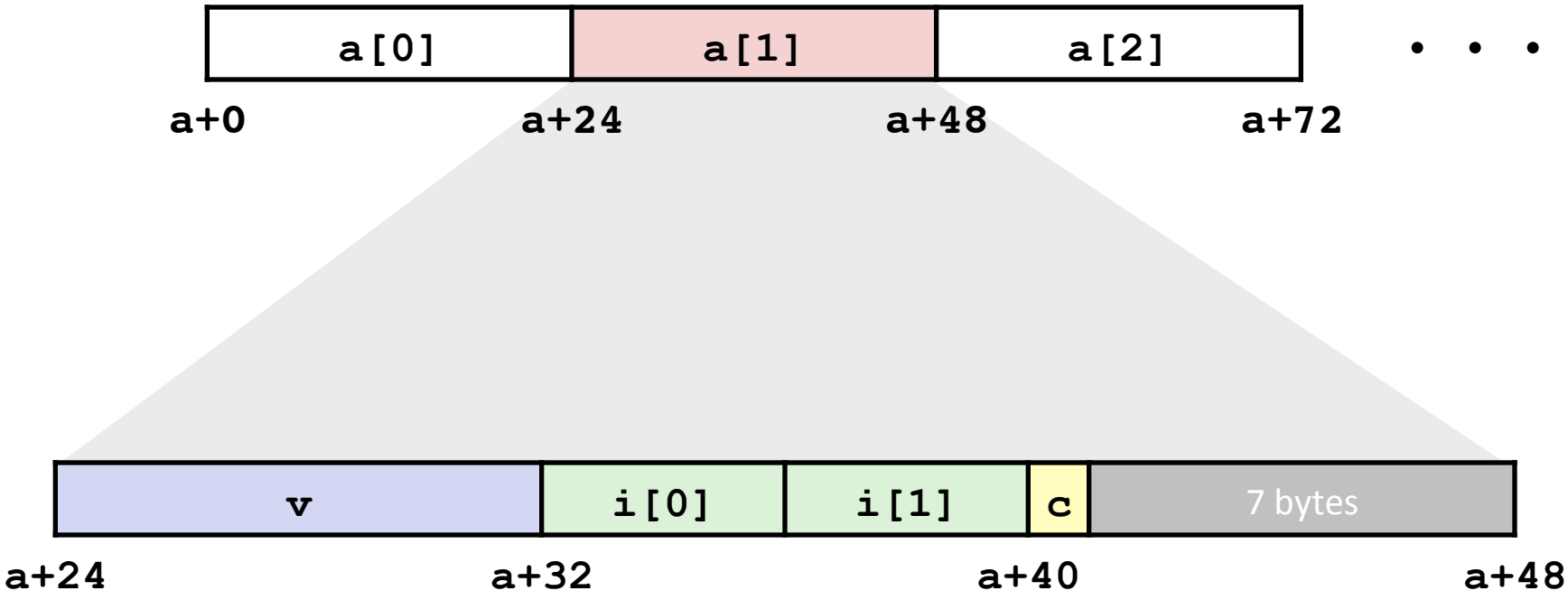
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

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

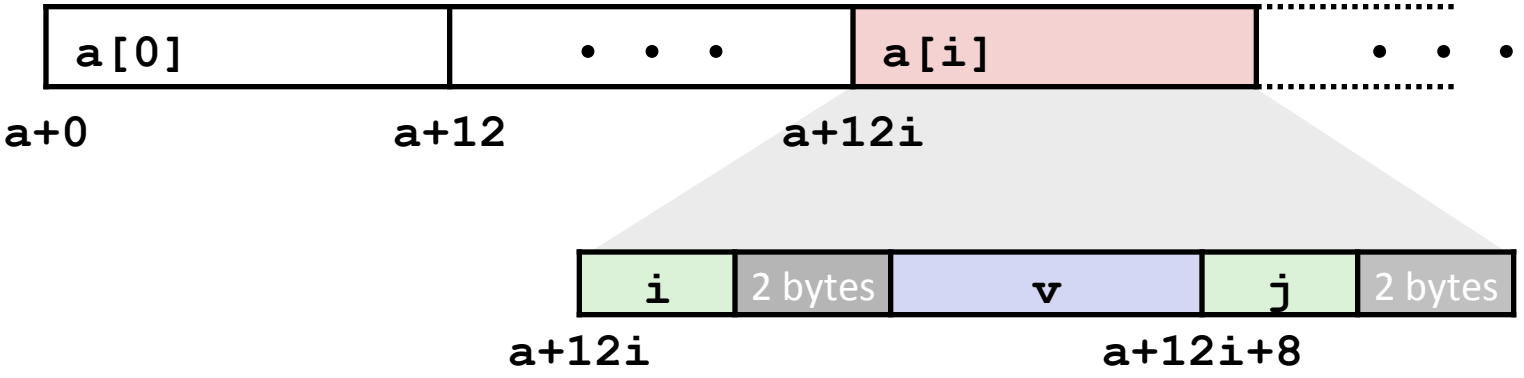
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

- **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




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

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

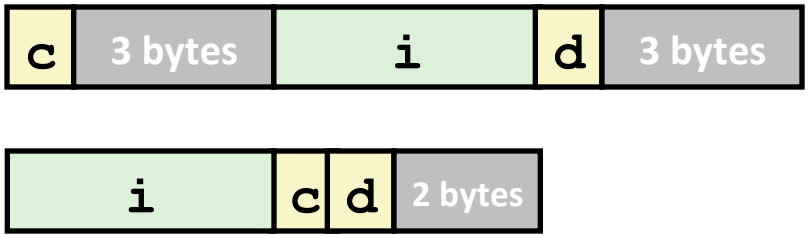
Saving Space

- 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

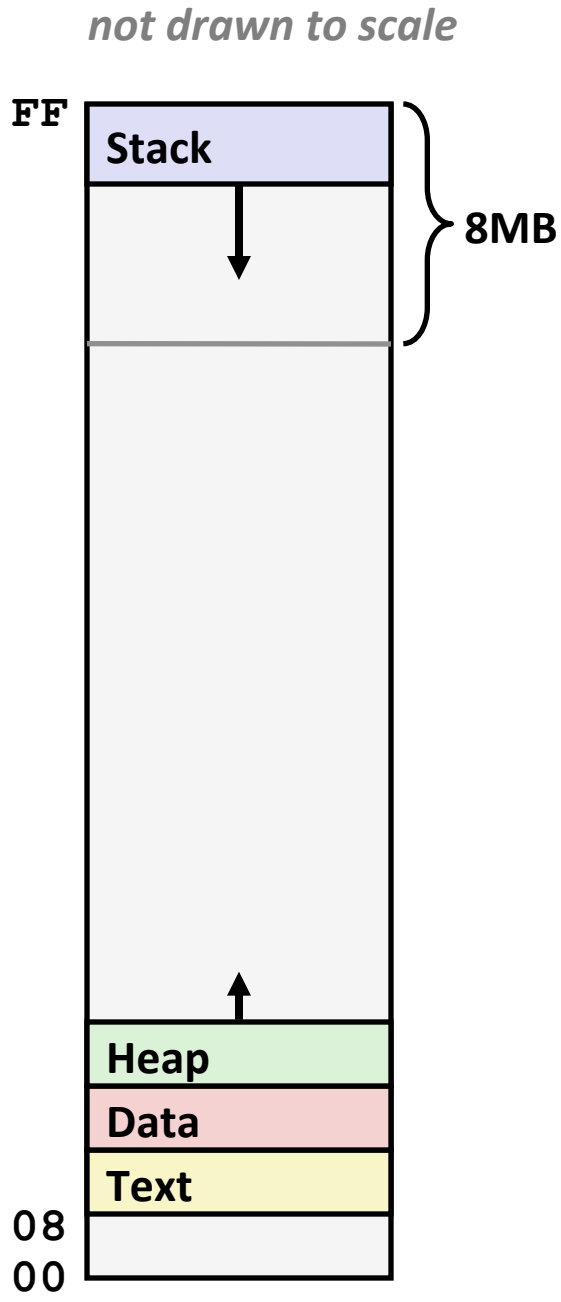
- **Structures**
  - Alignment
- **Unions – self reading if you can**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

Today

- **Structures**
  - Alignment
- **Unions**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

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



Upper 2 hex digits
= 8 bits of address

Memory Allocation Example

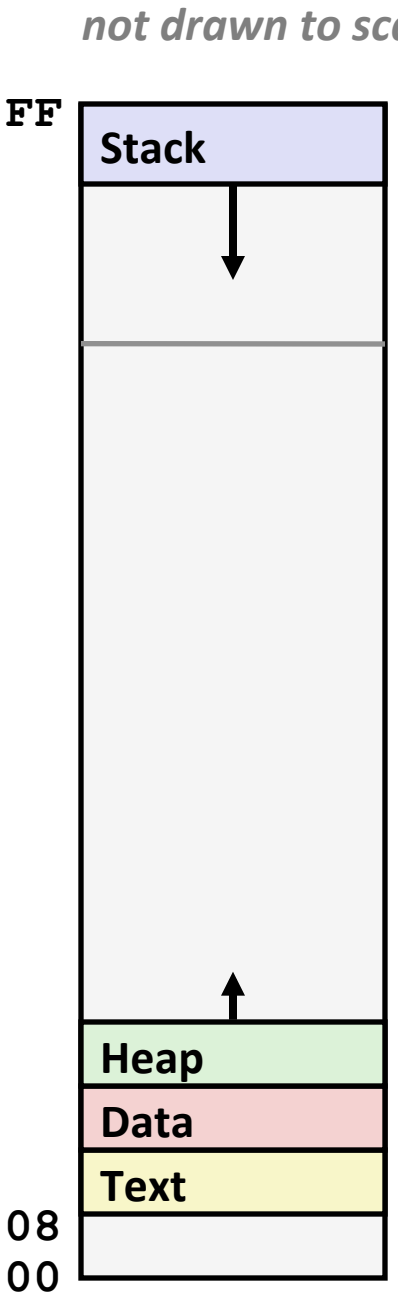
```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

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?



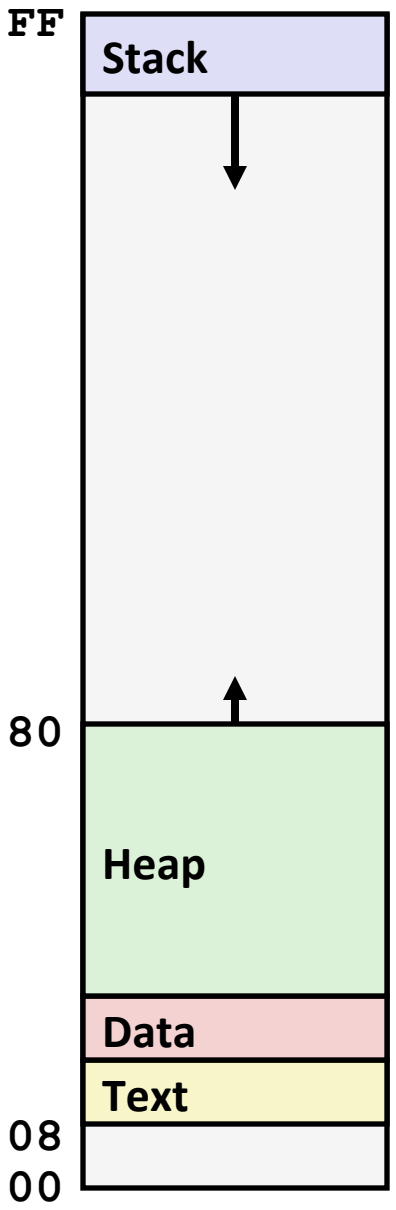
IA32 Example Addresses

address range ~2^{32}

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

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

not drawn to scale



Today

- **Structures**
  - Alignment
- **Unions**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

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();
}
```

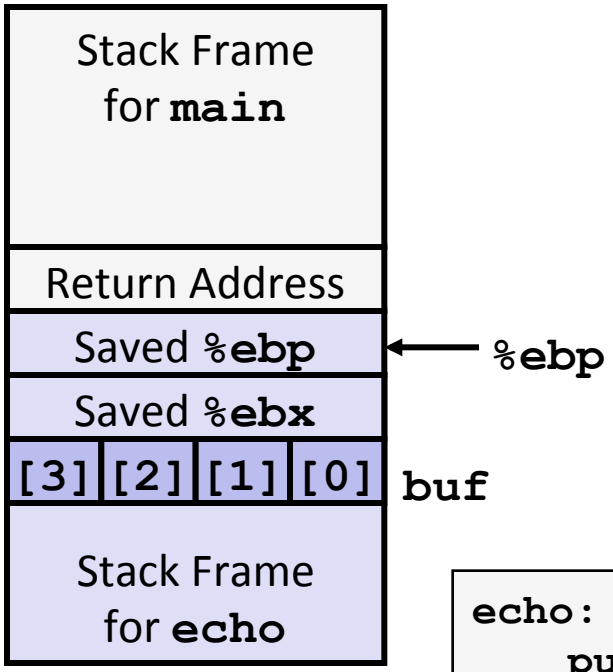
```
unix>./bufdemo
Type a string:1234567
1234567
```

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

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

Buffer Overflow Stack

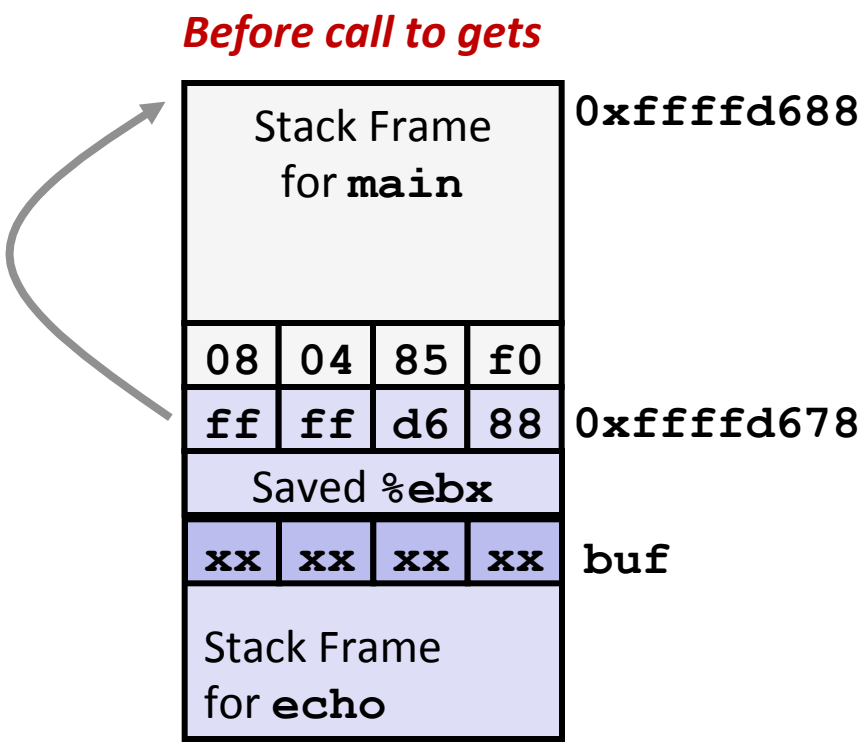
Before call to gets

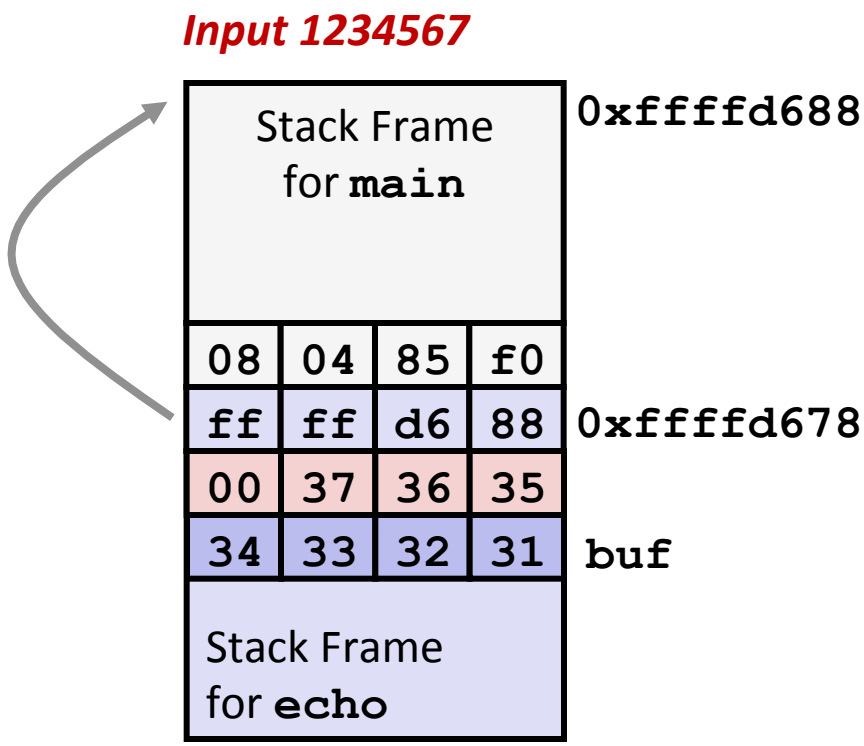



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
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
    . . .
```

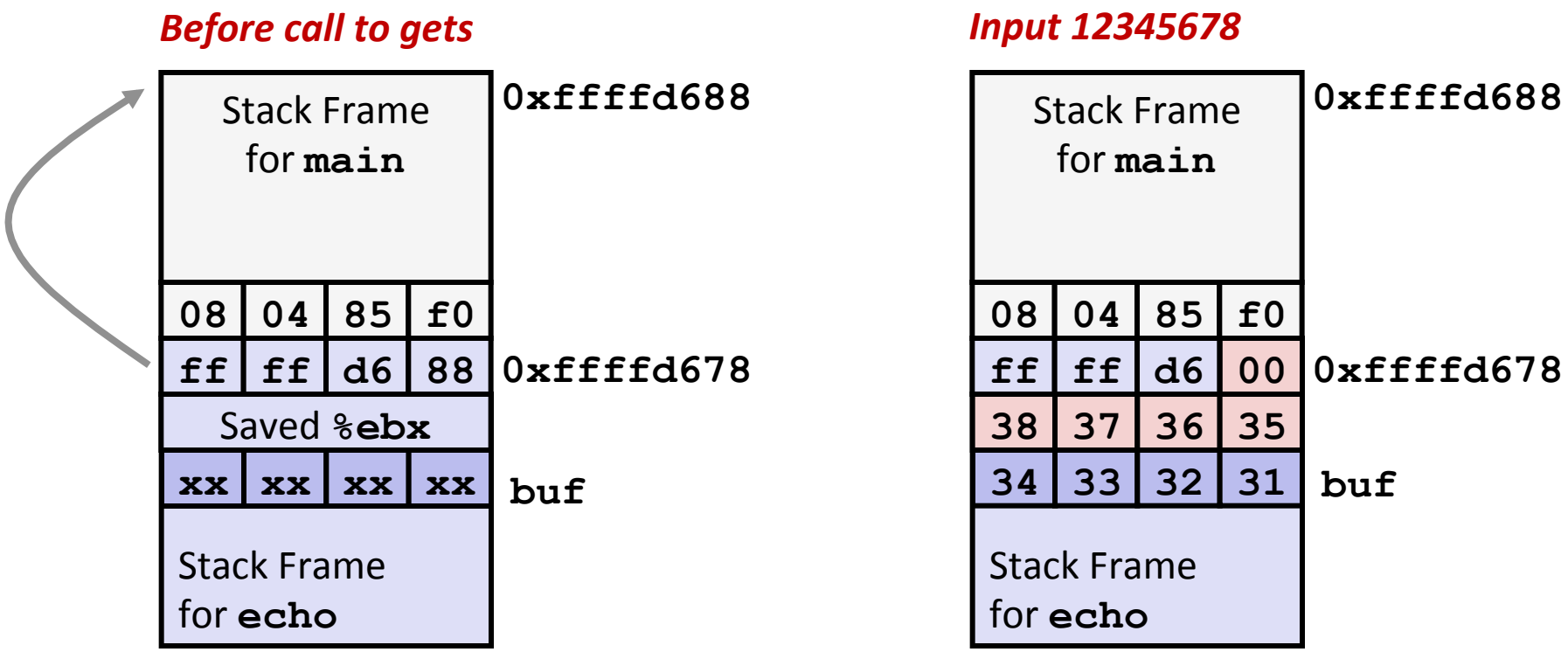
Buffer Overflow Example #1





Overflow buf, and corrupt %ebx, but no problem

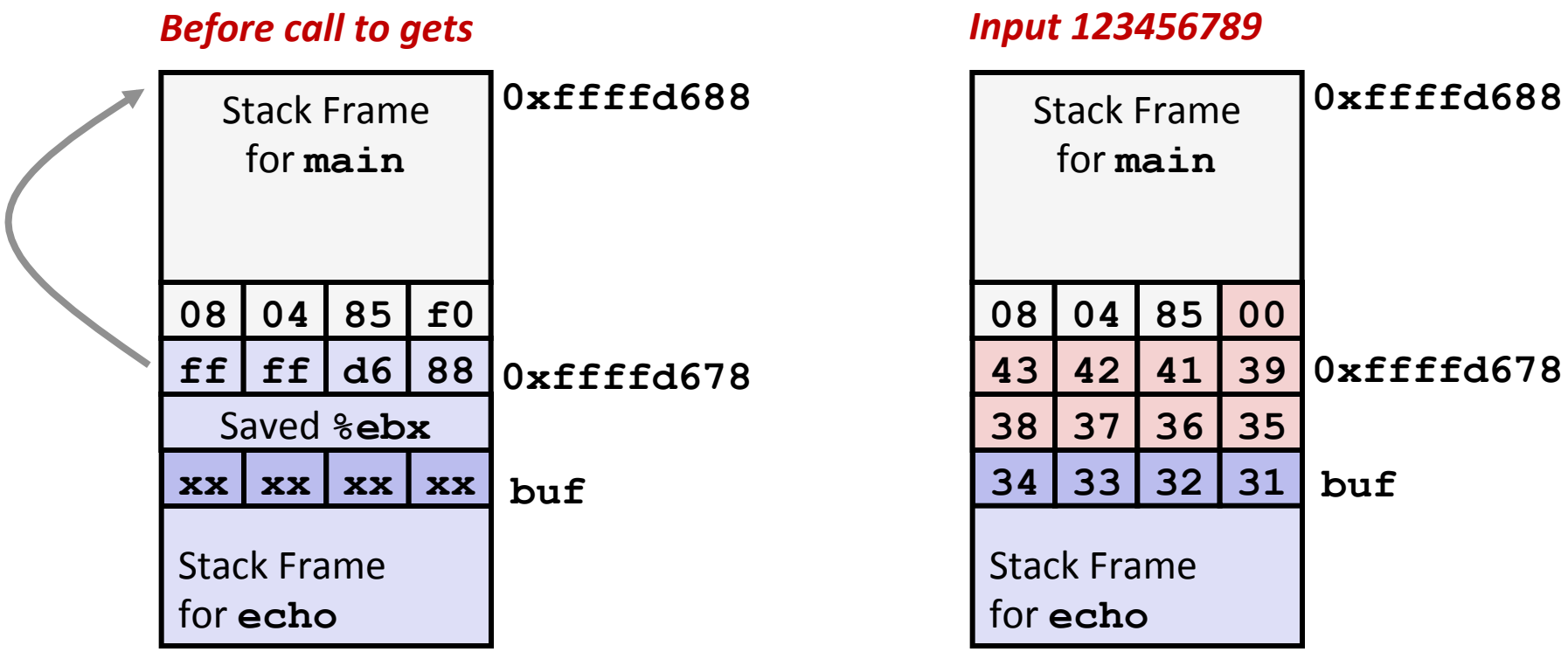
Buffer Overflow Example #2



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



Return address corrupted

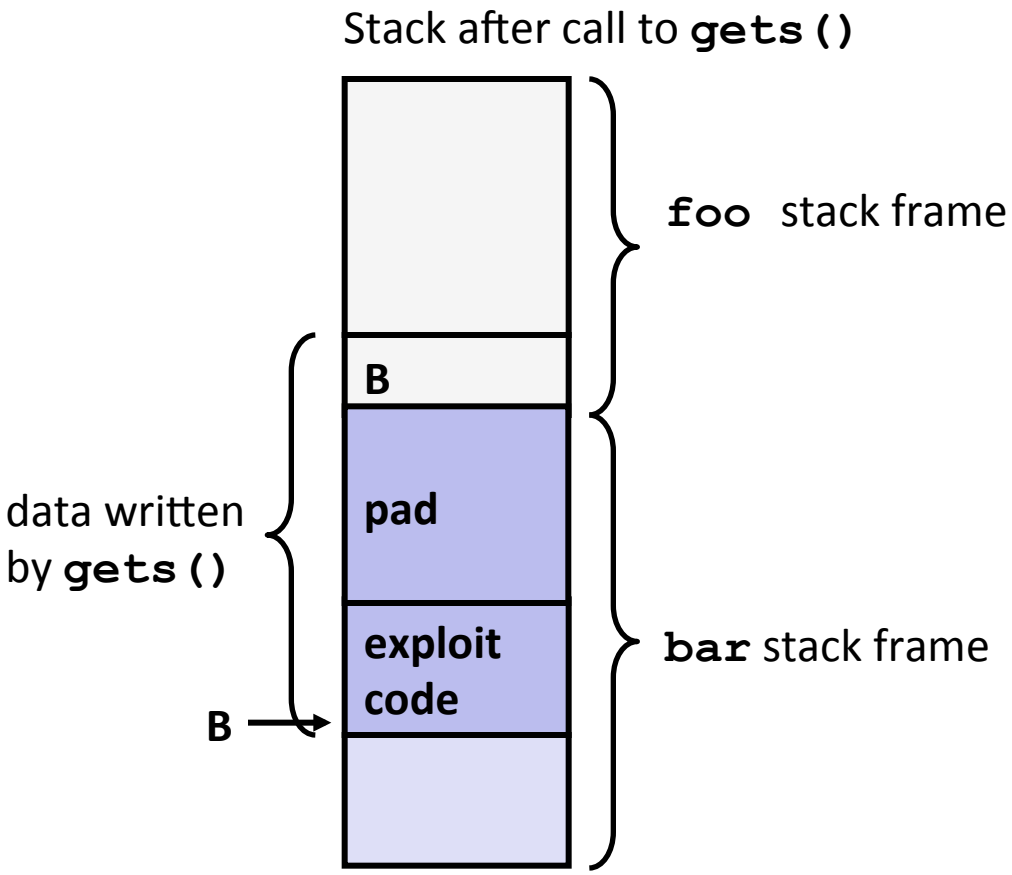
```
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave    # Desired return point
```

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.

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

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**

Today

- **Structures**
  - Alignment
- **Unions**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection