Machine-Level Programming IV: Compound Data Types, Cont'd

B&O Readings: 3.8-3.9

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Today: Compound Types in C

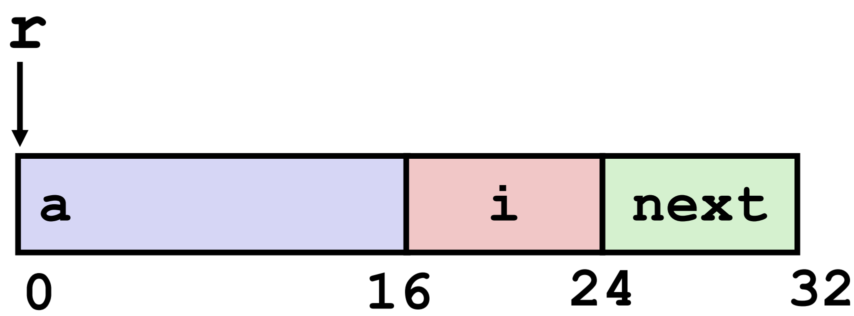
- **Arrays**
  - One-dimensional arrays
  - Multi-dimensional / nested arrays
  - Multi-level arrays
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Unions**

Struct in C

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
struct rec g;
struct rec *r = &g;
```

or

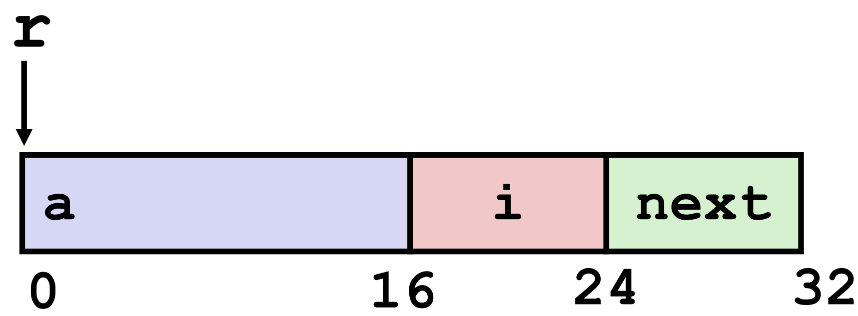
```c
typedef struct rec {
    int a[4];
    long i;
    struct rec *next;
} rec_t;
rec_t g;
rec_t *r = &g;
```



- **Concept**
  - Groups data of possibly different types into a single object
  - Refer to members within structure by names
    - r->a[2]
    - g.a[2]

Structure Representation

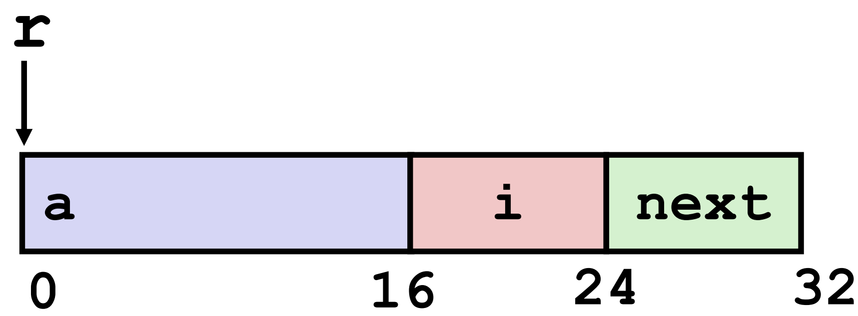
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} g;
struct rec *r = &g;
```



- **Structure represented as block of memory**
  - Big enough to hold all of the fields
- **Fields ordered according to declaration**
  - Even if another ordering could yield a more compact representation
- **Compiler determines overall size + positions of fields**
  - Machine-level program has no understanding of the structures in the source code

Structure Access

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```



- **Accessing Structure Member**
  - Pointer indicates first byte of structure
  - Access elements with offsets

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

```
# %rdi = r
# %esi = val
movslq %esi, %rsi       # sign extend val to (long)val
movq   %rsi, 16(%rdi)   # store (long)val in r->i
```

Example: Struct Access

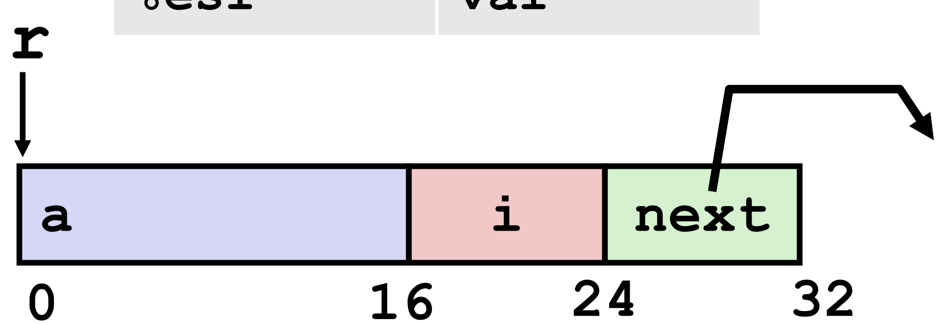
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```

```
jmp      .L11                 # goto test
.L12:                         # loop:
  movslq  16(%rdi), %rax      #   tmp = *(r+16)
  movl    %esi, (%rdi,%rax,4) #   *(r+4*tmp) = val
  movq    24(%rdi), %rdi      #   r = *(r+24)
.L11:                         # test:
  testq   %rdi, %rdi          #   Test r
  jne     .L12                #   if !=0 goto loop
```

- **C Code**

```
void set_val
  (struct rec *r, int val){
  while (...) {
    ...;
    ...;
    ...;
  }
}
```

Register	Value
%rdi	r
%esi	val



Example: Struct Access

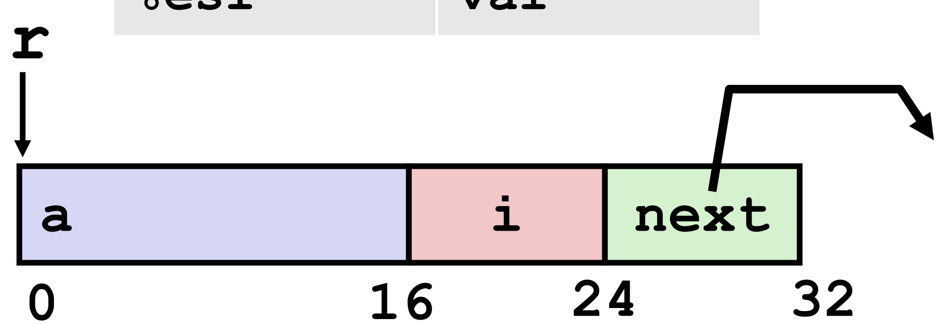
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

```
jmp     .L11                   # goto test
.L12:                          # loop:
  movq    16(%rdi), %rax       #   tmp = Mem[r+16]
  movl    %esi, (%rdi,%rax,4)  #   Mem[r+4*tmp] = val
  movq    24(%rdi), %rdi       #   r = Mem[r+24]
.L11:                          # test:
  testq   %rdi, %rdi           #   Test r
  jne     .L12                 #   if !=0 goto loop
```

- **C Code**

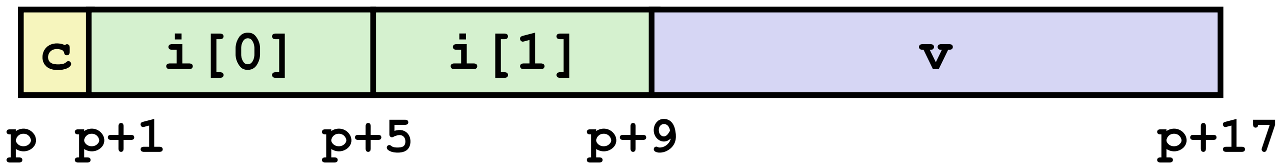
```
void set_val
  (struct rec *r, int val){
  while (r) {
    long tmp = r->i;
    r->a[tmp] = val;
    r = r->next;
  }
}
```

Register	Value
%rdi	r
%esi	val



Structures & Alignment

Unaligned Data




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

Alignment Principle

- Primitive data type requires K bytes: field must be ***K-byte aligned*** (i.e., its address is multiple of K).
- The struct itself must also be multiple of K, where K is the largest alignment requirement among its fields.

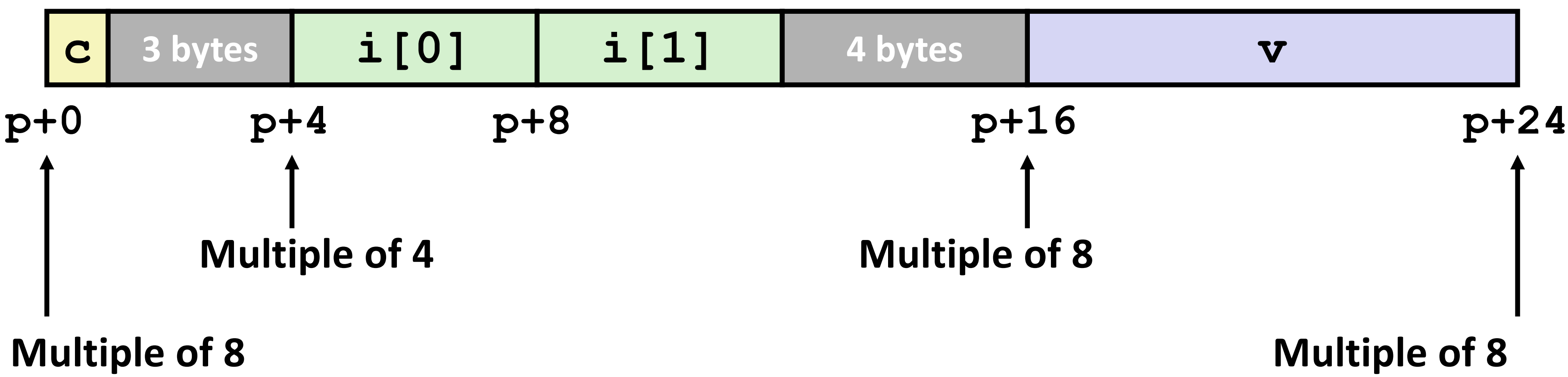
Specific Cases of Alignment (x86-64)

- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, `long`, `char *`, …**
  - lowest 3 bits of address must be 000_2
- **16 bytes: `long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

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
    (so an array of structs just works!)
- **Example:**
  - K = 8, due to **`double`** element

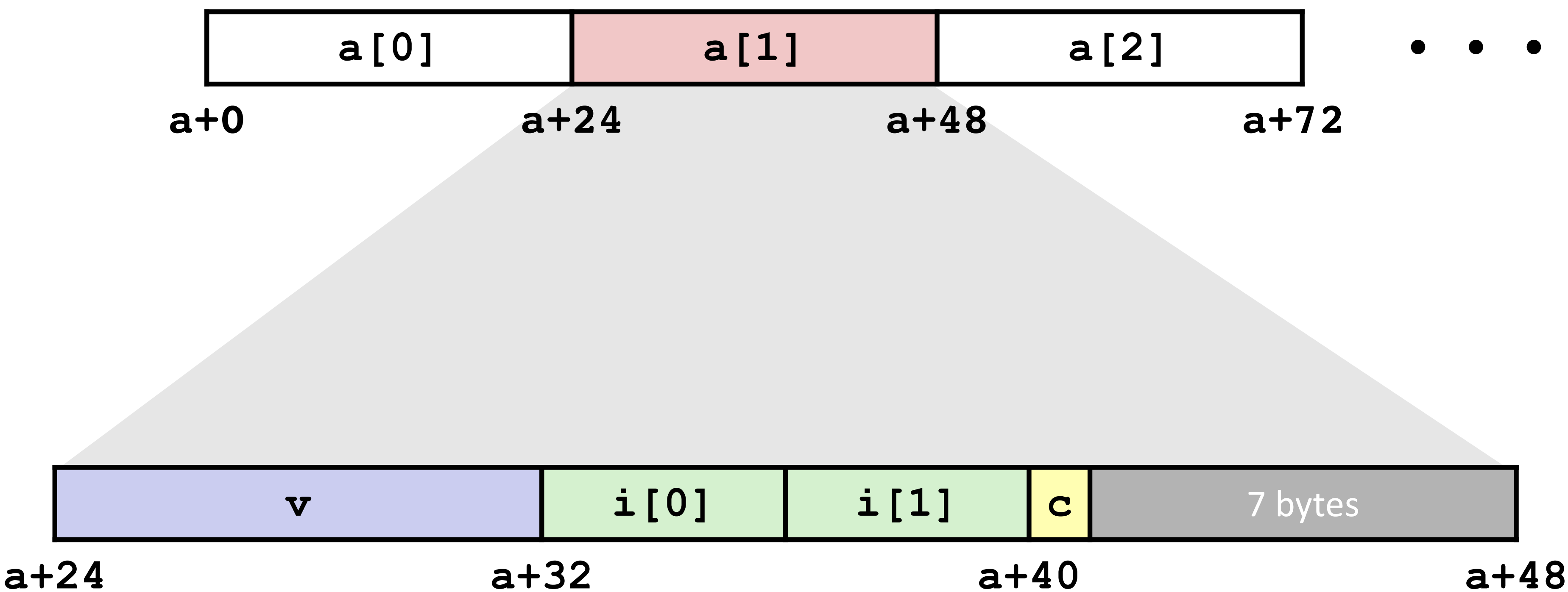
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Arrays of Structures

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

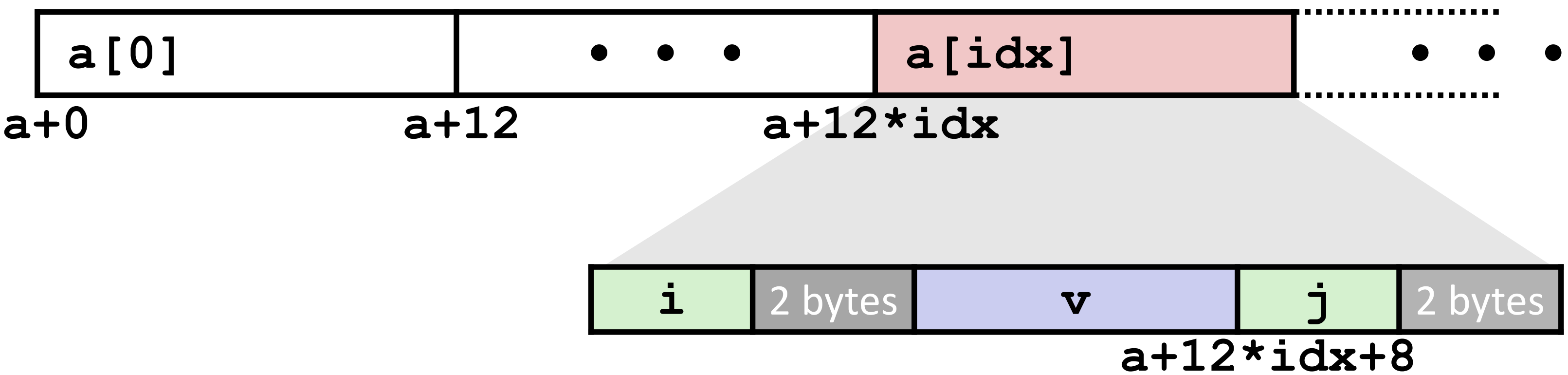
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
};
```

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



Q: What's the assembly? (Note that a is desclared as a global array)

```
Struct S3 a[10];
short get_j(int idx){
  return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax    #3*idx
movzwl a+8(,%rax,4),%eax   #a+8+12*idx
```

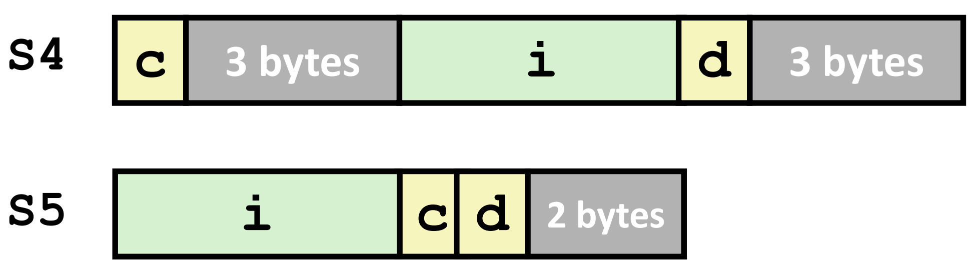
Saving Space

- **Put large data types first**

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

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

- **Effect: saving 4 bytes**



Alignment Principles

- **Aligned Data**
  - For primitive data type that requires K bytes, its address must be a multiple of K.
  - The whole struct must have size that is a multiple of K (where K is the largest alignment requirement among its fields).
  - Required on some machines; advised on x86-64
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

Meeting Overall Alignment Requirement

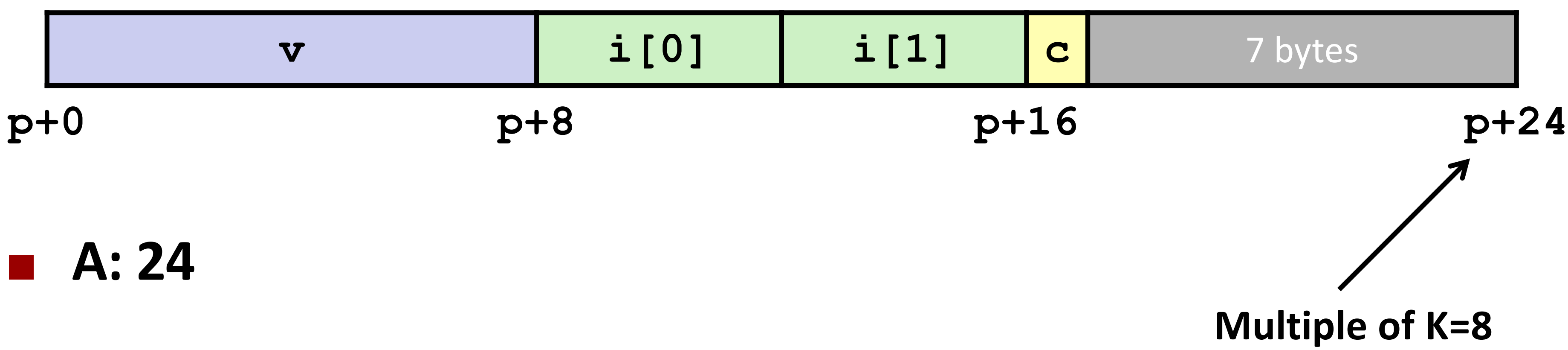
- **Q: What's the size of the following struct?**

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

Meeting Overall Alignment Requirement

- Q: What's the size of the following struct?

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



- A: 24

Recap: What We Learned Thus Far

- **The `struct` type in C:**
  - The size and alignment requirement are determined by the largest alignment requirement amongst its fields (determined by the compiler).

- **Once we get to the assembly-level, the assembly instruction knows nothing about the compound type.**
  - Access to a field in a struct is just like any memory reference.
  - The compiler generates the memory reference with the address of the struct + the right offset.

Alignment Calculation Example

- Q: What's the alignment requirement for `P3`?

```
struct P1 { int i; char c; char d; long j; }

struct P2 { short w[3]; char c[3]; };

struct P3 { struct P2 a[2]; struct P1 t };
```

- A: 16
- B: 8
- C: 24
- D: None of the above

Alignment Calculation Example

- Q: What's the alignment requirement for `P3`?

```
struct P1 { int i; char c; char d; long j; }
```

offset: 0 4 5 8

alignmt: 8
size: 16

```
struct P2 { short w[3]; char c[3]; };
```

offset: 0 6

alignmt: 2
size: 10

```
struct P3 { struct P2 a[2]; struct P1 t };
```

offset: 0 24

alignmt: 8
size: 40

- A: 16
- B: 8
- C: 24
- D: None of the above

Answer: B

Today: Compound Types in C

- **Arrays**
  - One-dimensional arrays
  - Multi-dimensional / nested arrays (next time)
  - Multi-level arrays (next time)
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Unions**

Union in C

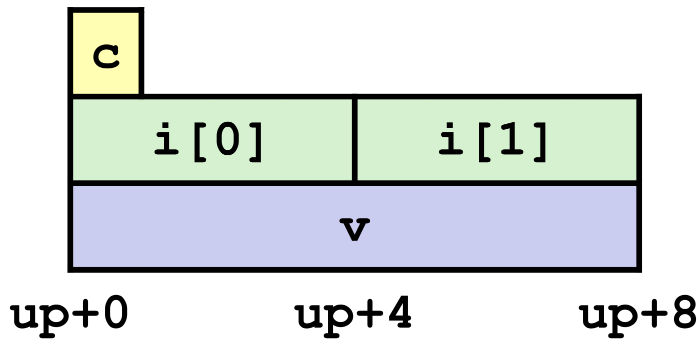
- Circumvent the type system of C
- Allowing a single object to be referenced according to multiple types
- Fields share the same memory location
- Refer to members within structure by names
  - up->i[2]
  - (*up).i[2]

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

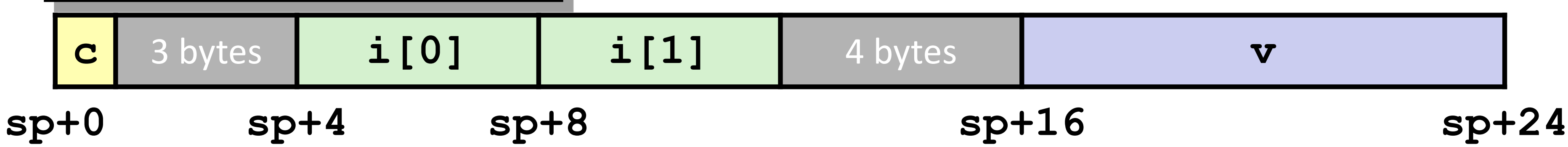
Union Allocation

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

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

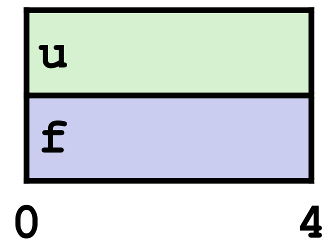



```
struct S1 {
  char c;
  int i[2];
  double v;
} s;
struct S1 *p = &s;
```



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;
}
```

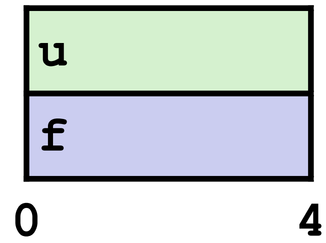
```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(float) u` ?

Same as `(unsigned) f` ?

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;
}
```

Same as `(float) u` ?

Same as `(unsigned) f` ?

Answr: NO! With type cast, you change the bit representations.

Byte Ordering Puzzle

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

64-bit

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

lower address … **higher address**

What's the size of this union? 8 bytes

Byte Ordering Puzzle (Cont.)

Little Endian

lower address

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

higher address

What are the output on x86-64?

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 ==
Ints       0-1 ==
Long       0   ==
```

Byte Ordering Puzzle (Cont.)

Little Endian

lower address

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

higher address

What are the 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]
```

Byte Ordering Revisited

- **Idea**
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- **Big Endian**
  - Most significant byte has the lowest address
  - Sparc
- **Little Endian**
  - Least significant byte has the lowest address
  - Intel x86, ARM Android and IOS

Recap: What We Learned Thus Far

- **The `union` type in C:**
  - All fields share the same storage (unlike **`struct`**)
    - Every field has offset 0; some are padded
    - Same bits just reinterpreted!
    - **Byte ordering depends on systems (Little vs Big Endian)**
  - The size and alignment for a union type are determined by the largest alignment requirement amongst its fields (determined by the compiler, just like **`struct`**).

- **Once we get to the assembly-level, the assembly instruction knows nothing about the compound type.**

Machine-Level Programming V: Advanced Topic

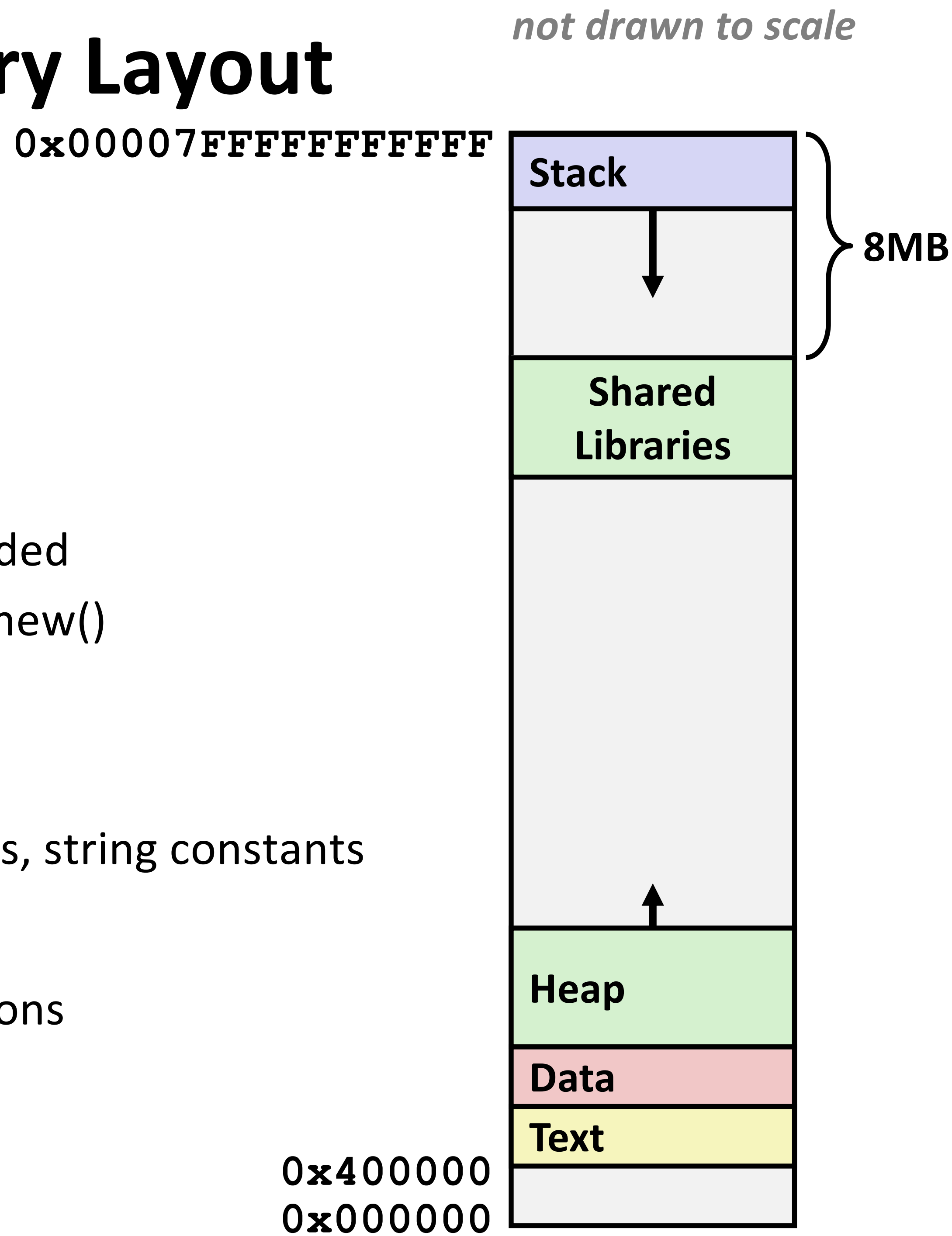
B&O Readings: 3.9-3.10

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only



Memory Allocation Example

not drawn to scale

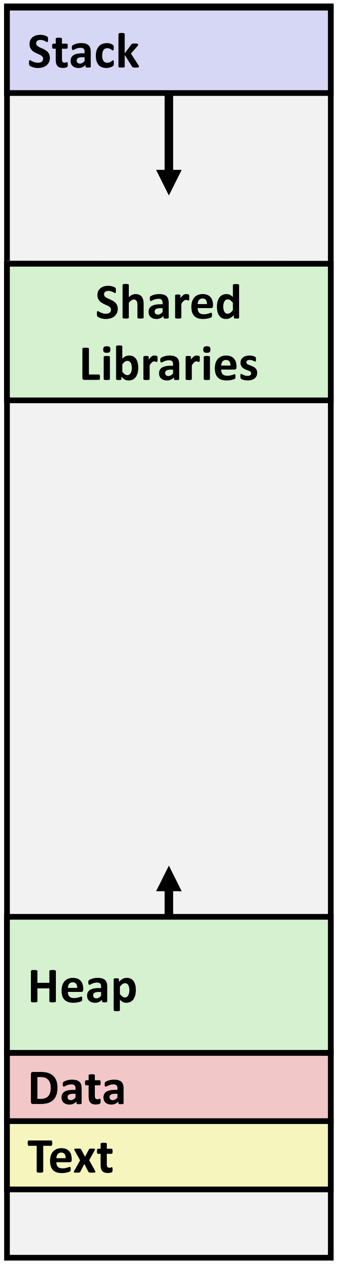
```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```

Where does everything go?

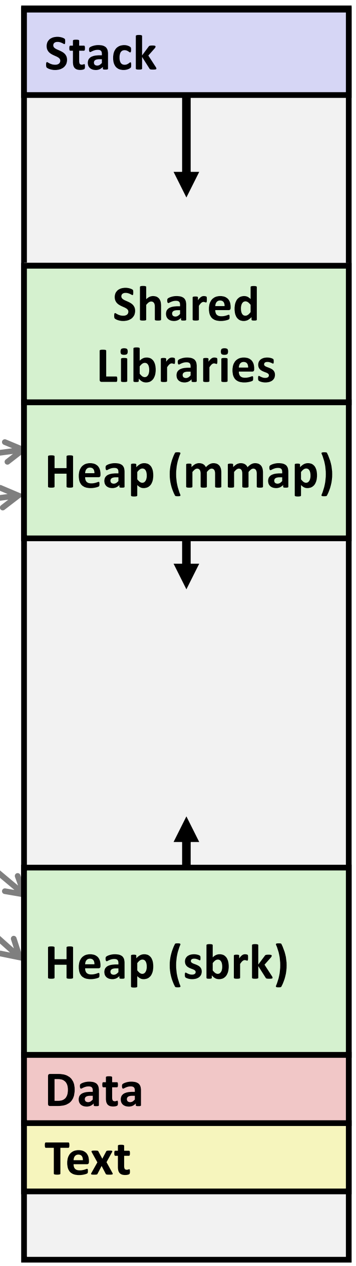


x86-64 Example Addresses

address range ~2^{47}

not drawn to scale

`&local`	`0x00007ffee4379f54`
`&p1`	`0x00007ffee4379f58`
`p1`	`0x00007fffe7a19010`
`p3`	`0x00007ffee7a18010`
`p4`	`0x0000000081602120`
`p2`	`0x0000000081602010`
`big_array`	`0x0000000080601060`
`huge_array`	`0x0000000000601060`
`main()`	`0x00000000004005c6`
`useless()`	`0x00000000004005c0`



Today

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection

Recall: Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  long l;
} struct_t;

long fun(int i) {
  volatile struct_t s;
  s.l = 999;
  s.a[i] = 1000; /* Possibly out of bounds */
  return s.l;
}
```

`fun(0)`	→	`999`
`fun(1)`	→	`999`
`fun(2)`	→	`1000`
`fun(3)`	→	`4294967296999`
`fun(4)`	→	**Segmentation fault**

- Result is system specific

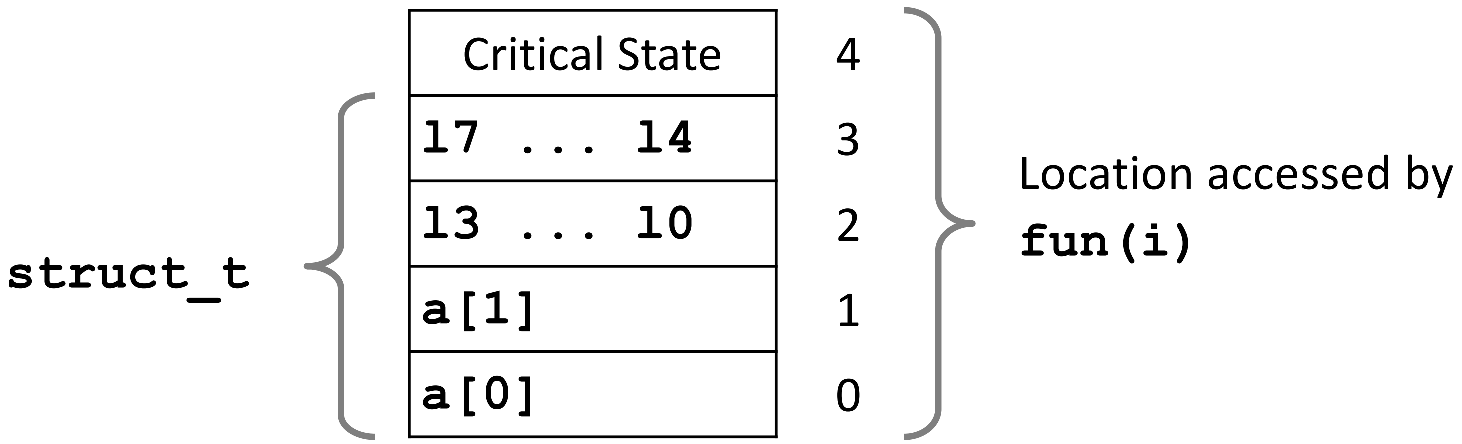
Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  long l;
} struct_t;
```

```
fun(0)  →            999
fun(1)  →            999
fun(2)  →           1000
fun(3)  →  4294967296999
```

`fun(4)` → **Segmentation fault**

Explanation:



Result is system specific!

Such problems are a BIG deal

- **Generally called a “buffer overflow”**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It’s the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing