Machine-Level Programming IV: Data

Slides adapted from the CMU version of the course (thanks to Randal E. Bryant and David R. O'Hallaron)

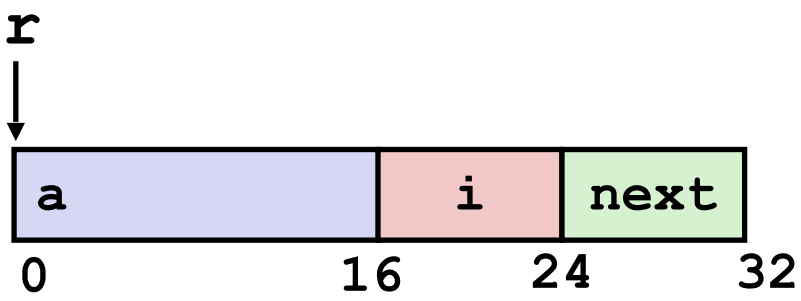
Today

- **Structures**
  - Allocation
  - Access
  - Alignment

Structure Representation

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

Composite data type (multiple members)



- Structure represented as block of memory
  - Big enough to hold all of the fields (members)
- 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

Side Note (&r->a[idx])

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

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

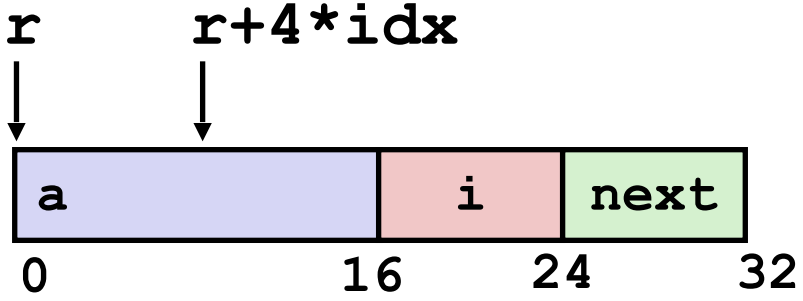
- **Use of -> in structures**
  - Shortcut: Instead of writing (*r).a[idx] we can write r->a[idx]
- **What does &r->a[idx] mean?**
  - &(r->a[idx]) or (&r)->a[idx] ?
- **Answer**
  - It means &(r->a[idx]) because -> has a high precedence

Generating Pointer to Structure Member

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

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

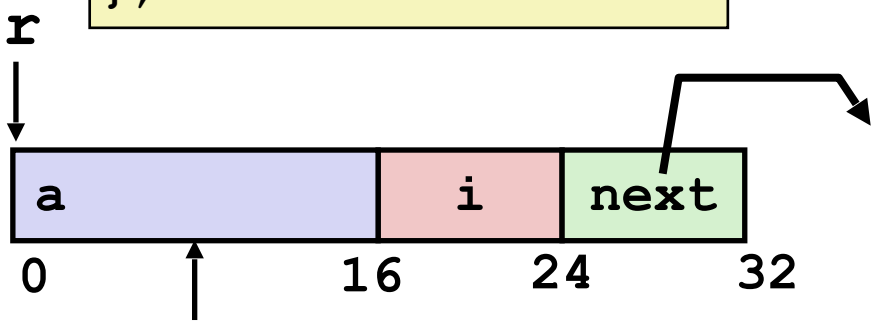
- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time
  - Compute as **`r + 4*idx`**




```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Following Linked List

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




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

Register	Value
%rdi	r
%rsi	val

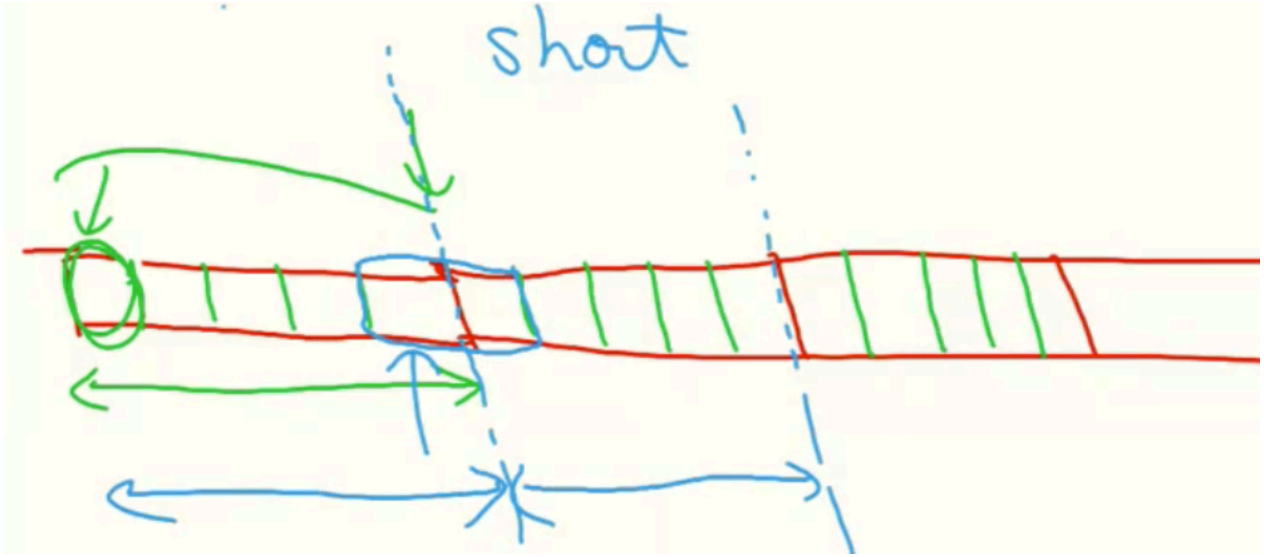
```
.L11:                                  # loop:
  movq    16(%rdi), %rax               #   c = *(r+16)
  movl    %esi, (%rdi,%rax,4)          #   *(r+4*c) = val
  movq    24(%rdi), %rdi               #   r = *(r+24)
  testq   %rdi, %rdi                   #   Test r
  jne     .L11                 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

Today

- **Structures**
  - Allocation
  - Access
  - Alignment

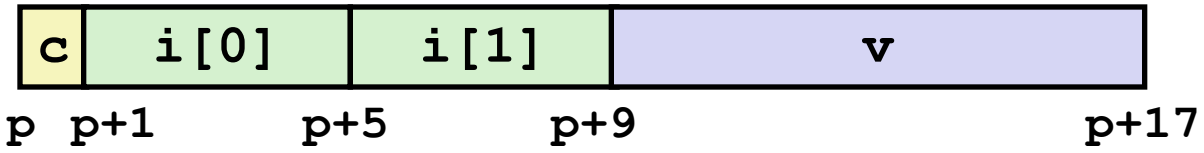
Structures & Alignment

- Why do we Need Alignment?



Structures & Alignment

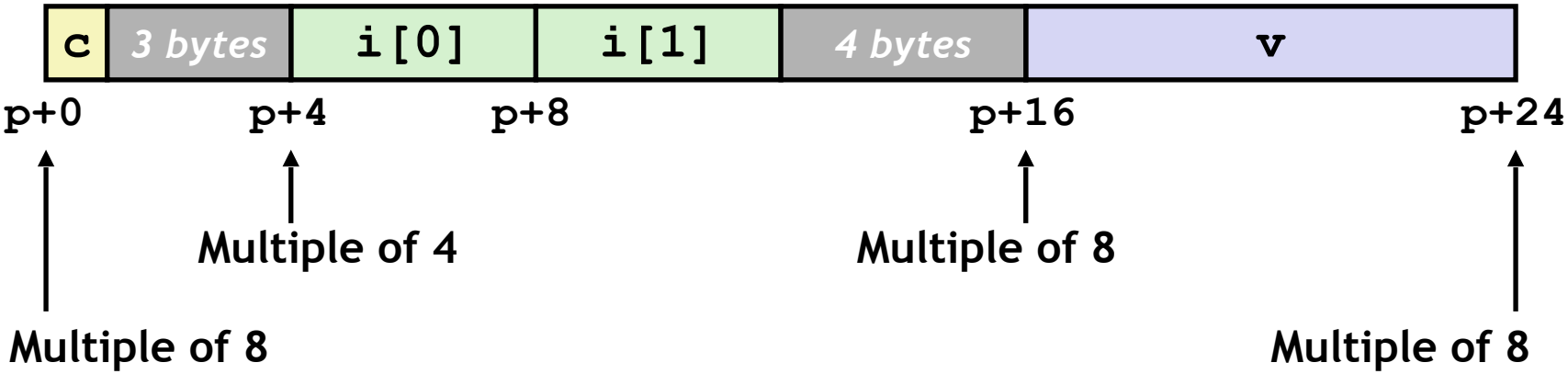
Unaligned Data




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

Aligned Data

- If a primitive data type requires ***K*** bytes
- Then, the address must be multiple of ***K***



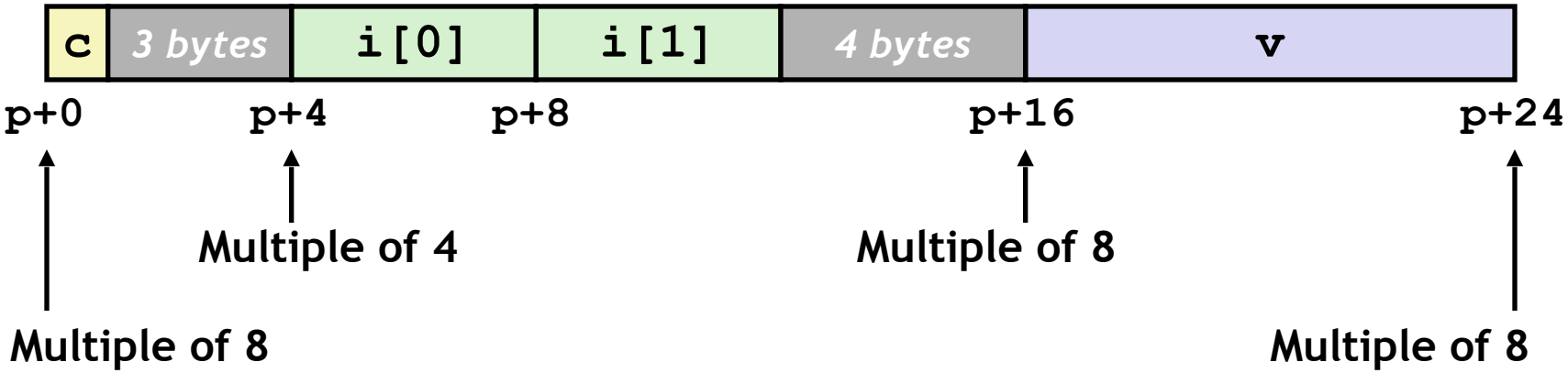
Alignment Principles

- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***
  - Required on some machines; advised on 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
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

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

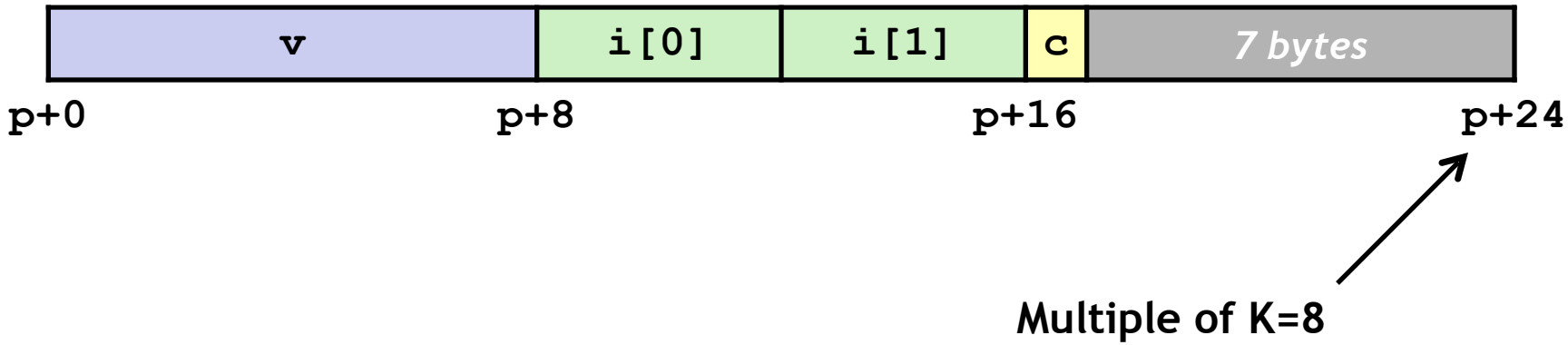
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Example 2 (Reorder Struct Fields)

- Assuming the order is different
- Now padding bytes are added to the end
- Why?
  - The largest alignment requirement is K
  - Overall structure size must be multiple of K

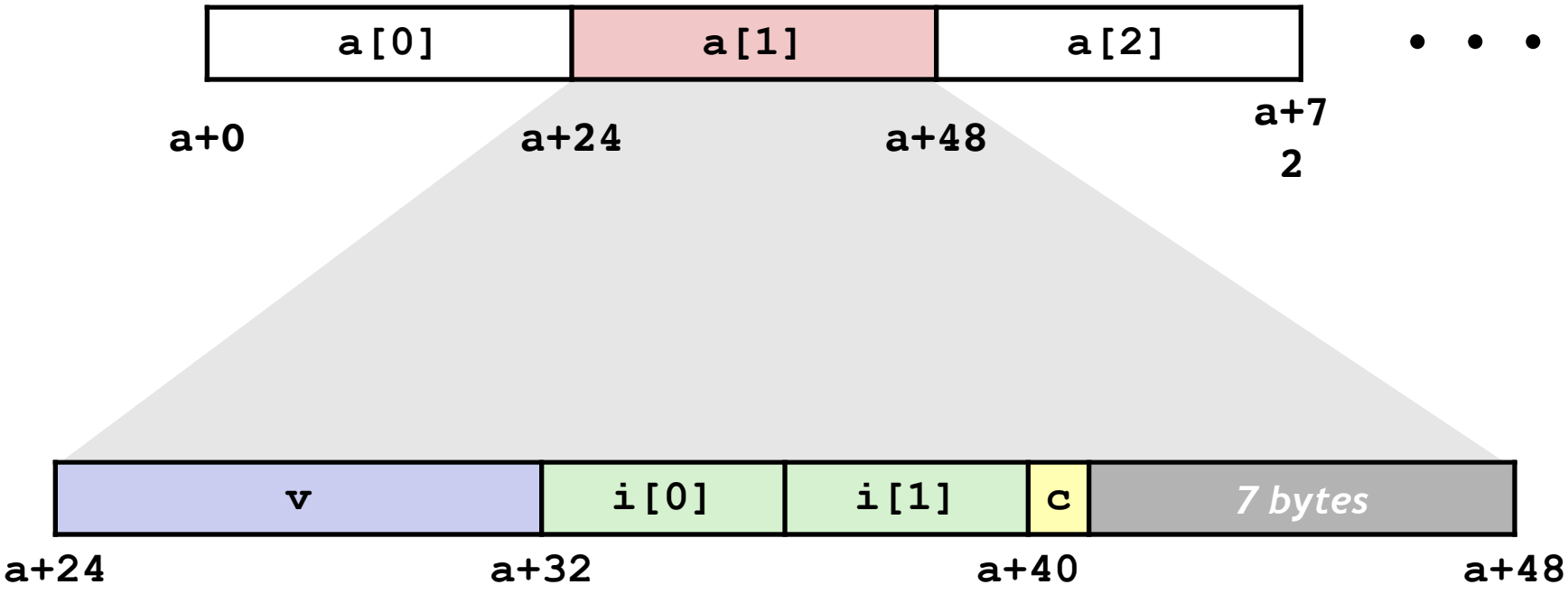
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

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

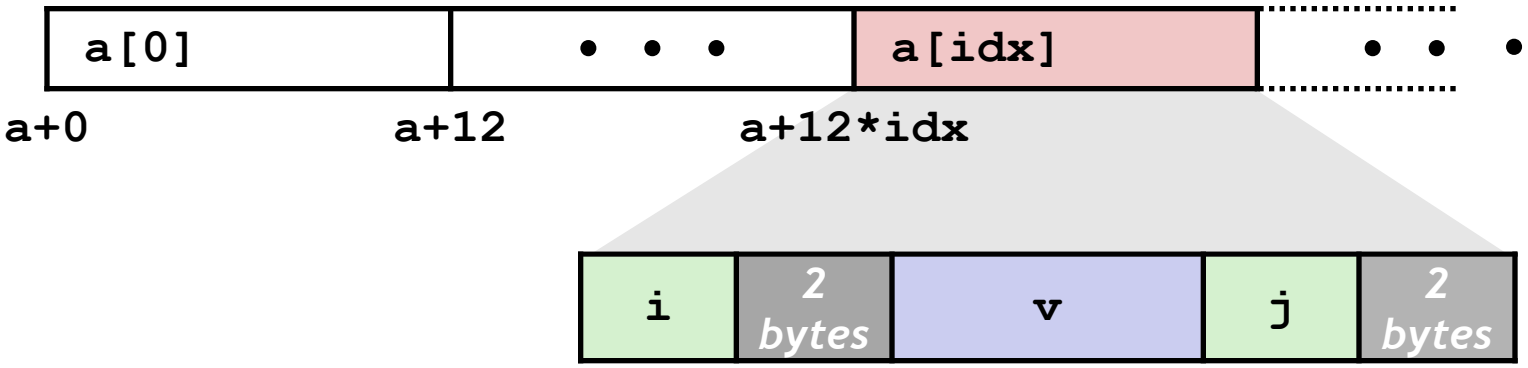
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

- Compute array offset 12*idx
- Element `j` is at offset 8 within structure
- Address: a+12*idx+8



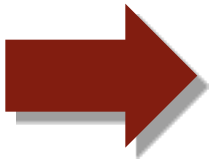

```
short get_j(int idx)
{
  return a[idx].j;
}
```

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

Saving Space

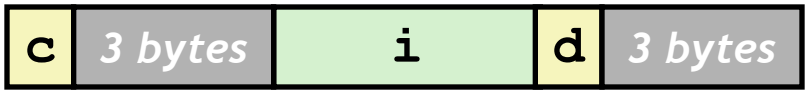
- Put large data types first

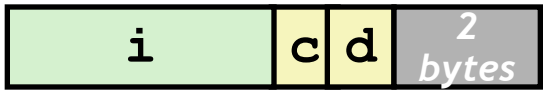
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```



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

- Effect (K=4)





Practice

- Reverse engineer the following code to C

```
struct S4 {
  short v;
  Struct S4 *p;
};
short f(struct S4 *ptr);
```

```
; short f(struct S4 *ptr)
; ptr in %rdi

  movl $1, %eax
  jmp .L2
.L3:
  imulq (%rdi), %rax
  movq 2(%rdi), %rdi
.L2:
  testq %rdi, %rdi
  jne .L3
  rep; ret
```

Practice

- Reverse engineer the following code to C

```
struct S4 {
  short v;
  Struct S4 *p;
};
short f(struct S4 *ptr);
```

```
short f(struct S4 *ptr)
{
  short val = 1;
  while (ptr)
  {
    val *= ptr->v;
    ptr = ptr->p;
  }
return val;
}
```

```
; short f(struct S4 *ptr)
; ptr in %rdi

  movl $1, %eax
  jmp .L2
.L3:
  imulq (%rdi), %rax
  movq 2(%rdi), %rdi
.L2:
  testq %rdi, %rdi
  jne .L3
  rep; ret
```