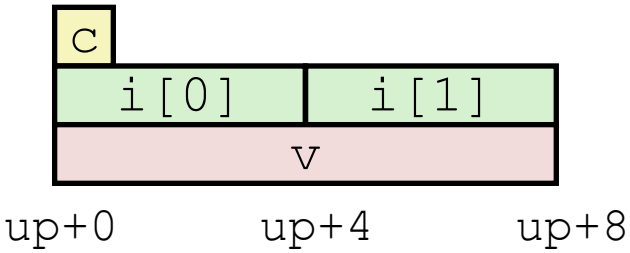
- Reading Responses
  - 6 Yes, 1 No
  - More often, review questions?
- Current Lab Format
  - Yes: 6 yes, 3 No
  - Maybe a 5-min introduction to help get started?
- More in-class activities / questions
  - Yes: 5, No: 1
  - Long Slides are boring / hard to follow: 3

- BTW, no luck with Flash lab on linux

Union Allocation

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

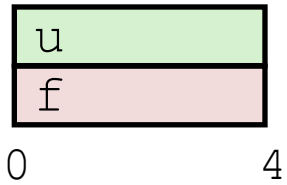
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```




```
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;
}
```

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

Same as `(float) u` ?

Same as `(unsigned) f` ?

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

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

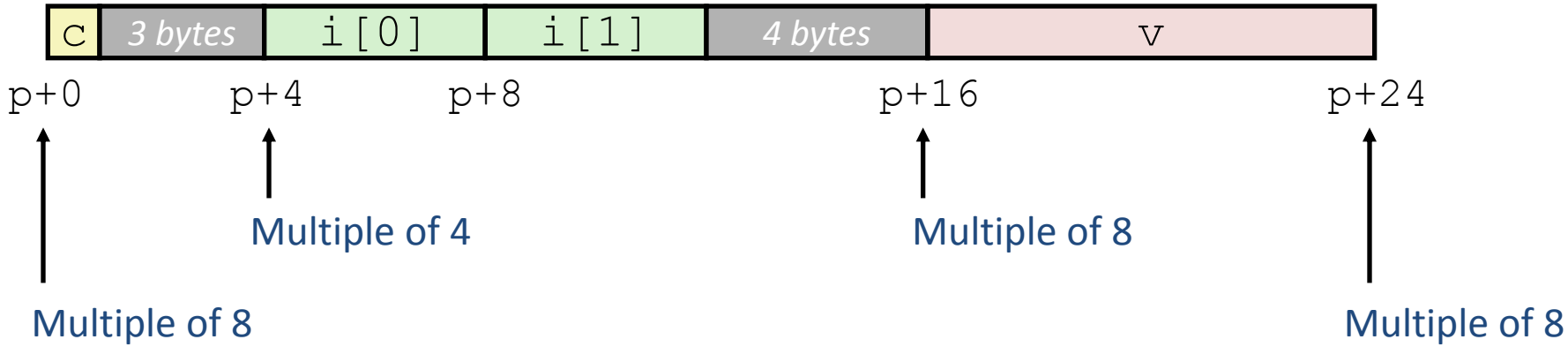
Specific Cases of Alignment (IA32)

- 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

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy 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

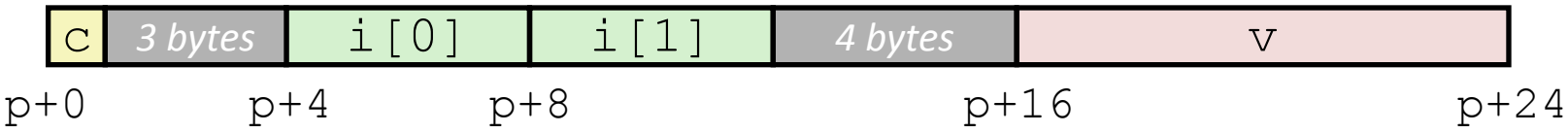
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

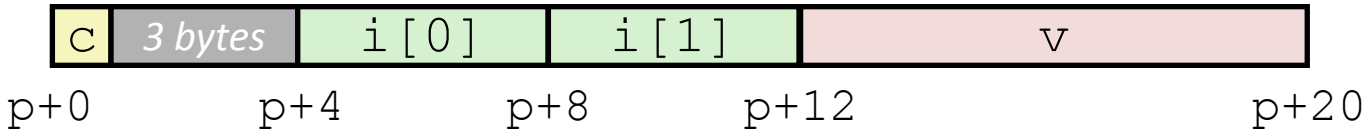


Different Alignment Conventions

```
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

Saving Space

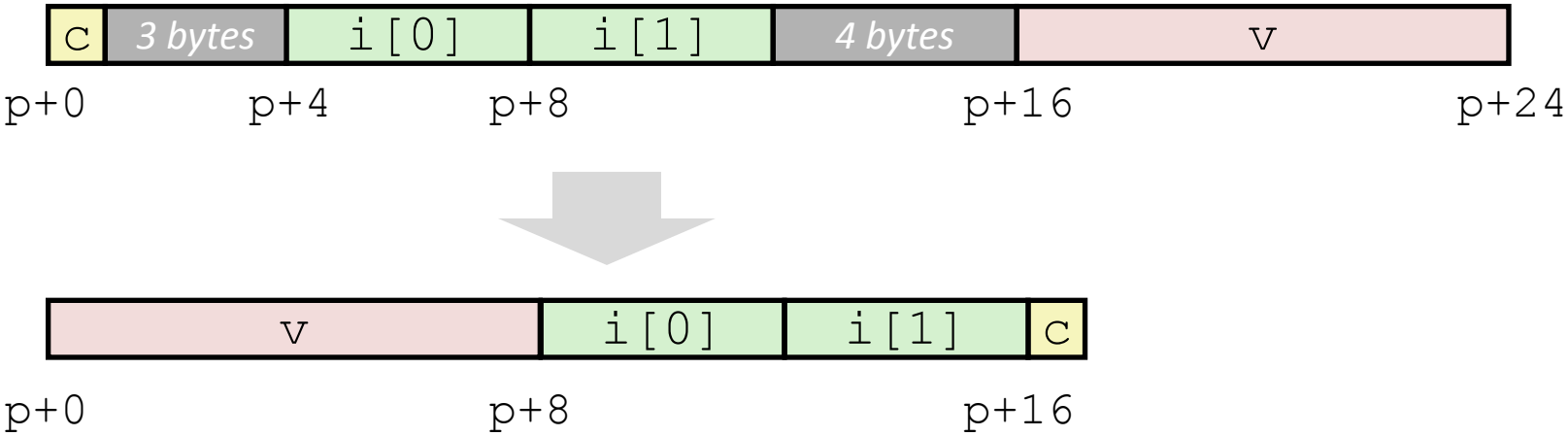
- Put large data types first

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



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

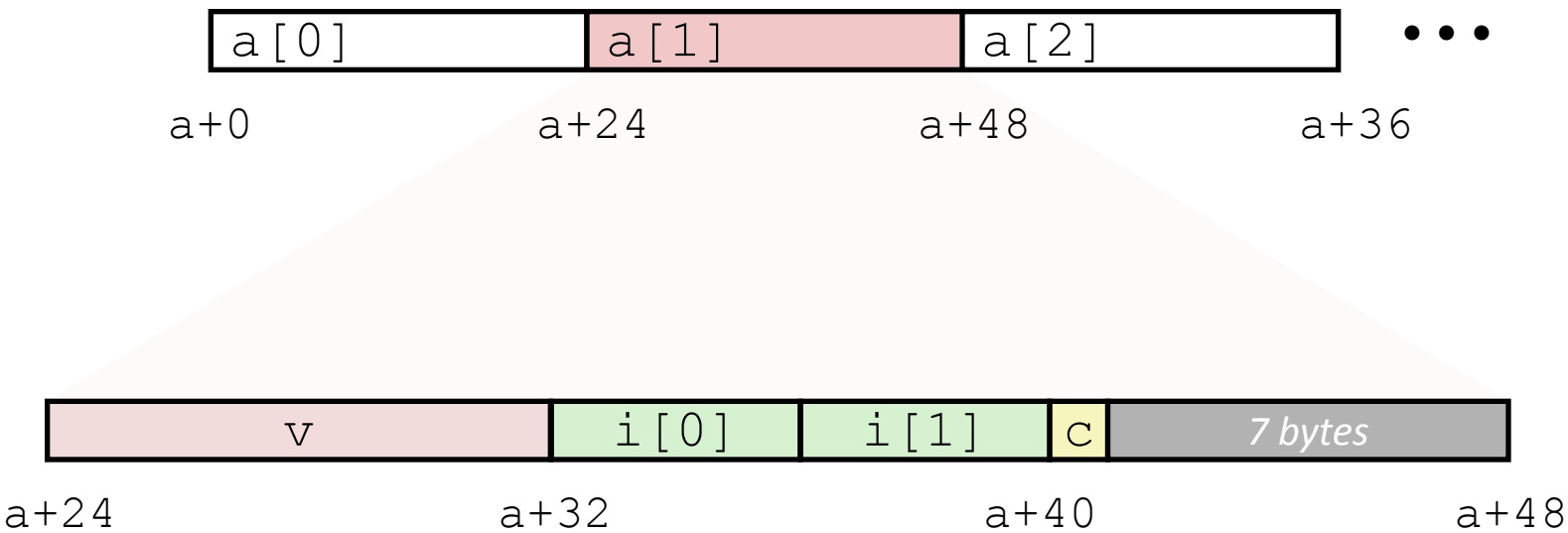
- Effect (example x86-64, both have *K=8)*



Arrays of Structures

- Satisfy alignment requirement for every element

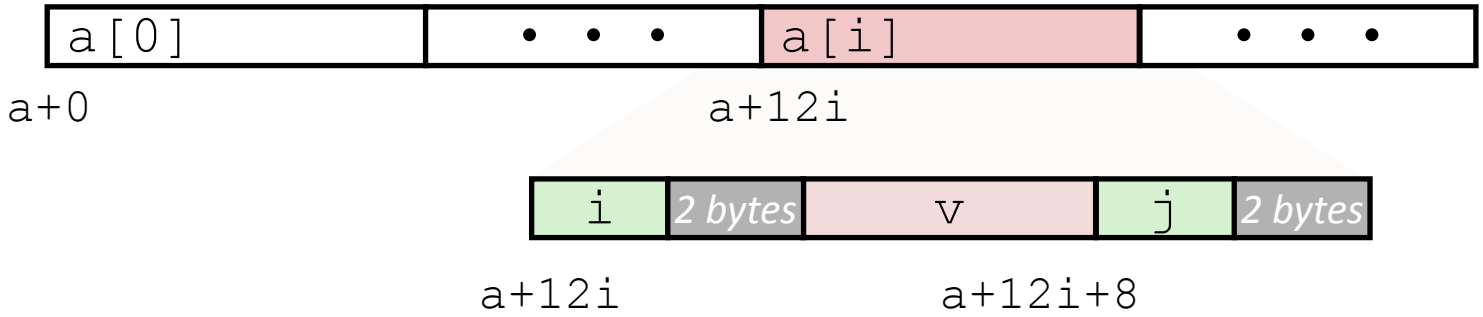
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

- Compute array offset 12i
- Compute offset 8 with 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
```

Offset per field, total size, and alignment requirement for IA32

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

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

```
struct {
    short w[3];
    char *c[3];
}  P3;
```

Offset per field, total size, and alignment requirement for IA32

- P1
  - 0 4 8 12
  - Total = 16
  - Alignment = 4
- P2
  - 0 6
  - Total = 10
  - Alignment = 2
- P3
  - 0 8
  - Total = 20
  - Alignment = 4