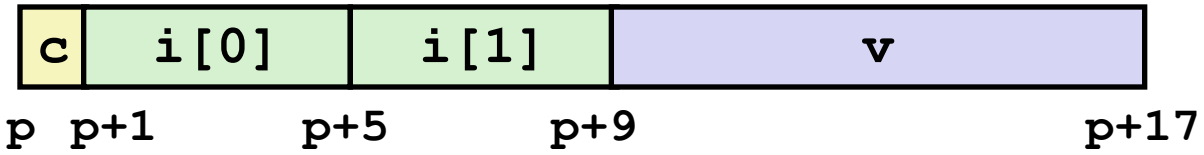
Section 5: Arrays & Other Data Structures

- Array allocation and access in memory
- Multi-dimensional or nested arrays
- Multi-level arrays
- Other structures in memory
- Data structures and alignment

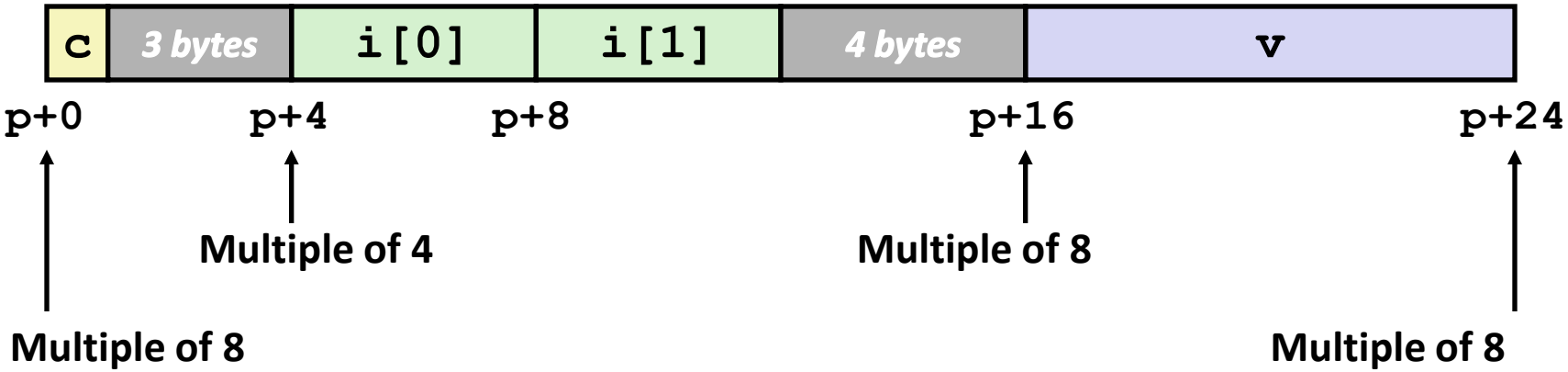
Structures & Alignment

- **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
- **Aligned data is required on some machines; it is *advised* on IA32**
  - Treated differently by IA32 Linux, x86-64 Linux, and Windows!
- **What is the motivation for alignment?**

Alignment Principles

- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
- **Aligned data is required on some machines; it is *advised* on IA32**
  - Treated differently by IA32 Linux, x86-64 Linux, and Windows!
- **Motivation for Aligning Data**
  - Physical memory is accessed by aligned chunks of 4 or 8 bytes (system-dependent)
    - Inefficient to load or store datum that spans quad word boundaries
  - Also, virtual memory is very tricky when datum spans two pages (later…)
- **Compiler**
  - Inserts padding in structure to ensure correct alignment of fields
  - **`sizeof()`** should be used to get true size of structs

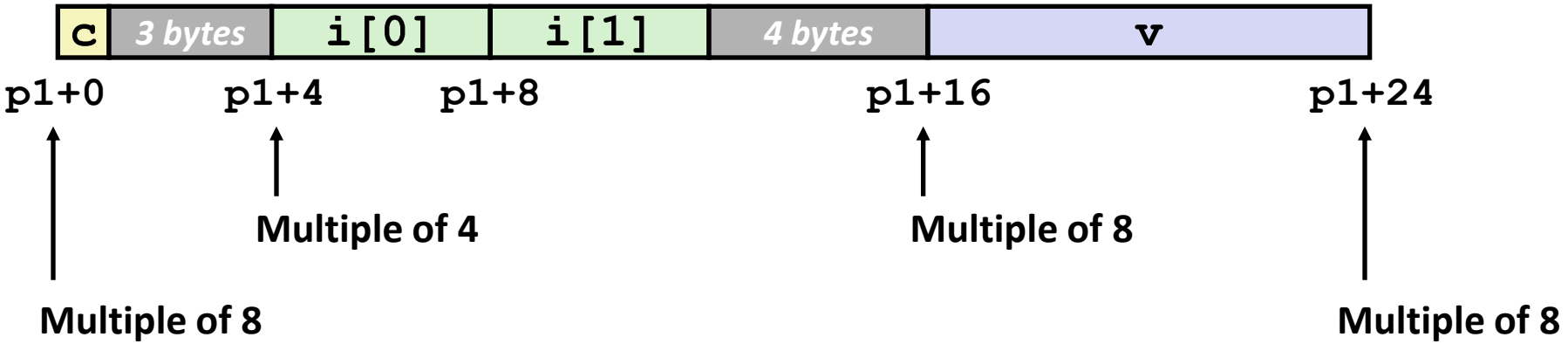
Specific Cases of Alignment (IA32)

- **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 OSs & instruction sets): lowest 3 bits 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

Satisfying Alignment with Structures

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

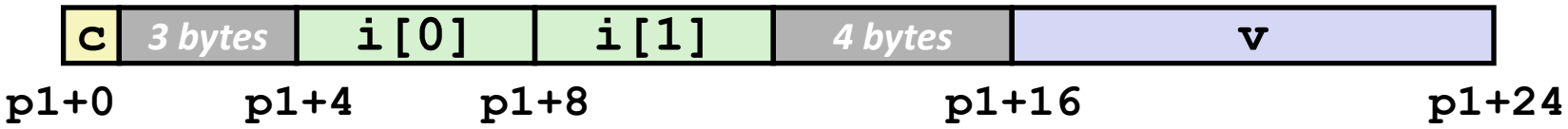
- **Within structure:**
  - Must satisfy every member's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K
- **Example (under Windows or x86-64):** K = ?
  - K = 8, due to **`double`** member



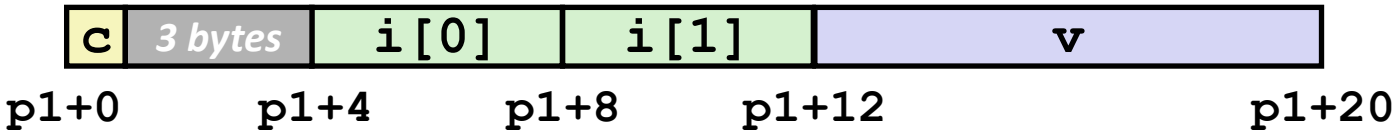
Different Alignment Conventions

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

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



- **IA32 Linux:** K = ?
  - K = 4; **`double`** aligned like a 4-byte data type



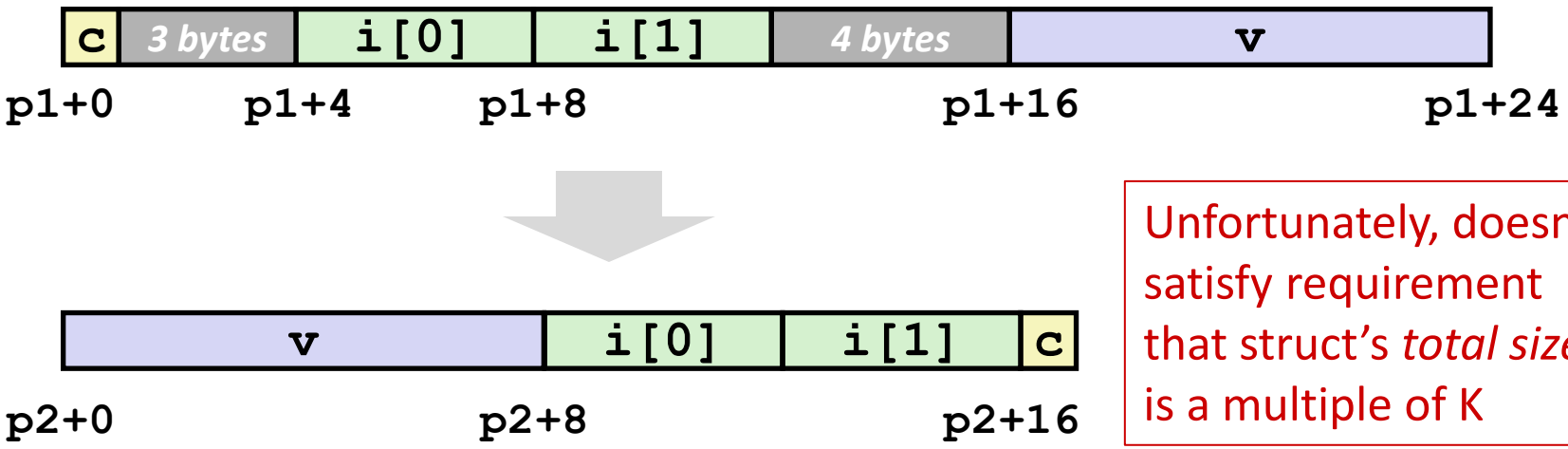
Saving Space

- Put large data types first:

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p2;
```

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

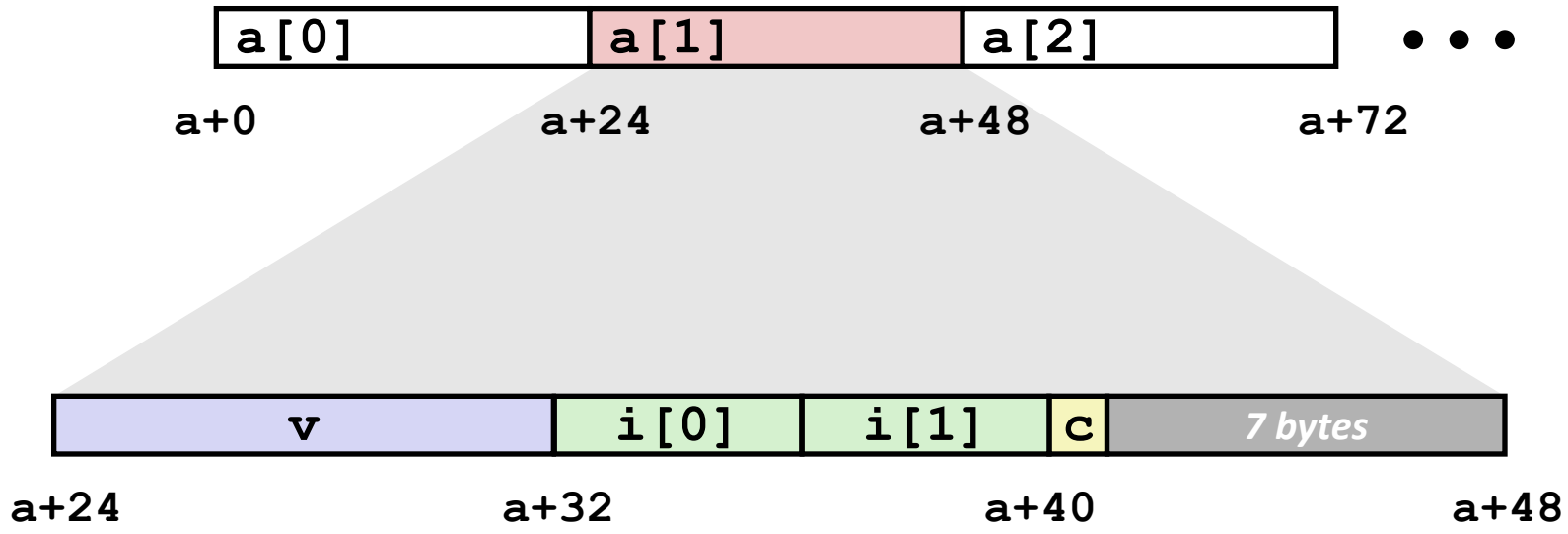


Unfortunately, doesn't satisfy requirement that struct's *total size* is a multiple of K

Arrays of Structures

- Satisfy alignment requirement for every element

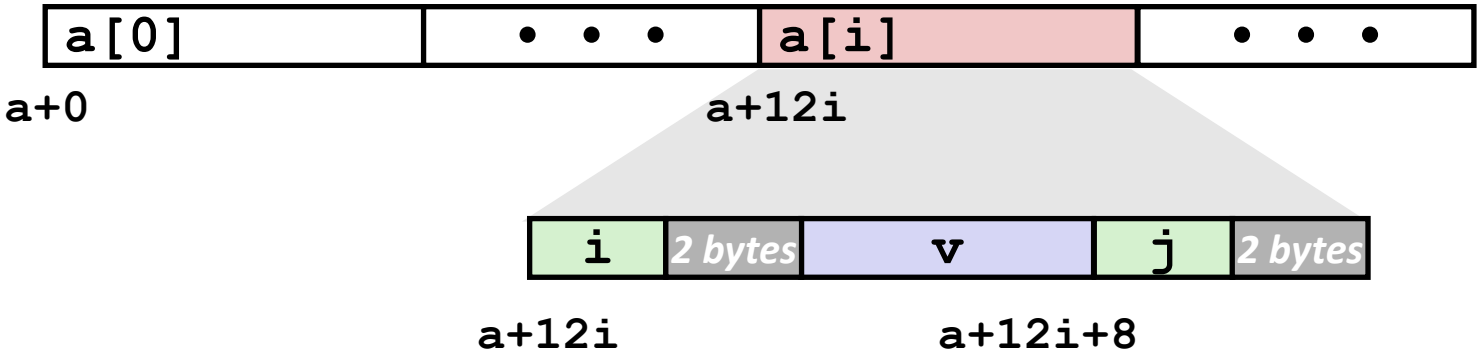
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

- Compute array offset 12i (`sizeof(S3)`)
- Element `j` is at offset 8 within structure
- Since a is static array, assembler gives offset a+8




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

```
# %eax = idx
leal (%eax,%eax,2),%eax  # 3*idx
movswl a+8(,%eax,4),%eax # a+12*idx+8
```