Announcements

- Assignment 5
  - Due November 4
  - String operations in IA-32
- CPSC 316: Foundation of Programming Languages
  - Study of PLs: design, evaluation, and implementation
  - Compiler construction
- CPSC 375: High-Performance Computing
  - Parallel algorithms
  - Concurrent programming
  - Distributed computing
  - GPU programming

Lecture 22

Data Alignment

CPSC 275

Introduction to Computer Systems

What's the Size?

- Consider

```
struct S {
    char c;
    int  i;
    short s;
};
```

Q: What is `sizeof(S)` under IA-32?

A: 12 (??)

What Is Data Alignment?

- Storing variables at memory addresses that are multiples of their sizes.
- Example:
  - int (4 bytes) stored at address 0x1000 (multiple of 4).
  - short (2 bytes) stored at address 0x1002 (multiple of 2).
- Unaligned access: occurs when data crosses a boundary not matching its size (e.g., `int` at 0x1003).

Memory and Data in IA-32

- IA-32 operates with a 32-bit address bus, providing a linear, 4-gigabyte (4GB) address space
- The smallest unit of memory that can be individually addressed is a single byte.
- The architecture handles data in various sizes (1, 2, **4**, or 8).
  - 4 bytes on IA-32
- Data is fetched from memory in fixed-size "chunks“ over bus (32 bits).
  - Aligned access → 1 memory operation
  - Misaligned access → 2 or more memory operations
  - On older CPUs: could even trigger a bus error or fault.

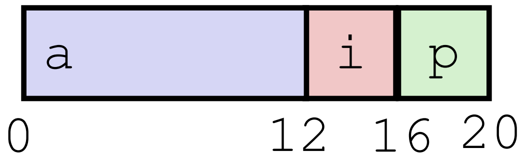
Why Data Alignment?

- Best performance when data sizes align perfectly with the boundaries of the memory bus
  - Allows single, efficient transactions.
- Unaligned memory access allowed, but suboptimal performance
  - CPU must perform extra work to retrieve data that spans across its natural memory access boundaries
- Accessing data that crosses *cache lines* results in *cache misses* and multiple memory cycles (TBD).

Structure Allocation

```
struct rec {
  int a[3];
  int i;
  char *p;
};
```

Memory Layout

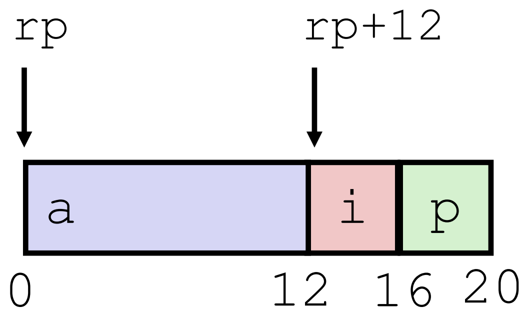


- Concept
  - Contiguously-allocated region of memory
  - Members may be of different types
  - Refer to members within structure by names
  - Examples:

```
struct rec r;        struct rec *rp = &r;
r.i = 10;            rp->i = 10;
```

Structure Access

```
struct rec {
  int a[3];
  int i;
  char *p;
};
```



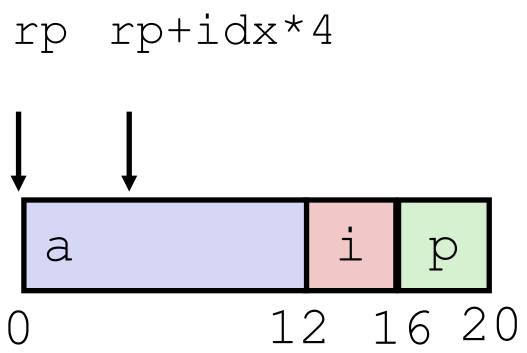
- Accessing Structure Member
  - Pointer indicates first byte of structure
  - Access elements with offsets

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

```
                      # %edx = val, %eax = rp
movl %edx, 12(%eax)   # Mem[rp+12] = val
```

Pointer to Structure Member

```
struct rec {
  int a[3];
  int i;
  char *p;
};
```

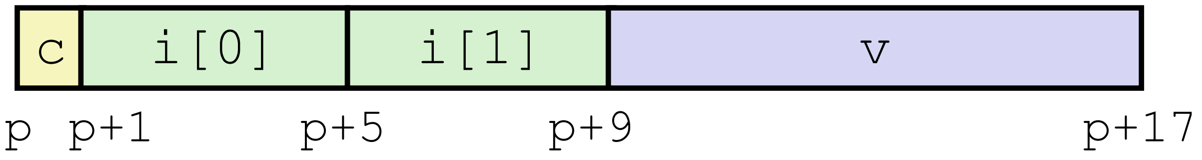



```
int *get_ap(struct rec *rp, int idx) {
  return &(rp->a[idx]);
}
```

```
movl    12(%ebp),%eax    # get idx
sall    $2,%eax          # idx*4
addl    8(%ebp),%eax     # r+idx*4
```

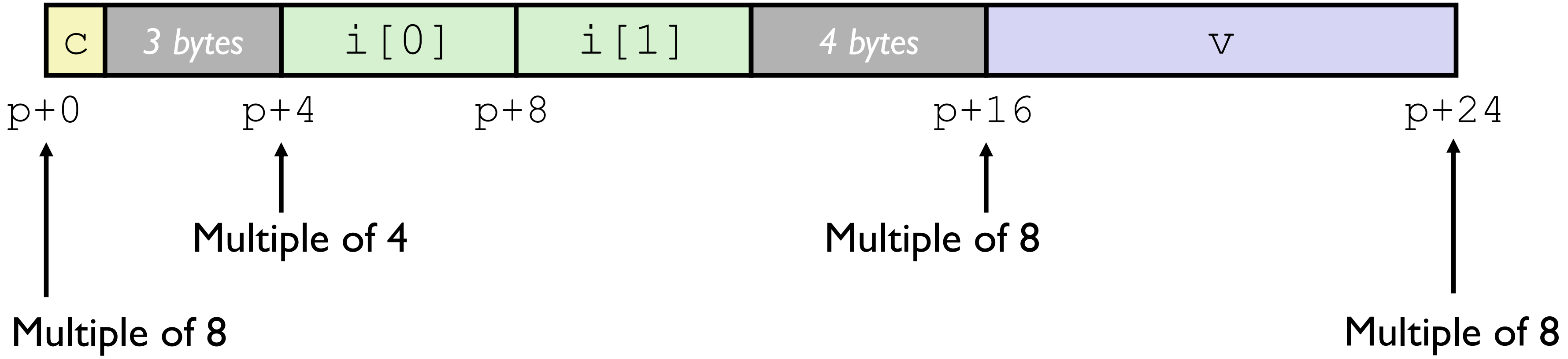
Structures and Alignment

- 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*



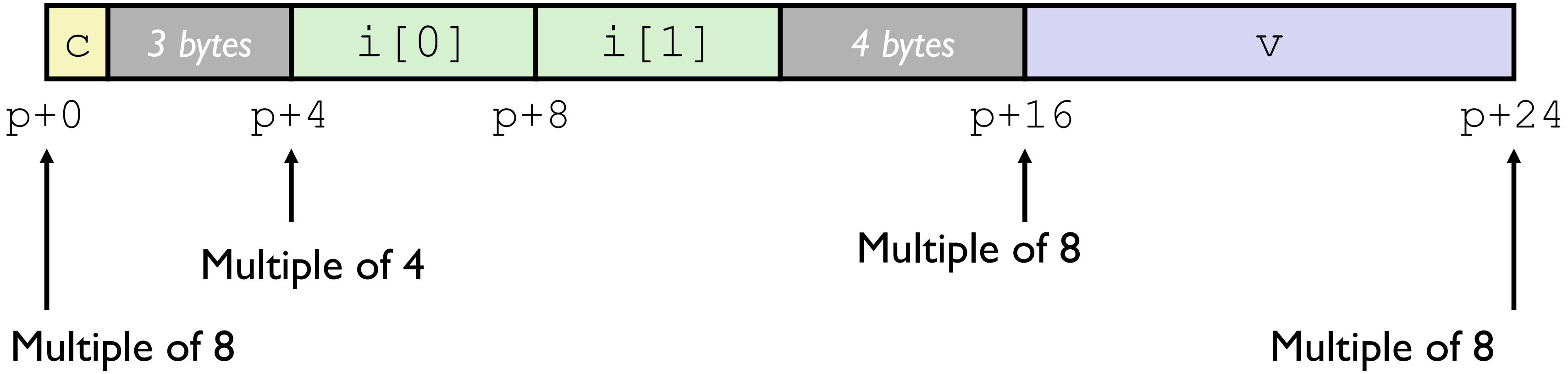
Alignment in IA-32

- 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:
    - 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

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 and structure length must be multiples of *K*
- Example (Windows):
  - *K* = 8, due to **`double`** element

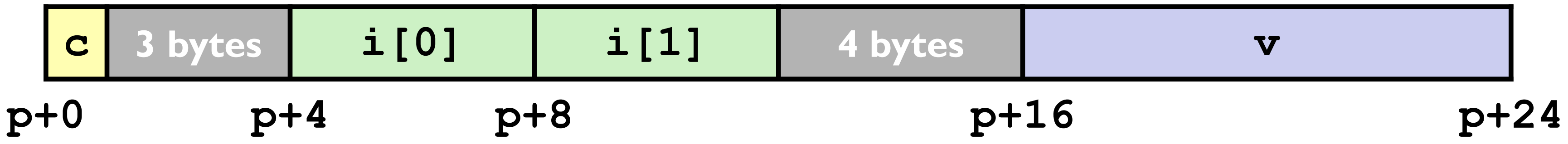
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



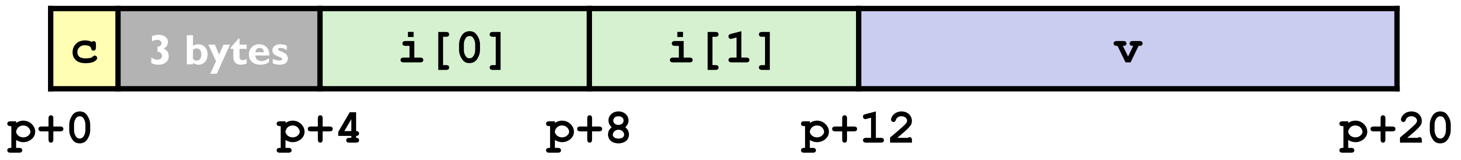
Different Alignment Conventions

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

- IA-32 Windows:
  - *K* = 8, due to **`double`** element



- IA-32 Linux
  - *K* = 4; **`double`** treated like a 4-byte data type

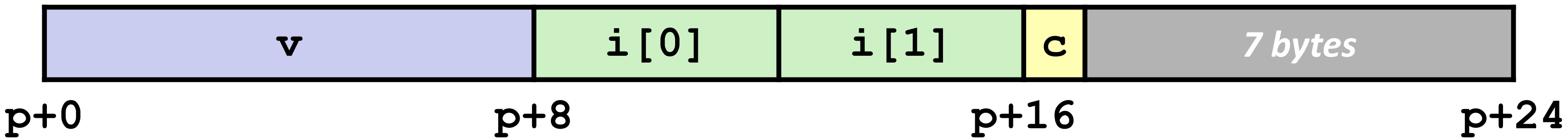


Meeting Overall Alignment Requirement

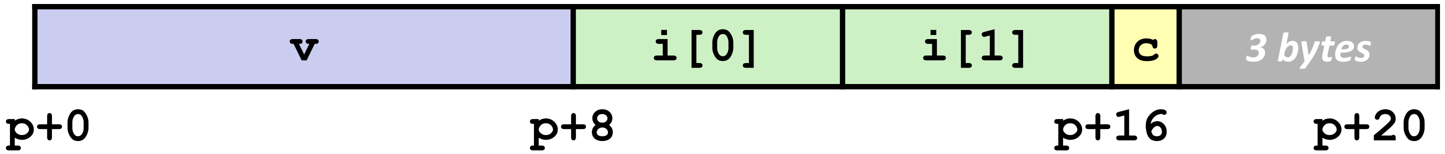
- For largest alignment requirement *K*
- Overall structure must be multiple of *K*

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

- IA-32 Windows:



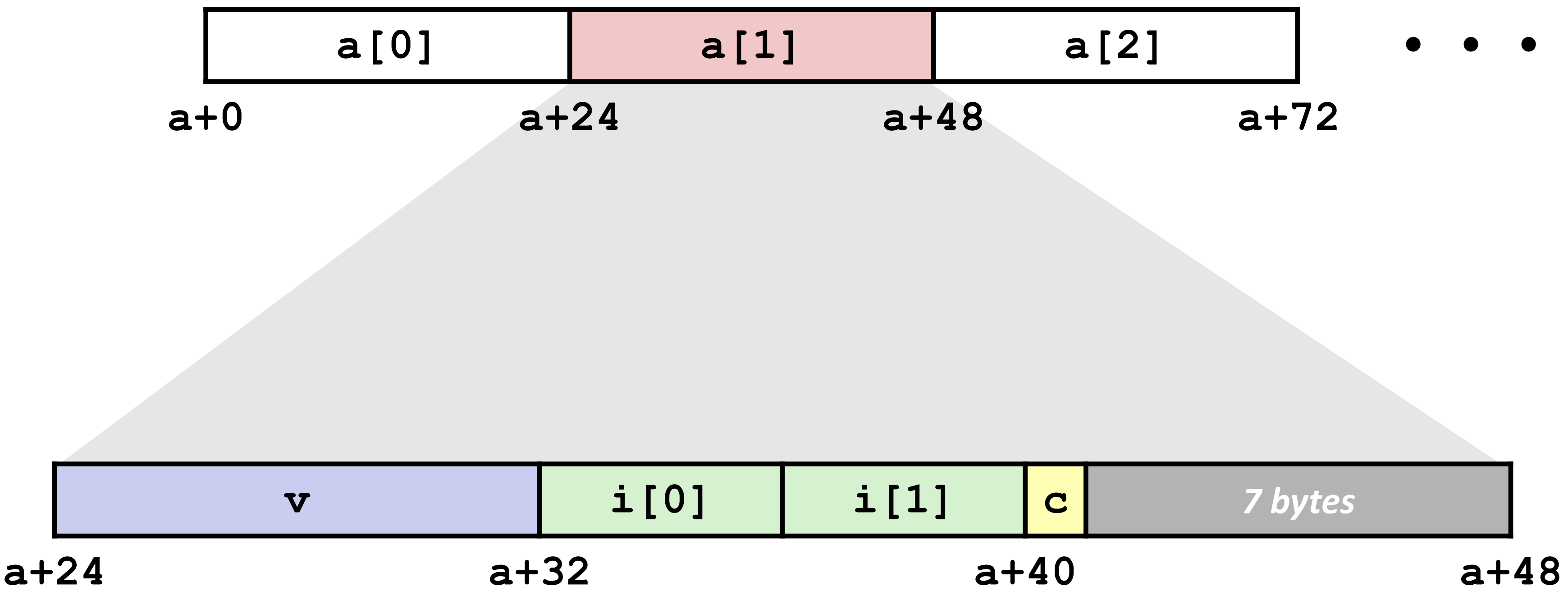
- IA-32 Linux



Arrays of Structures

- Overall structure length multiple of *K*
- Satisfy alignment requirement for every element
- Example: IA-32 Windows

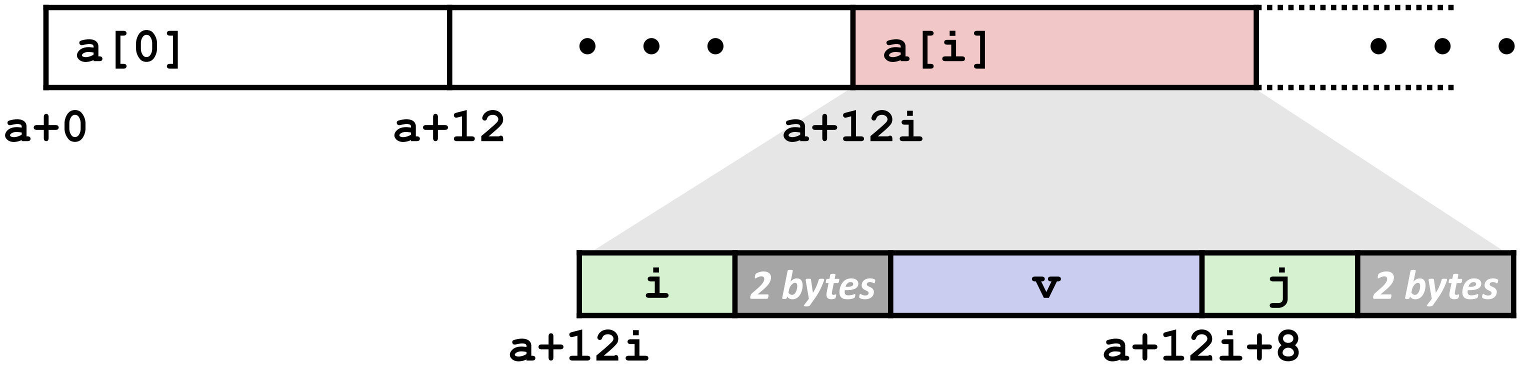
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

- What is the value of **`sizeof(struct S3)`**?
- Element **`j`** is at offset 8 within structure
- Assembler gives offset **`a+8`**
- Compute array offset **`12i`** for element **`i`**

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



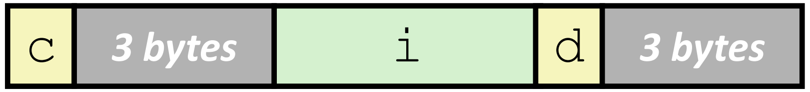
Saving Space

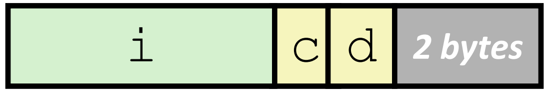
- Put large data types first

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

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

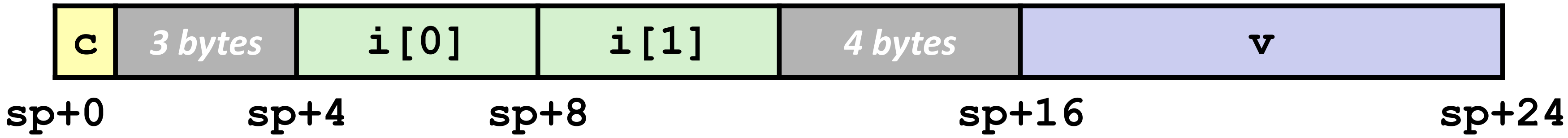
- Effect ($K = 4$)





Union Allocation

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



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

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