Data Structures in Assembly

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structs**
  - Alignment
- **Unions**

Structures

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

Memory Layout

- **Characteristics**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

Structures

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

- **Accessing Structure Member**
  - Given an instance of the struct, we can use the `.` operator, just like Java:
    - `struct rec r1;  r1.i = val;`
  - What if we have a *pointer* to a struct: `struct rec* r = &r1;`

Structures

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

- **Accessing Structure Member**
  - Given an instance of the struct, we can use the `.` operator, just like Java:
    - `struct rec r1;  r1.i = val;`
  - What if we have a *pointer* to a struct: `struct rec* r = &r1;`
    - Using `*` and `.` operators: `(*r).i = val;`
    - Or, use `->` operator for short: `r->i  = val;`
  - Pointer indicates first byte of structure; access members with offsets

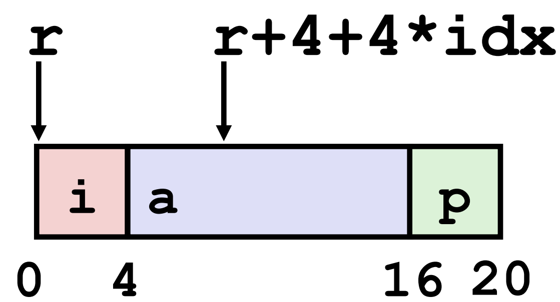
```
void
set_i(struct rec* r,
      int val)
{
     r->i = val;
}
```

IA32 Assembly

```
# %eax = val
# %edx = r
movl %eax,0(%edx)   # Mem[r+0] = val
```

Generating Pointer to Structure Member

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



- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time

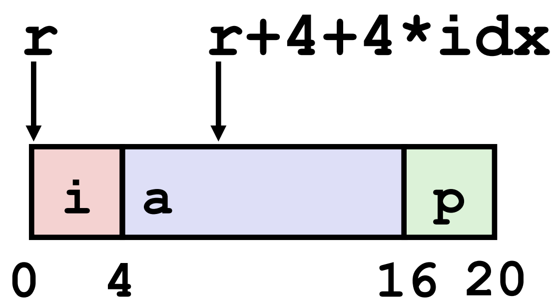
```
int* find_address_of_elem
 (struct rec* r, int idx)
{
   return &r->a[idx];
}
```

&(r->a[idx])

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Generating Pointer to Structure Member

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



- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time

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

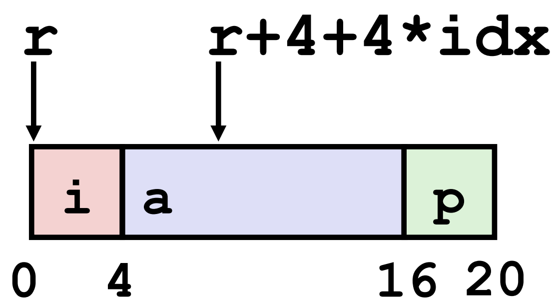
&(r->a[idx])

```
# %ecx = idx
# %edx = r
leal 4(%eax,%edx,4),%eax   # r+4*idx+4
```

OR

Accessing to Structure Member

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



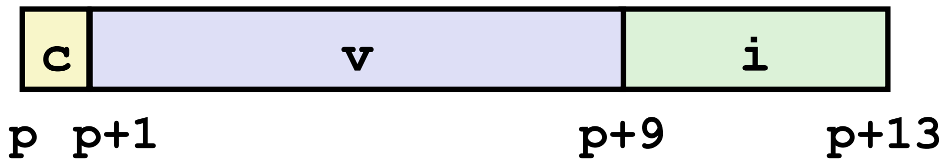
- **Reading Array Element**
  - Offset of each structure member *still* determined at compile time

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

```
# %ecx = idx
# %edx = r
movl 4(%eax,%edx,4),%eax  # Mem[r+4*idx+4]
```

Structures & Alignment

- **Unaligned Data**

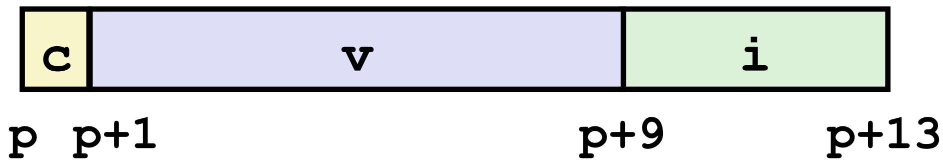



```
struct S1 {
  char c;
  double v;
  int i;
} * p;
```

- **How would it look like if data items were *aligned* *(address multiple of type size)* ?**

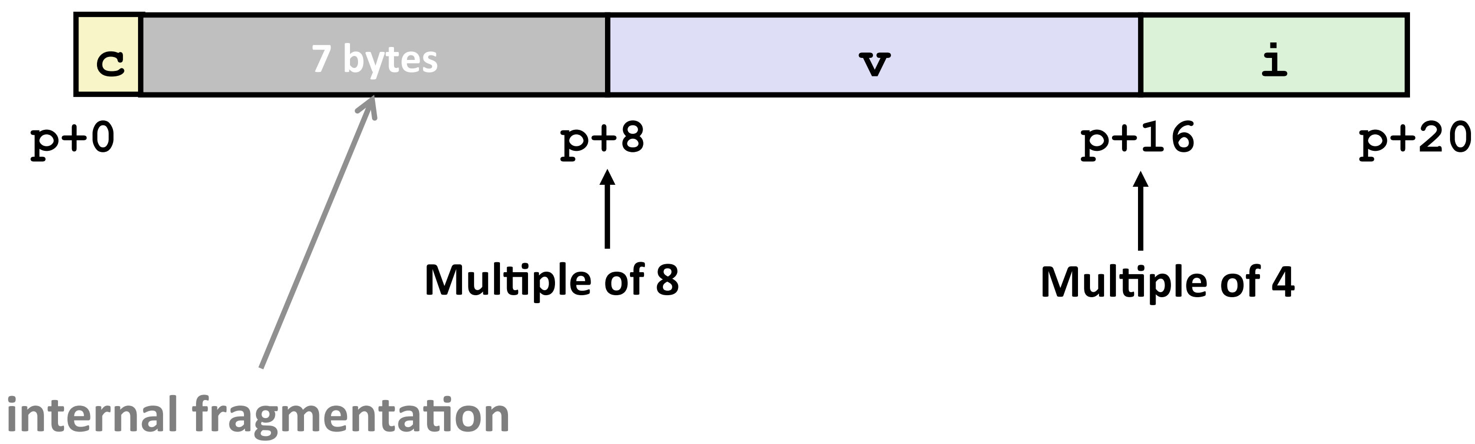
Structures & Alignment

- **Unaligned Data**




```
struct S1 {
  char c;
  double v;
  int i;
} * 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, Windows, Mac OS X, …
- **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, Windows, Mac OS X, …
- **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 these 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 liked 2 contiguous 4-byte primitive data items

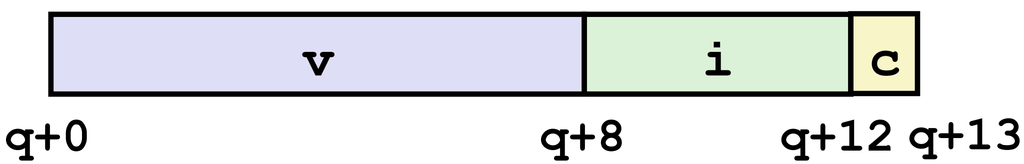
Saving Space

- **Put large data types first:**

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

```
struct S2 {
  double v;
  int i;
  char c;
} * q;
```

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

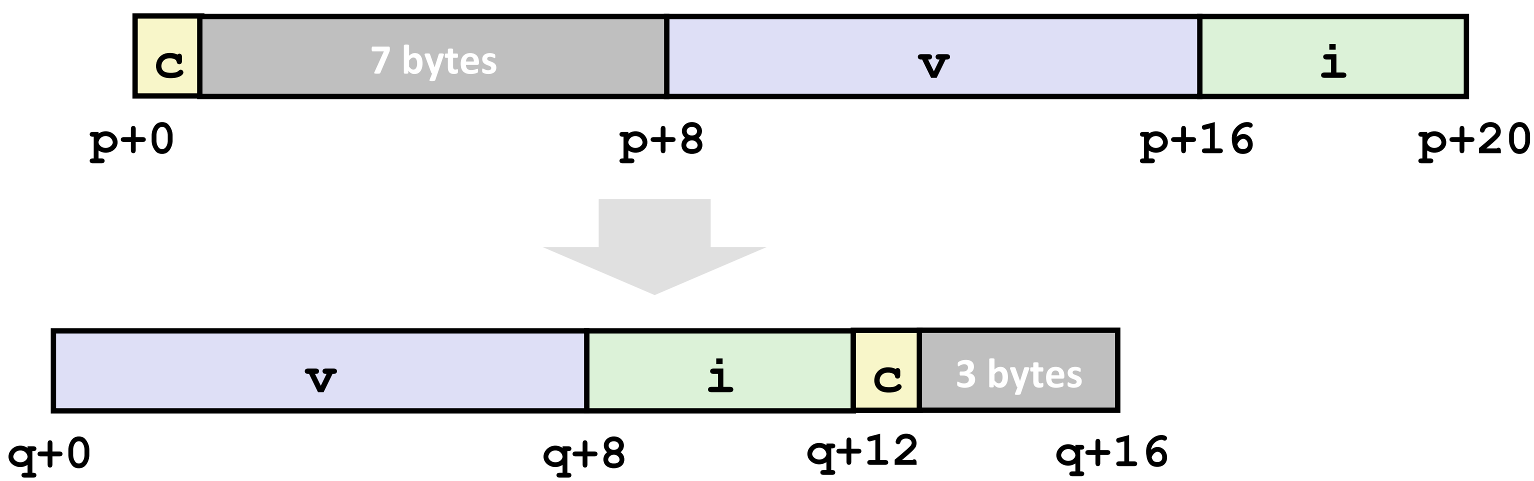


But actually…

Struct Alignment Principles

- **Size must be a multiple of the largest primitive type inside.**

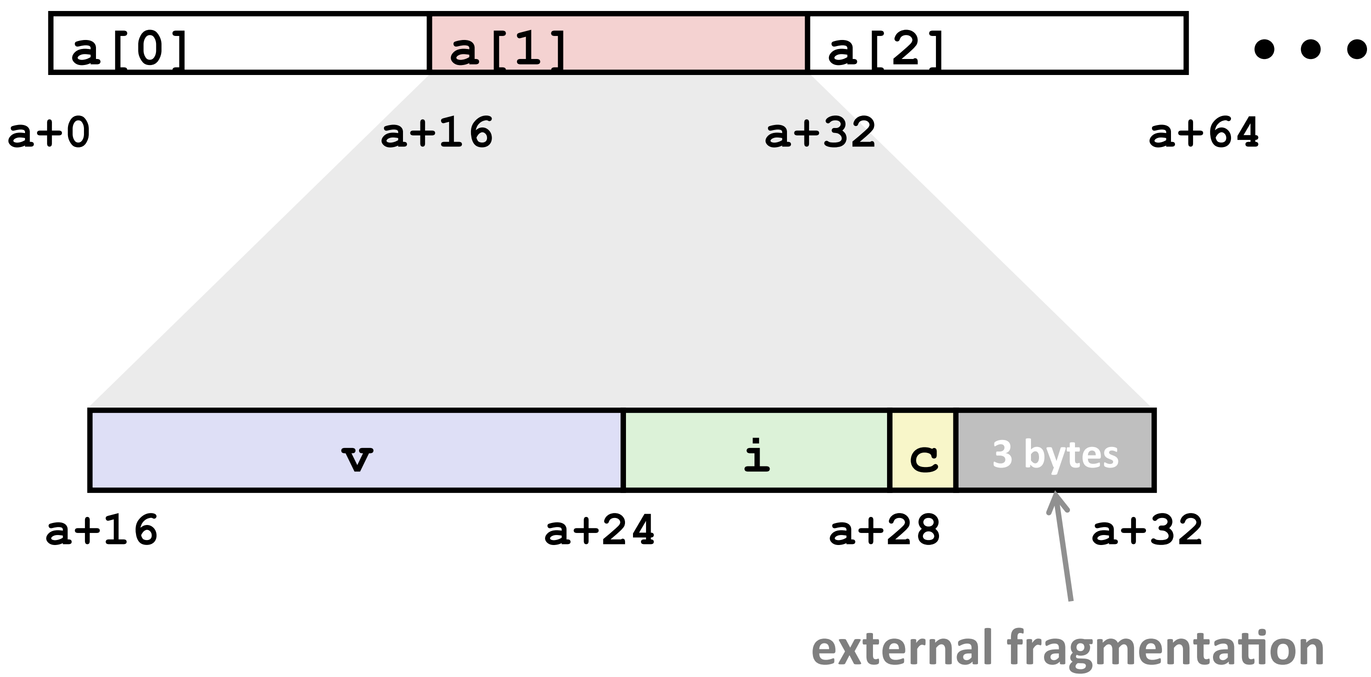
K **= 8 so** ***size*** **mod 8 = 0**



Arrays of Structures

- Satisfy alignment requirement for every element
- How would accessing an element work?

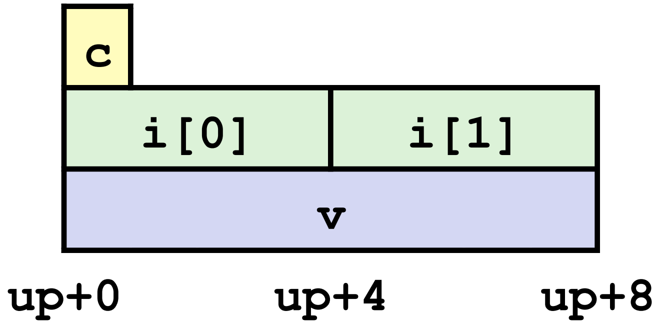
```
struct S2 {
  double v;
  int i;
  char c;
} a[10];
```



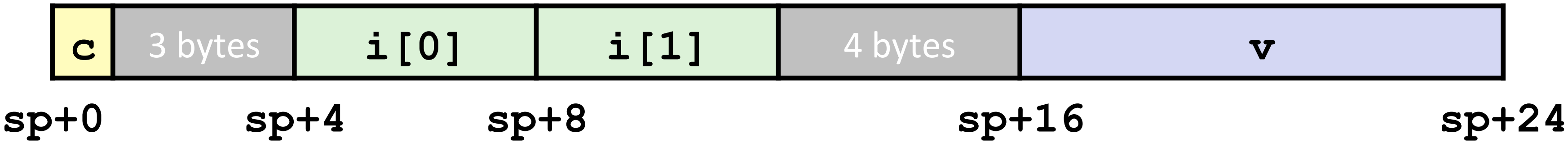
Unions

- Allocated according to largest element
- Can only use one member at a time

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




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



What Are Unions Good For?

- **Unions allow the same region of memory to be referenced as different types**
  - Different "views" of the same memory location
  - Can be used to circumvent C's type system (bad idea)
- **Better idea: use a struct inside a union to access some memory location either as a whole or by its parts**

Unions For Embedded Programming

```
typedef union
{
   unsigned char byte;
   struct {
      unsigned char b0:1;
      unsigned char b1:1;
      unsigned char b2:1;
      unsigned char b3:1;
      unsigned char reserved:4;
   } bits;
} hw_register;

hw_register reg;
reg.byte = 0x3F;          // 00111111_2
reg.bits.b2 = 0;          // 00111011_2
reg.bits.b3 = 0;          // 00110011_2
unsigned short a = reg.byte;
printf("0x%X\n", a);      // output: 0x33
```

(Note: the placement of these fields and other parts of this example are implementation-dependent)

Summary

- **Arrays in C**
  - Contiguous allocations of memory
  - No bounds checking
  - Can usually be treated like a pointer to first element
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Provide different views of the same memory location

Midterm Exam: Friday, July 26, in class

- **memory organization and addressing**
- **integer representations**
- **floating point representations**
- **x86 assembly programming, IA32 + x86-64**
  - addressing, arithmetic, basics
  - control flow
  - procedures, stacks, and associated conventions
- **pointers, arrays, and structs**
- **translation from C to assembly and back**
  - for all of the above, except floating point

Midterm Exam: Friday, July 26, in class

- **Closed book, closed notes, closed electronics, open mind!**
- **We provide you with:**
  - A list of powers of 2 in decimal (e.g., $2^{10} = 1024$)
  - A list of x86 assembly instructions and their meanings.
- **Likely: open Q+A review session(s)**
  - Your bring questions or we pick random problems from past exams.
  - Part of section on Thursday (vs. lots of buffer overflow fun)
  - Part of lecture Wednesday if we're ahead (I expect so)
- **HW 2 is good review.**
- **Lab 2 got you thinking in all the right ways about assembly.**