

Structs

15-411/15-611 Compiler Design

Seth Copen Goldstein

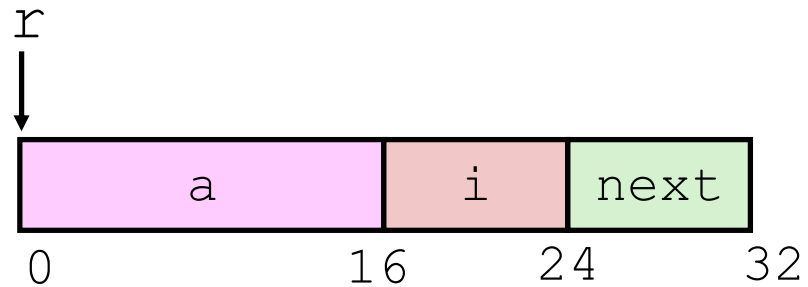
October 21, 2021

Today

- Structures and their machine requirements
- Small and big types
- Language issues
 - restrictions
 - parsing
 - static semantics
- Dynamic Semantics
 - &: pointers, arrays, and structures
 - assignment
- Registers and small type sizes

Structure Representation

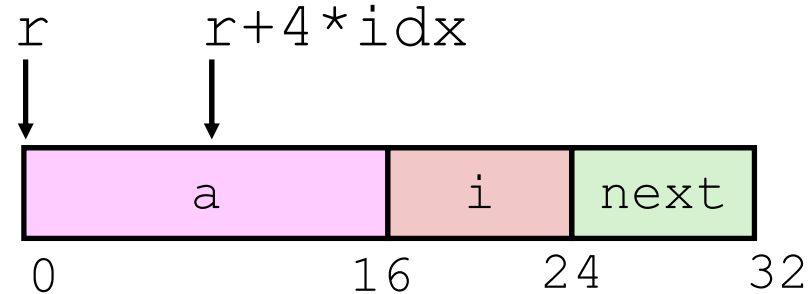
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- Structure represented as block of memory
 - **Big enough to hold all of the fields** Not important if you **stay in C0**
- Fields ordered according to declaration you **stay in C0**
 - 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**

Generating Pointer to Structure Member

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



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

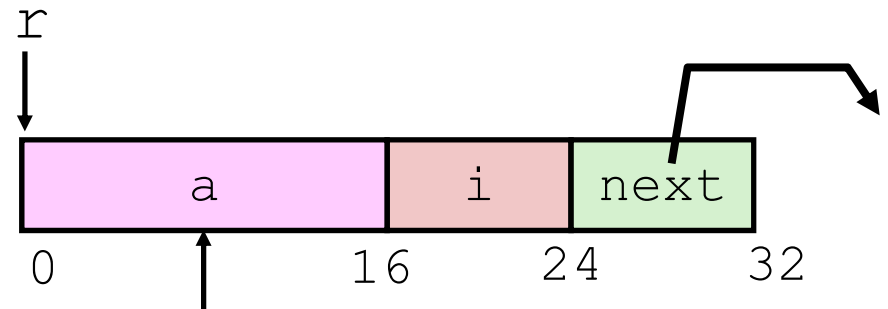
```
int *  
get_ap(struct rec *r, size_t idx)  
{  
    return &r->a[idx];  
}
```

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

Following Linked List

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

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



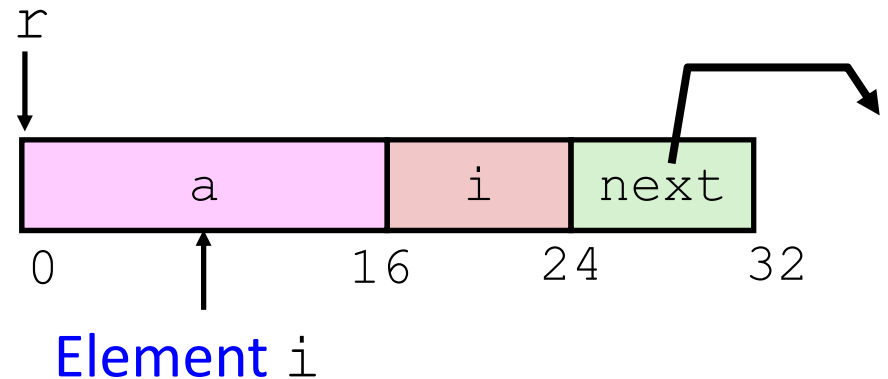
Register	Value
%rdi	r
%rsi	val

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

Which Registers to Use?

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

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

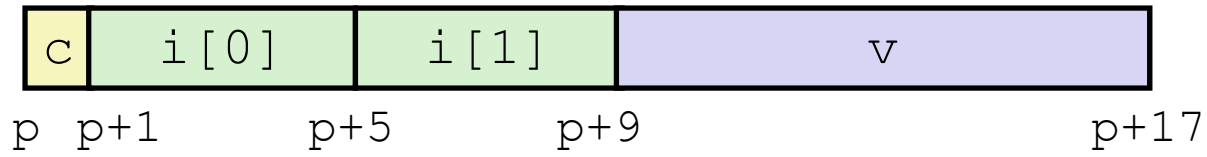


Register	Value
<code>%rdi</code>	<code>r</code>
<code>%esi</code>	<code>val</code>

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

Structures & Alignment

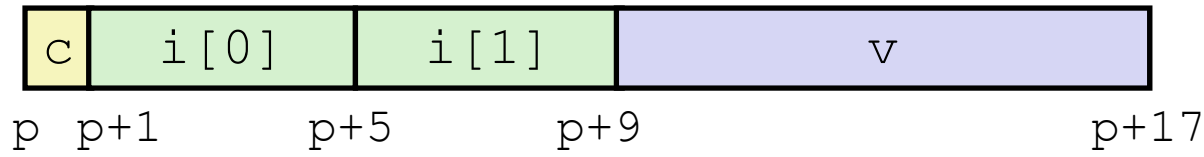
- Unaligned Data



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

Structures & Alignment

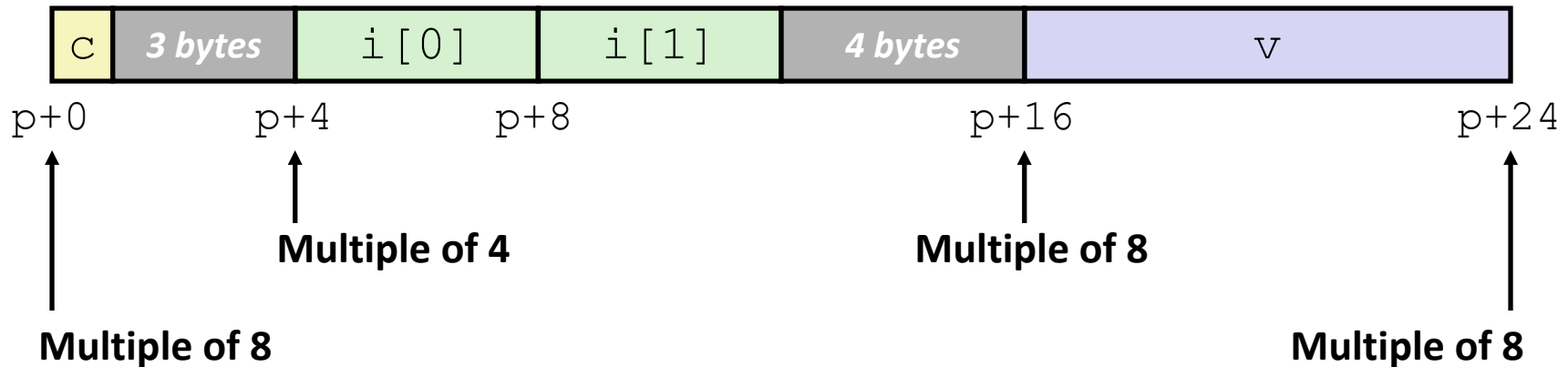
- Unaligned Data



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

- Aligned Data

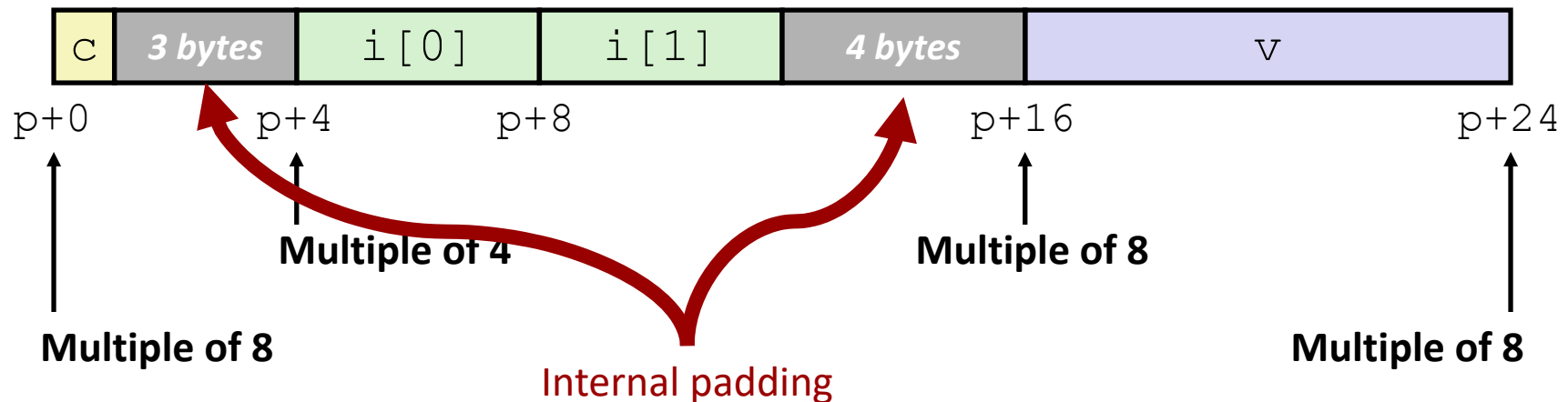
- Primitive data type requires ***K*** bytes
- Address must be multiple of ***K***



Satisfying Alignment within Structures

- Within structure:
 - 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

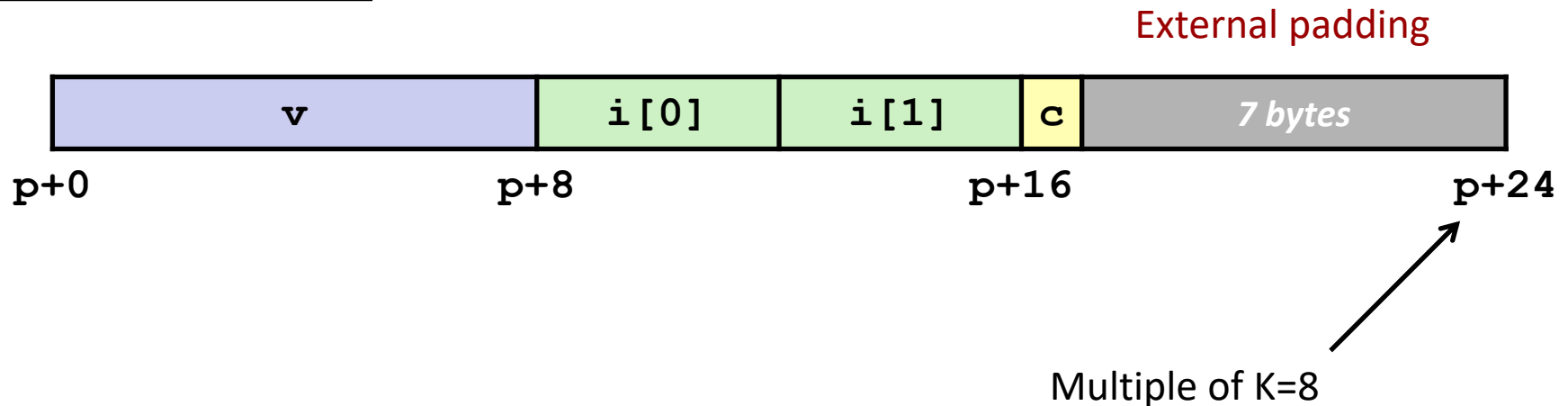
```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```



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;
```



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 cache lines (64 bytes). Intel states should avoid crossing 16 byte boundaries.
 - Virtual memory trickier when datum spans 2 pages (4 KB pages)
- Compiler
 - Inserts gaps in structure to ensure correct alignment of fields

Size & Alignment of C types (x86-64)

- 1 byte: **char**, ...
 - no restrictions on address
- 2 bytes: **short**, ...
 - lowest 1 bit of address must be 0_2
- 4 bytes: **int**, **float**, ...
 - lowest 2 bits of address must be 00_2
- 8 bytes: **double**, `long`, **char ***, ...
 - lowest 3 bits of address must be 000_2

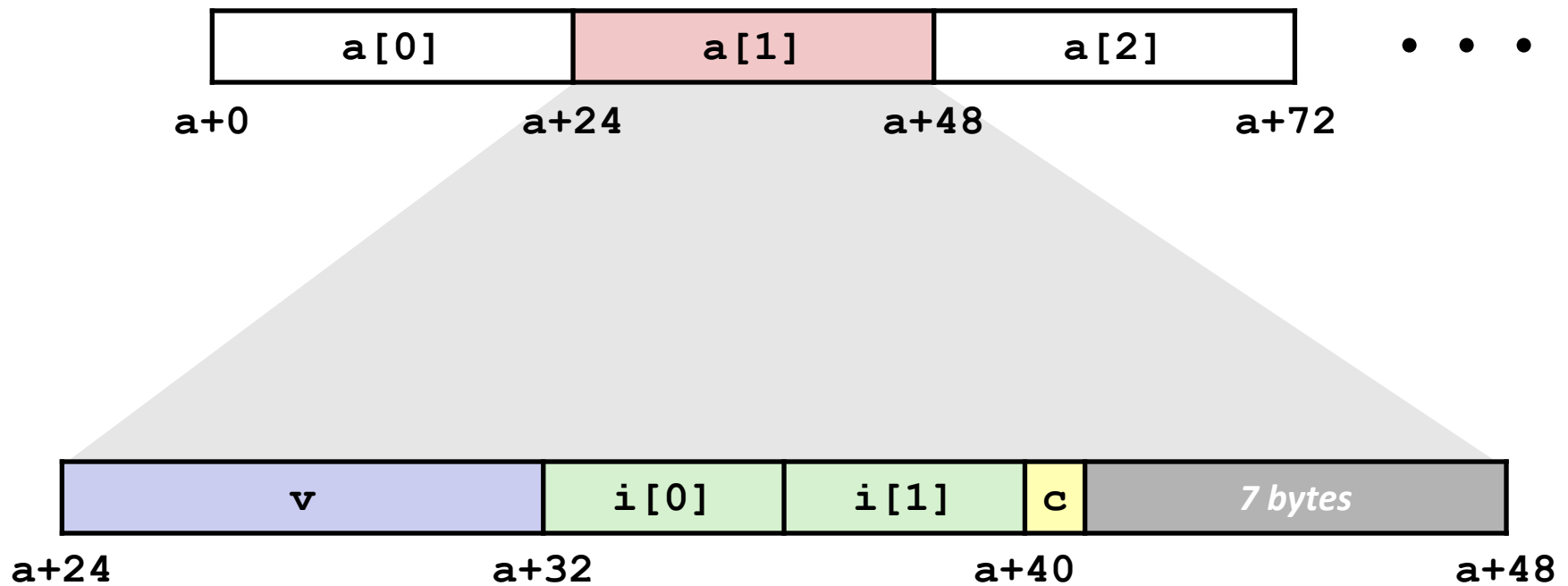
Size & Alignment of C0 types (x86-64)

- 4 bytes: **int**, **bool**
 - lowest 2 bits of address must be 00_2
- 8 bytes: τ^* , $\tau[]$
 - lowest 3 bits of address must be 000_2

Arrays of Structures

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

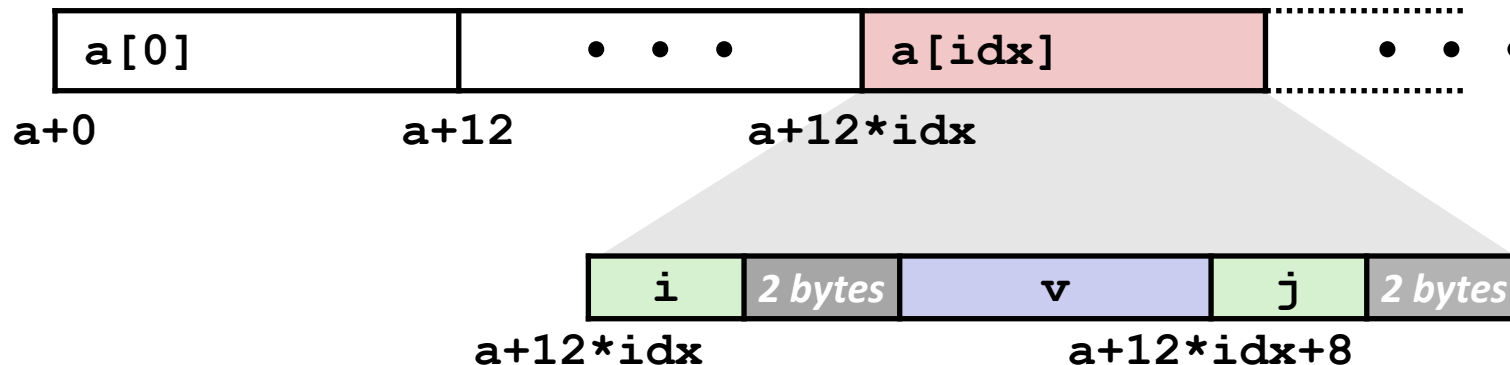
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} a[10];
```



Accessing Array Elements

- Compute array offset $12 * \text{idx}$
 - `sizeof(S3)`, including alignment spacers
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8`
 - Resolved during linking

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



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

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

L4 structs

- Must be allocated on the heap.
- Field names are in their own namespace
- In each **struct**, field names must be unique
- Can only be defined once.
- Can be used (in special cases) before declared!

Big and Small Types

- Small types fit in registers
 - int, bool, τ^* , $\tau[]$
 - 4 or 8 bytes in L4
- Large types are allocated on the heap
 - **struct** *s*

Restrictions vis a vis Small Types

All of the following must be small types:

- Local variables
- function parameters
- return values
- lval and rval in assignments
- Expressions
 - conditional expressions
 - `==` and `!=`
 - simple expressions (i.e., expressions as statements)

Namespaces

- Each struct definition creates its own namespace, so
 - fieldnames never conflict with other variables, function names, type names, field names in other **structs**
- Field names in a structure must be unique

Declare v. Define

- Declaration: `struct s;`
- Definition: `struct s { $\tau_1 f_1;$... $\tau_n f_n;$ };`
- Only 1 definition allowed
- If size is irrelevant:
 - Can be used before defined
 - Can be used without prior declaration!
- Size is relevant in
 - `alloc(struct s)` and
`alloc_array(struct s, e)`

Static Semantics

- Extend types

$$\tau ::= \text{int} \mid \text{bool} \mid \tau^* \mid \tau[] \mid \text{struct } s$$

- Extend expressions

$$d ::= \dots \mid d.f$$
$$e ::= \dots \mid e.f \mid e \rightarrow f$$

- Elaboration

$$e \rightarrow f \equiv (* e).f$$

- Typing

$$\frac{\Gamma \vdash e : \text{struct } s \quad s.f : \tau}{\Gamma \vdash e.f : \tau}$$

Note: **struct** s
must be defined

Static Semantics

- Extend types

$$\tau ::= \text{int} \mid \text{bool} \mid \tau^* \mid \tau[] \mid \text{struct } s$$

- Extend expressions

$$d ::= \dots \mid d.f$$
$$e ::= \dots \mid e.f \mid e \rightarrow f$$

- Elaboration

Because we defined d to be an expression no other typing rules are needed!

(Note: restrictions to small types on all previous rules.)

- Typing

$$\frac{\Gamma \vdash e : \text{struct } s \quad s.f : \tau}{\Gamma \vdash e.f : \tau}$$

Parsing L4

- What is meaning of “ $\mathbf{x} * \mathbf{y};$ ”

Parsing L4

- What is meaning of “**x * y;**”
 - Is it variable x times variable y?
 - Is it variable y is a pointer to type x?
- How to resolve this context sensitive Issue?
 - top-down parser will require backtracking
 - bottom-parser:
 - Solve after parse. How?
 - Get lexer involved.
(beware lexer lookahead - but suggested)

Dynamic Semantics - Approach

```
struct Point {  
    int x;  
    int y;  
};  
...  
struct Point* p = alloc_struct(point);
```

- What is value of **p->x**?

Dynamic Semantics - Approach

```
struct Point {  
    int x;  
    int y;  
};  
...  
struct Point* p = alloc_struct(point);
```

- What is value of $(*p) . x$?
- What is value of $*p$?

Dynamic Semantics - Approach

```
struct Point {  
    int x;  
    int y;  
};  
...  
struct Point* p = alloc_struct(point);
```

- What is value of $(*p) . x$?
- What is value of $*p$?
- Two approaches

Dynamic Semantics - Approach

```
struct Point {  
    int x;  
    int y;  
};  
...  
struct Point* p = alloc_struct(point);
```

- What is value of $(*p) . x$?
- What is value of $*p$?
- Approach one: $*p$ is entire structure
 - read in entire structure
 - select field

$$\begin{array}{l} H ; S ; \eta \vdash e.f \triangleright K \\ H ; S ; \eta \vdash \{x = v_1, y = v_2\} \triangleright (_ . y , K) \end{array} \quad \longrightarrow \quad \begin{array}{l} H ; S ; \eta \vdash e \triangleright (_ . y , K) \\ H ; S ; \eta \vdash v_2 \triangleright K \end{array}$$

Dynamic Semantics - Approach

```
struct Point {  
    int x;  
    int y;  
};  
...  
struct Point* p = alloc_struct(point);
```

- What is value of $(*p) . x$?
- What is value of $*p$?
- Approach one: $*p$ is entire structure
- Approach two: $*p$ has no meaning in and of itself, rather p is an address and $(*p) . x$ is an address calculation followed by a load

Address of operator: &

- Introduce, into the dynamic semantics, the address-of operation, &
- So, $(*\mathbf{p}) . \mathbf{f}$ becomes:
 - get address of \mathbf{p}
 - get offset from start of \mathbf{p} to \mathbf{f}
 - calculate a = sum of above
 - load proper number of bytes from a
- I.e., $(*\mathbf{p}) . \mathbf{f} \Rightarrow *(\&((*\mathbf{p}) . \mathbf{f}))$
- Notice similarity to $*d$ used as an lval (from last lecture)

Writing to the heap

- left to right evaluation of address and rval

$$H ; S ; \eta \vdash \text{assign}(*d, e) \blacktriangleright K \quad \longrightarrow \quad H ; S ; \eta \vdash d \triangleright (\text{assign}(*_, e), K)$$

$$H ; S ; \eta \vdash a \triangleright (\text{assign}(*_, e), K) \quad \longrightarrow \quad H ; S ; \eta \vdash e \triangleright (\text{assign}(*a, _), K)$$

- Then making assignment (if $a \neq 0$)

$$\begin{array}{ll}
 H ; S ; \eta \vdash c \triangleright (\text{assign}(*a, _), K) & \longrightarrow \quad H[a \mapsto c] ; S ; \eta \vdash \text{nop} \blacktriangleright K & (a \neq 0) \\
 H ; S ; \eta \vdash c \triangleright (\text{assign}(*a, _), K) & \longrightarrow \quad \text{exception}(\text{mem}) & (a = 0)
 \end{array}$$

Address of operator: &

- Introduce, into the dynamic semantics, the address-of operation, &
- So, $(*\mathbf{p}) . \mathbf{f}$ becomes:
 - get address of \mathbf{p}
 - get offset from start of \mathbf{p} to \mathbf{f}
 - calculate $a = \text{sum of above}$
 - load proper number of bytes from a
- I.e., $(*\mathbf{p}) . \mathbf{f} \Rightarrow *(\&((*\mathbf{p}) . \mathbf{f}))$
- Notice similarity to $*d$ used as an lval (from last lecture)

Field access

- We evaluate $e.f$ as $*(&(e.f))$

$$H ; S ; \eta \vdash e.f \triangleright K \quad \longrightarrow \quad H ; S ; \eta \vdash *(&(e.f)) \triangleright K$$

Addresses and Large Types

$$H ; S ; \eta \vdash \&(*e) \triangleright K \quad \longrightarrow \quad H ; S ; \eta \vdash e \triangleright K$$

Addresses and Large Types

$$\begin{array}{ll} H ; S ; \eta \vdash \&(*e) \triangleright K & \longrightarrow & H ; S ; \eta \vdash e \triangleright K \\ \\ H ; S ; \eta \vdash \&(e.f) \triangleright K & \longrightarrow & H ; S ; \eta \vdash \&e \triangleright (\&(_ .f) , K) \\ H ; S ; \eta \vdash a \triangleright (\&(_ .f) , K) & \longrightarrow & H ; S ; \eta \vdash a + \text{offset}(s, f) \triangleright K \\ & & & (a \neq 0, a : \text{struct } s) \end{array}$$

Addresses and Large Types

$$\begin{array}{ll} H ; S ; \eta \vdash \&(*e) \triangleright K & \longrightarrow & H ; S ; \eta \vdash e \triangleright K \\ \\ H ; S ; \eta \vdash \&(e.f) \triangleright K & \longrightarrow & H ; S ; \eta \vdash \&e \triangleright (\&(_ .f) , K) \\ H ; S ; \eta \vdash a \triangleright (\&(_ .f) , K) & \longrightarrow & H ; S ; \eta \vdash a + \text{offset}(s, f) \triangleright K \\ & & & (a \neq 0, a : \text{struct } s) \\ H ; S ; \eta \vdash a \triangleright (\&(_ .f) , K) & \longrightarrow & \text{exception}(\text{mem}) \quad (a = 0) \\ \\ H ; S ; \eta \vdash \&(e_1[e_2]) \triangleright K & \longrightarrow & H ; S ; \eta \vdash e_1 \triangleright (\&(_ [e_2]) , K) \\ H ; S ; \eta \vdash a \triangleright (\&(_ [e_2]) , K) & \longrightarrow & H ; S ; \eta \vdash e_2 \triangleright (\&(a[_]) , K) \end{array}$$

Addresses and Large Types

$$\begin{array}{ll}
 H ; S ; \eta \vdash \&(*e) \triangleright K & \longrightarrow & H ; S ; \eta \vdash e \triangleright K \\
 \\
 H ; S ; \eta \vdash \&(e.f) \triangleright K & \longrightarrow & H ; S ; \eta \vdash \&e \triangleright (\&(_ .f) , K) \\
 H ; S ; \eta \vdash a \triangleright (\&(_ .f) , K) & \longrightarrow & H ; S ; \eta \vdash a + \text{offset}(s, f) \triangleright K \\
 & & & (a \neq 0, a : \text{struct } s) \\
 H ; S ; \eta \vdash a \triangleright (\&(_ .f) , K) & \longrightarrow & \text{exception}(\text{mem}) \quad (a = 0) \\
 \\
 H ; S ; \eta \vdash \&(e_1[e_2]) \triangleright K & \longrightarrow & H ; S ; \eta \vdash e_1 \triangleright (\&(_ [e_2]) , K) \\
 H ; S ; \eta \vdash a \triangleright (\&(_ [e_2]) , K) & \longrightarrow & H ; S ; \eta \vdash e_2 \triangleright (\&(a[_]) , K) \\
 H ; S ; \eta \vdash i \triangleright (\&(a[_]) , K) & \longrightarrow & H ; S ; \eta \vdash a + i|\tau| \triangleright K \\
 & & & a \neq 0, 0 \leq i < \text{length}(a), a : \tau[]
 \end{array}$$

Example

```
struct Point {  
    int x;  
    int y;  
};  
struct Line {  
    struct Point A;  
    struct Point B;  
};  
...  
struct Line* L = alloc(struct Line);  
...  
int x = L->B.y;
```

After elaboration =>

Example

```
struct Point {
    int x;
    int y;
};
struct Line {
    struct Point A;
    struct Point B;
};
...
struct Line* L = alloc(struct Line);
...
int x = (*L).B.y;
```

$x = (*L) . B . y ;$

$H ; S ; \eta \vdash \text{assign}(x, (*L) . B . y) \blacktriangleright K$

$x = (*L) . B . y ;$

$H ; S ; \eta \vdash \text{assign}(x, (*L) . B . y) \blacktriangleright K$
 $\longrightarrow H ; S ; \eta \vdash ((*L) . B . y) \triangleright (\text{assign}(x, \blacksquare) , K)$

$x = (*L) . B . y ;$

$H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K$

$\longrightarrow H ; S ; \eta \vdash ((*L).B.y) \triangleright (\text{assign}(x, _), K)$

$\longrightarrow H ; S ; \eta \vdash *(&((*L).B.y)) \triangleright (\text{assign}(x, _), K)$

$x = (*L) . B . y ;$

$H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K$

$\longrightarrow H ; S ; \eta \vdash ((*L).B.y) \triangleright (\text{assign}(x, _), K)$

$\longrightarrow H ; S ; \eta \vdash *(\&((*L).B.y)) \triangleright (\text{assign}(x, _), K)$

$\longrightarrow H ; S ; \eta \vdash \&((*L).B.y) \triangleright (*(\blacksquare), \text{assign}(x, _), K)$

$x = (*L) . B . y ;$

$H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K$

$\longrightarrow H ; S ; \eta \vdash ((*L).B.y) \triangleright (\text{assign}(x, _) , K)$

$\longrightarrow H ; S ; \eta \vdash *(&((*L).B.y)) \triangleright (\text{assign}(x, _) , K)$

$\longrightarrow H ; S ; \eta \vdash \&(\boxed{(*L).B}.y) \triangleright (*(_) , \text{assign}(x, _) , K)$

$\longrightarrow H ; S ; \eta \vdash \&((*L).B) \triangleright (\&(\blacksquare).y) , *(_) , \text{assign}(x, _) , K)$

$x = (*L) . B . y ;$

$H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K$

$\longrightarrow H ; S ; \eta \vdash ((*L).B.y) \triangleright (\text{assign}(x, _) , K)$

$\longrightarrow H ; S ; \eta \vdash *(&((*L).B.y)) \triangleright (\text{assign}(x, _) , K)$

$\longrightarrow H ; S ; \eta \vdash \&((*L).B.y) \triangleright (*(_) , \text{assign}(x, _) , K)$

$\longrightarrow H ; S ; \eta \vdash \&(*L).B \triangleright (\&(_.y) , *(_) , \text{assign}(x, _) , K)$

$\longrightarrow H ; S ; \eta \vdash \&(*L) \triangleright (\&(\blacksquare).B) , \&(_.y) , *(_) , \text{assign}(x, _) , K)$

$$\mathbf{x} = (*L) . B . y ;$$

	$H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K$
\longrightarrow	$H ; S ; \eta \vdash ((*L).B.y) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash *(&((*L).B.y)) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B.y) \triangleright (*(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B) \triangleright (\&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \boxed{\&(*L)} \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash L \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$

$$\mathbf{x} = (*L) . B . y ;$$

$$\begin{array}{l}
H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K \\
\longrightarrow H ; S ; \eta \vdash ((*L).B.y) \quad \triangleright (\text{assign}(x, _) , K) \\
\longrightarrow H ; S ; \eta \vdash *(&((*L).B.y)) \quad \triangleright (\text{assign}(x, _) , K) \\
\longrightarrow H ; S ; \eta \vdash \&((*L).B.y) \quad \triangleright (*(_) , \text{assign}(x, _) , K) \\
\longrightarrow H ; S ; \eta \vdash \&((*L).B) \quad \triangleright (\&(_.y) , *(_) , \text{assign}(x, _) , K) \\
\longrightarrow H ; S ; \eta \vdash \&(*L) \quad \triangleright (\&(_.B) , \&(_.y) , *(_) , \text{assign}(x, _) , K) \\
\longrightarrow H ; S ; \eta \vdash \boxed{L} \quad \triangleright (\&(_.B) , \&(_.y) , *(_) , \text{assign}(x, _) , K) \\
\longrightarrow H ; S ; \eta \vdash a \quad \triangleright (\&(_.B) , \&(_.y) , *(_) , \text{assign}(x, _) , K) \\
\hspace{15em} (\text{given that } H ; S ; \eta(L) = a, a \neq 0)
\end{array}$$

$$\mathbf{x} = (*L) . B . y ;$$

	$H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K$
\longrightarrow	$H ; S ; \eta \vdash ((*L).B.y) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash *(&((*L).B.y)) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B.y) \triangleright (*(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B) \triangleright (\&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&(*L) \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash L \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash a \triangleright (\boxed{\&(_.B)}, \&(_.y), *(_), \text{assign}(x, _), K)$ (given that $H ; S ; \eta(L) = a, a \neq 0$)
\longrightarrow	$H ; S ; \eta \vdash a + 8 \triangleright (\&(_.y), *(_), \text{assign}(x, _), K)$ (given that $\text{offset}(\text{line}, B) = 8$)

$x = (*L) . B . y ;$

	$H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K$
\longrightarrow	$H ; S ; \eta \vdash ((*L).B.y) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash *(&((*L).B.y)) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B.y) \triangleright (*(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B) \triangleright (\&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&(*L) \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash L \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash a \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$ <i>(given that $H ; S ; \eta(L) = a, a \neq 0$)</i>
\longrightarrow	$H ; S ; \eta \vdash a + 8 \triangleright (\&(_.y), *(_), \text{assign}(x, _), K)$ <i>(given that $\text{offset}(\text{line}, B) = 8$)</i>
\longrightarrow	$H ; S ; \eta \vdash a + 12 \triangleright *(_), \text{assign}(x, _), K$ <i>(given that $\text{offset}(\text{point}, y) = 4$)</i>

$$\mathbf{x} = (*L) . B . y ;$$

	$H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K$
\longrightarrow	$H ; S ; \eta \vdash ((*L).B.y) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash *(&((*L).B.y)) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B.y) \triangleright (*(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B) \triangleright (\&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&(*L) \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash L \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash a \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$ (given that $H ; S ; \eta(L) = a, a \neq 0$)
\longrightarrow	$H ; S ; \eta \vdash a + 8 \triangleright (\&(_.y), *(_), \text{assign}(x, _), K)$ (given that $\text{offset}(\text{line}, B) = 8$)
\longrightarrow	$H ; S ; \eta \vdash a + 12 \triangleright *(_), \text{assign}(x, _), K$ (given that $\text{offset}(\text{point}, y) = 4$)
\longrightarrow	$H ; S ; \eta \vdash c \triangleright (\text{assign}(x, _), K)$ (given that $H(a + 12) = c$)

$x = (*L) . B . y ;$

	$H ; S ; \eta \vdash \text{assign}(x, (*L).B.y) \blacktriangleright K$
\longrightarrow	$H ; S ; \eta \vdash ((*L).B.y) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash *(&((*L).B.y)) \triangleright (\text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B.y) \triangleright (*(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&((*L).B) \triangleright (\&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash \&(*L) \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash L \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$
\longrightarrow	$H ; S ; \eta \vdash a \triangleright (\&(_.B), \&(_.y), *(_), \text{assign}(x, _), K)$ <i>(given that $H ; S ; \eta(L) = a, a \neq 0$)</i>
\longrightarrow	$H ; S ; \eta \vdash a + 8 \triangleright (\&(_.y), *(_), \text{assign}(x, _), K)$ <i>(given that $\text{offset}(\text{line}, B) = 8$)</i>
\longrightarrow	$H ; S ; \eta \vdash a + 12 \triangleright *(_), \text{assign}(x, _), K$ <i>(given that $\text{offset}(\text{point}, y) = 4$)</i>
\longrightarrow	$H ; S ; \eta \vdash c \triangleright \boxed{\text{assign}(x, _), K}$ <i>(given that $H(a + 12) = c$)</i>
\longrightarrow	$H ; S ; \eta[x \mapsto c] \vdash \text{nop} \blacktriangleright K$

Allocation

- Similar to regular alloc, but size is defined by the struct, as per alignment rules.
- Initialization (for L4) is to set all to 0.

$$H ; S ; \eta \vdash \text{alloc_struct}(\text{struct } s) \triangleright K \quad \longrightarrow \quad H' ; S ; \eta \vdash a \triangleright K$$

$$a = H(\text{next})$$

$$z = \text{sizeof}(\text{struct } s)$$

$$H' = H[a \mapsto 0, \dots, a + (z - 1) \mapsto 0, \text{next} \mapsto a + z]$$

Assignment

- Can simplify all rules for assignment to large types, i.e., heap allocated locations

$$\begin{array}{llll} H ; S ; \eta \vdash \text{assign}(d, e) \blacktriangleright K & \longrightarrow & H ; S ; \eta \vdash \&d \triangleright (\text{assign}(_, e), K) & (d \neq x) \\ H ; S ; \eta \vdash a \triangleright (\text{assign}(_, e), K) & \longrightarrow & H ; S ; \eta \vdash e \triangleright (\text{assign}(a, _), K) & & \\ H ; S ; \eta \vdash v \triangleright (\text{assign}(a, _), K) & \longrightarrow & H[a \mapsto v] ; S ; \eta \vdash \text{nop} \blacktriangleright K & & (a \neq 0) \\ H ; S ; \eta \vdash v \triangleright (\text{assign}(a, _), K) & \longrightarrow & \text{exception}(\text{mem}) & & (a = 0) \end{array}$$

- Likewise, can simplify assignment ops

$d \odot = e$

- When d is a small type, e.g., a variable x , elaborate to $assign(x, x \odot e)$
- When d is an address on the heap elaborate to $asnop(d, \odot, e)$ and we have:

$$\begin{array}{ll} H ; S ; \eta \vdash asnop(d, \odot, e) \blacktriangleright K & \longrightarrow H ; S ; \eta \vdash \&d \triangleright (asnop(_, \odot, e), K) \\ H ; S ; \eta \vdash a \triangleright (asnop(_, \odot, e), K) & \longrightarrow H ; S ; \eta \vdash e \triangleright (asnop(a, \odot, _), K) \\ H ; S ; \eta \vdash v \triangleright (asnop(a, \odot, _), K) & \longrightarrow H ; S ; \eta \vdash assign(a, *a \odot v) \blacktriangleright K \end{array}$$

- evaluation of address
- evaluation of value on right-hand side
- assignment (which continues as before)

Registers and small types

- two type sizes held in registers
 - 32-bit: int & bool
 - 64-bit: pointer
- Be careful moving these around
 - **movl, cmpl, addl vs. movq, cmpq, addq**
- Track stack space, heap space, etc. required
- Suggestions:
 - track sizes for your temps, vars, etc.
 - use explicit extend ops in your IR

32/64-bit implementation

- Use explicit extend ops in IR, e.g.,
dest64 <- zeroextend src32
dest64 <- signextend src32
- Remember, zeroextend comes for free:
 - **movl %eax, %eax**
sets high order 32 bits of **%rax** to 0!
 - similarly for other instructions

Starting next week analysis and optimization