

C Structs

A **struct** combines multiple field values:

```
struct element {
    char    name[10];
    char    symbol[5];
    int     atom_no;
    double  mass;
};

struct element h = {"Hydrogen",
                   "H",
                   1,
                   1.00794};

void report(struct element e) {
    printf("%s is atomic number %d\n",
           e.name,
           e.atom_no);
}
```

C Structs and Typedefs

Common to give a short name with **typedef**

```
struct element {
    char    name[10];
    char    symbol[5];
    int     atom_no;
    double  mass;
};

typedef struct element ELT;

ELT h = {"Hydrogen", .....};

void report(ELT e) {
    .....
}
```

C Structs and Typedefs

Common to recycle the `struct` name with `typedef`

```
struct element {
    char    name[10];
    char    symbol[5];
    int     atom_no;
    double  mass;
};

typedef struct element element;

element h = {"Hydrogen", ....};

void report(element e) {
    ....
}
```

C Structs and Typedefs

Shorthand: declare **struct** and **typedef** at once

```
typedef struct element {
    char    name[10];
    char    symbol[5];
    int     atom_no;
    double  mass;
} element;

element h = {"Hydrogen", ....};

void report(element e) {
    ....
}
```

C Structs and Typedefs

... but must use `struct` for self-reference

```
typedef struct element {  
    char    name[10];  
    char    symbol[5];  
    int     atom_no;  
    double  mass;  
    struct  element *next_in_table;  
} element;
```

C Struct Pointers and Fields

Structs are more like integers than arrays

```
void add_neutrons(element e) {
    e.mass++;
}

int main() {
    element h = {"Hydrogen", "H", 1, 1.00794};
    add_neutrons(h);
    printf("%f\n", h.mass);
}
```

Prints 1.00794

C Struct Pointers and Fields

Structs are more like integers than arrays

```
void add_neutrons(element *e) {
    (*e).mass++;
}

int main() {
    element h = {"Hydrogen", "H", 1, 1.00794};
    add_neutrons(&h);
    printf("%f\n", h.mass);
}
```

Prints 2.00794

C Struct Pointers and Fields

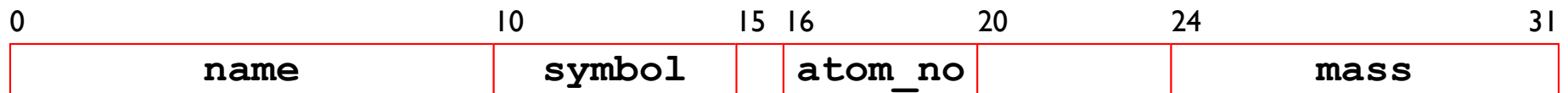
Use `__->__` as a shorthand for `(*__).__`

```
void add_neutrons(element *e) {  
    printf("Old mass: %f\n", e->mass);  
    e->mass++;  
}
```


Structure Layout

A **struct** value has its fields' values in order

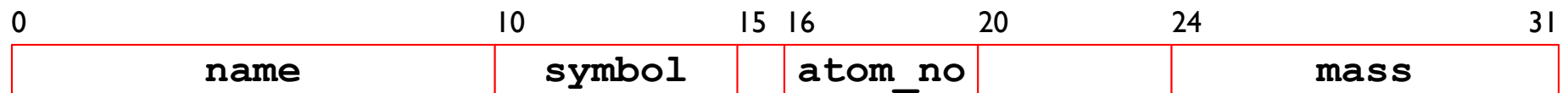
```
struct element {  
    char    name[10];  
    char    symbol[5];  
    int     atom_no;  
    double  mass;  
};
```



Structure Layout

A `struct` value has its fields' values in order

```
struct element {  
    char    name[10];  
    char    symbol[5];  
    int     atom_no;  
    double  mass;  
};
```

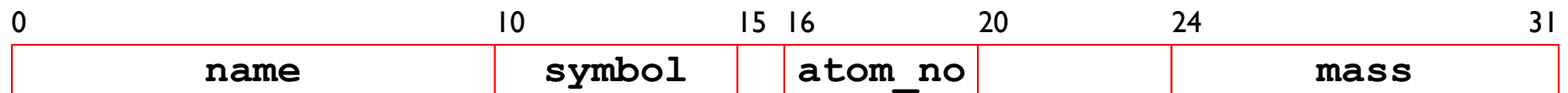


padding — because `int` is 4-byte **aligned**

Structure Layout

A **struct** value has its fields' values in order

```
struct element {  
    char    name[10];  
    char    symbol[5];  
    int     atom_no;  
    double  mass;  
};
```



padding — because **double** is 8-byte **aligned**

Inspecting Structure Layout

```
#include <stdio.h>
#include <stddef.h>

typedef struct element {
    char    name[10];
    char    symbol[5];
    int     atom_no;
    double  mass;
} element;

int main() {
    element e;
    printf("%ld\n", (char *)&e.atom_no - (char *)&e);
    return 0;
}
```

[Copy](#)

Alignment Rules

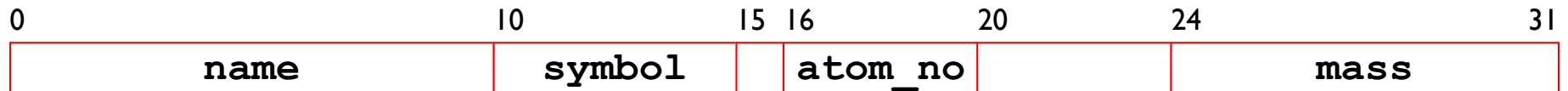
Why align?

- Load or store that spans quad-word boundaries is inefficient
- Virtual memory is trickier when a datum spans pages

Alignment Rules

Roughly, primitive data of size $N \Rightarrow$ align on N bytes

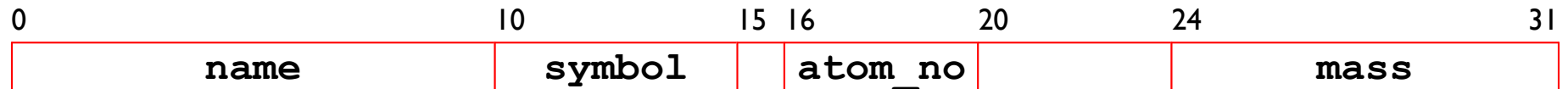
```
struct element {  
    char    name[10];  
    char    symbol[5];  
    int     atom_no;  
    double  mass;  
};
```



Alignment Rules

Roughly, primitive data of size $N \Rightarrow$ align on N bytes

```
struct element {  
    char    name[10];  
    char    symbol[5];  
    int     atom_no;  
    double  mass;  
};
```



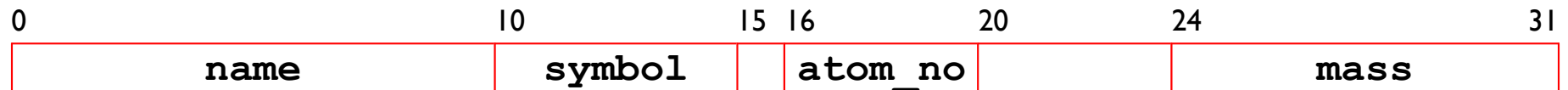
Required on some machines

Advised on x86-64

Alignment Rules

Pad struct size to multiple of largest alignment

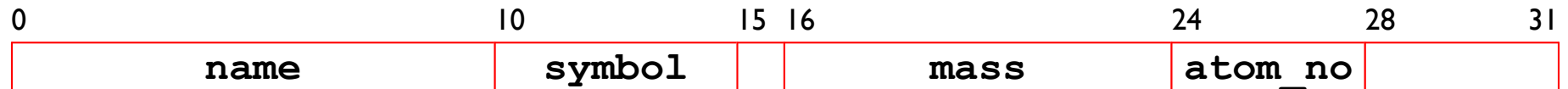
```
struct element {  
    char    name[10];  
    char    symbol[5];  
    int     atom_no;  
    double  mass;  
};
```



Alignment Rules

Pad struct size to multiple of largest alignment

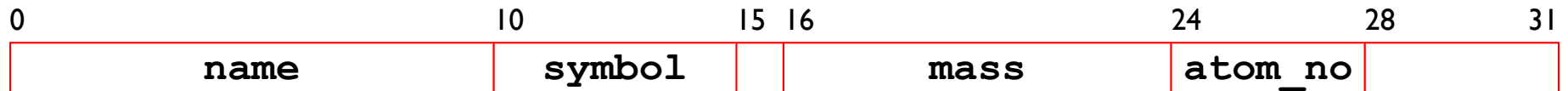
```
struct element {  
    char    name[10];  
    char    symbol[5];  
    double  mass;  
    int     atom_no;  
};
```



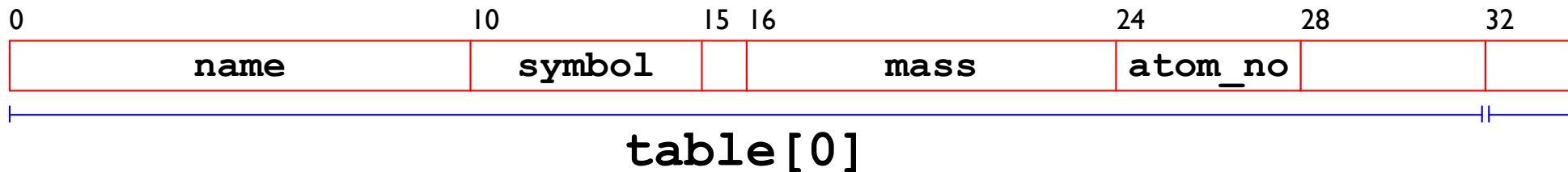
Alignment Rules

Pad struct size to multiple of largest alignment

```
struct element {  
    char    name[10];  
    char    symbol[5];  
    double  mass;  
    int     atom_no;  
};
```



```
struct element table[2];
```

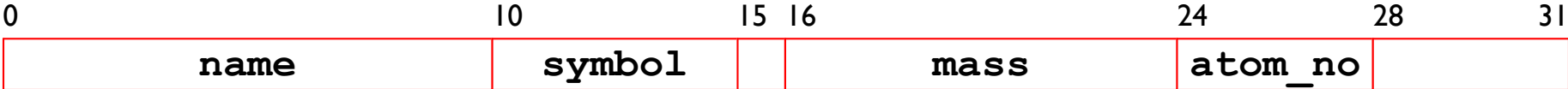


Alignment Rules

Pad struct size to multiple of largest alignment

```

struct element {
    char    name[10];
    char    symbol[5];
    double  mass;
    int     atom_no;
};
    
```



```

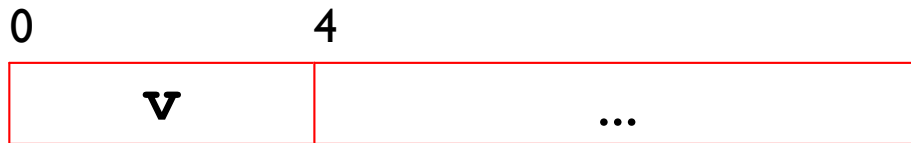
struct element table[2];
    
```



table[1]

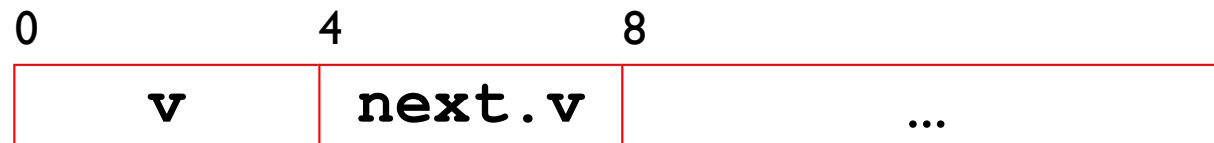
Linked Lists

```
struct int_list {  
    int v;  
    struct int_list next;  
};
```



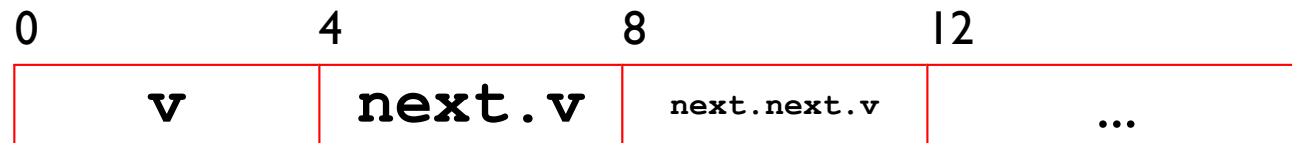
Linked Lists

```
struct int_list {  
    int v;  
    struct int_list next;  
};
```



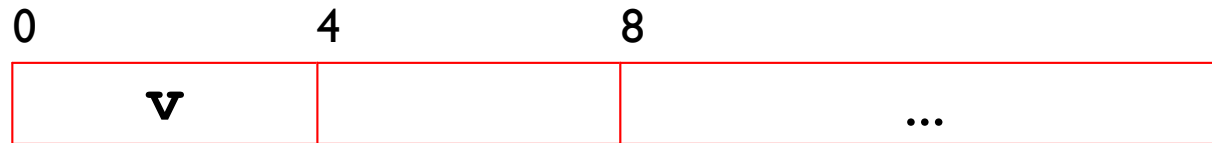
Linked Lists

```
struct int_list {  
    int v;  
    struct int_list next;  
};
```



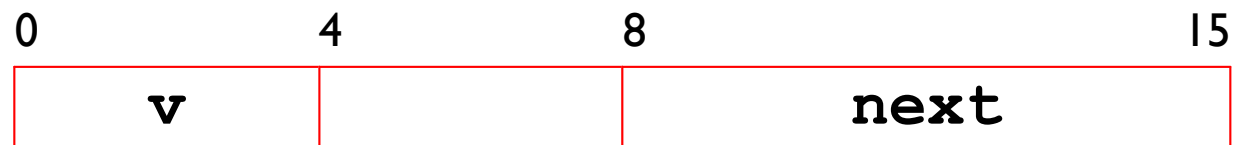
Linked Lists

```
struct int_list {  
    int v;  
    struct int_list *next;  
};
```



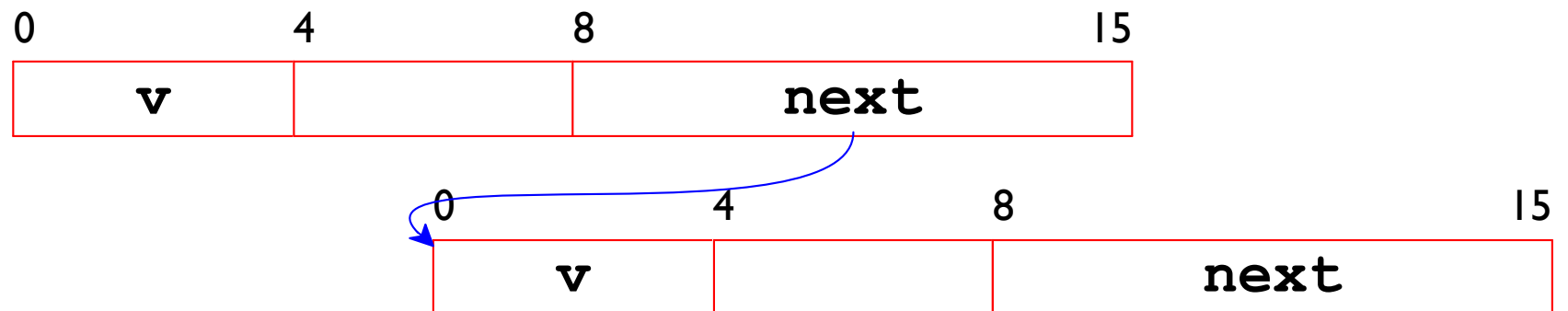
Linked Lists

```
struct int_list {  
    int v;  
    struct int_list *next;  
};
```



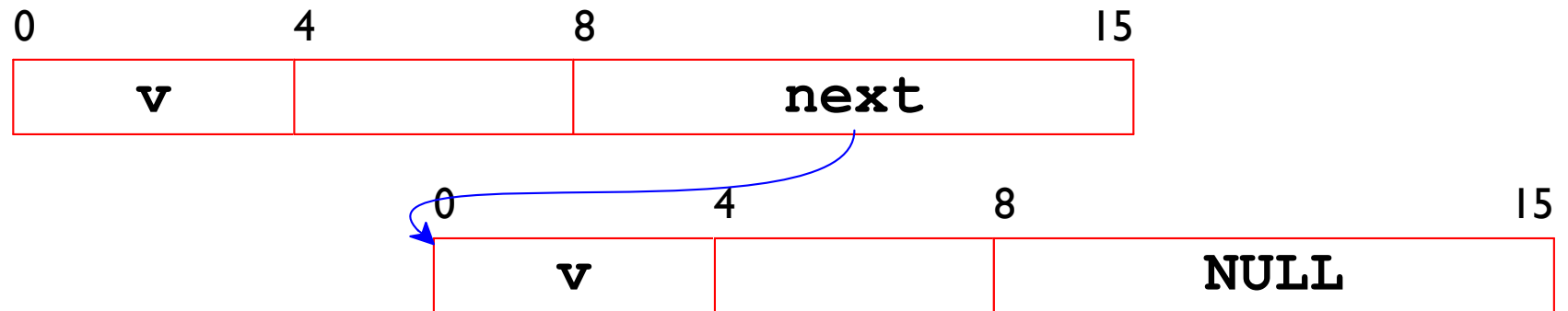
Linked Lists

```
struct int_list {  
    int v;  
    struct int_list *next;  
};
```



Linked Lists

```
struct int_list {  
    int v;  
    struct int_list *next;  
};
```



Example: Traversing a Linked List

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

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

[Copy](#)

Example: Traversing a Linked List

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

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

0	12	16	23
a	i	next	

[Copy](#)

Example: Traversing a Linked List

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

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

0	4	8	12	16	23
a[0]	a[1]	a[2]	i	next	

[Copy](#)

Example: Traversing a Linked List

	0	4	8	12	16	23
	a[0]	a[1]	a[2]	i	next	


```

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

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

```

.L3:
    movslq    12(%rdi), %rax
    movl     %esi, (%rdi,%rax,4)
    movq     16(%rdi), %rdi
    testq    %rdi, %rdi
    jne      .L3
    
```

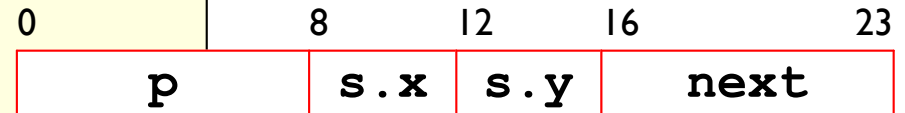
```

%rdi = r
%esi = val
loop:
    i = M[r+12]
    M[r+4×i] = val
    r = M[r+16]
    Test r
    if !NULL goto loop
    
```

Exercise: Machine Code with Structs

```
struct prob {  
    int* p;  
    struct {  
        int x;  
        int y;  
    } s;  
    struct prob* next;  
};
```

```
void sp_init(struct prob* sp) {  
    sp->s.x = ___;  
    sp->p = ___;  
    sp->next = ___;  
}
```



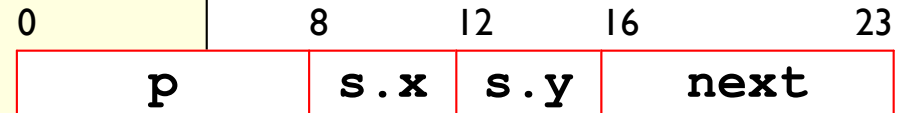
%rdi = sp

```
movl 12(%rdi),%edx  
movl %edx,8(%rdi)  
leaq 8(%rdi),%rdx  
movq %rdx,(%rdi)  
movq %rdi,16(%rdi)
```

Exercise: Machine Code with Structs

```
struct prob {  
    int* p;  
    struct {  
        int x;  
        int y;  
    } s;  
    struct prob* next;  
};
```

```
void sp_init(struct prob* sp) {  
    sp->s.x = sp->s.y;  
    sp->p = __;  
    sp->next = __;  
}
```



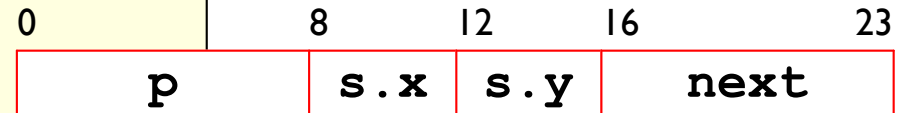
%rdi = sp

```
movl 12(%rdi),%edx  
movl %edx,8(%rdi)  
leaq 8(%rdi),%rdx  
movq %rdx,(%rdi)  
movq %rdi,16(%rdi)
```


Exercise: Machine Code with Structs

```
struct prob {  
    int* p;  
    struct {  
        int x;  
        int y;  
    } s;  
    struct prob* next;  
};
```

```
void sp_init(struct prob* sp) {  
    sp->s.x = sp->s.y;  
    sp->p = &sp->s.x;  
    sp->next = __;  
}
```



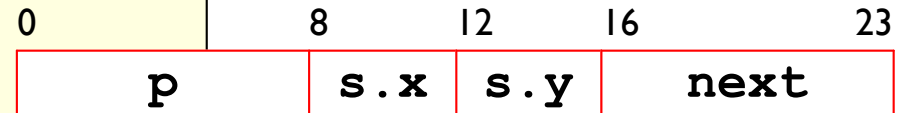
%rdi = sp

```
movl 12(%rdi),%edx  
movl %edx,8(%rdi)  
leaq 8(%rdi),%rdx  
movq %rdx,(%rdi)  
movq %rdi,16(%rdi)
```

Exercise: Machine Code with Structs

```
struct prob {  
    int* p;  
    struct {  
        int x;  
        int y;  
    } s;  
    struct prob* next;  
};
```

```
void sp_init(struct prob* sp) {  
    sp->s.x = sp->s.y;  
    sp->p = &sp->s.x;  
    sp->next = sp;  
}
```



%rdi = sp

```
movl 12(%rdi),%edx  
movl %edx,8(%rdi)  
leaq 8(%rdi),%rdx  
movq %rdx,(%rdi)  
movq %rdi,16(%rdi)
```

C Unions

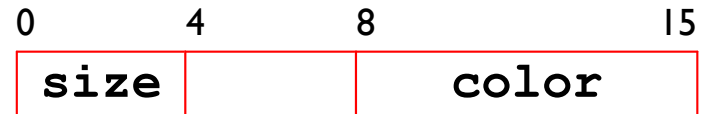
A `struct` is an *and* of field values:

```
/* a number and a string */  
struct t_shirt {  
    int    size;  
    char *color;  
};
```

C Unions

A **struct** is an *and* of field values:

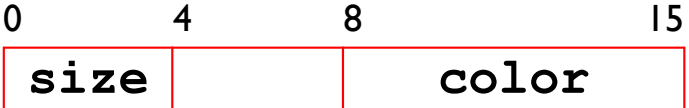
```
/* a number and a string */  
struct t_shirt {  
    int    size;  
    char  *color;  
};
```



C Unions

A **struct** is an *and* of field values:

```
/* a number and a string */  
struct t_shirt {  
    int    size;  
    char  *color;  
};
```



A diagram illustrating the memory layout of the struct t_shirt. It shows a horizontal bar divided into three segments. The first segment, labeled 'size', starts at memory address 0 and ends at 4. The second segment is empty and starts at 4 and ends at 8. The third segment, labeled 'color', starts at 8 and ends at 15. The addresses 0, 4, 8, and 15 are marked above the bar.

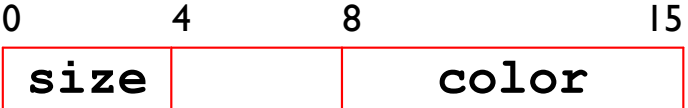
A **union** is an *or* of field values:

```
/* a number or a string */  
union homework_result {  
    int    grade;  
    char  *excuse;  
};
```

C Unions

A **struct** is an *and* of field values:

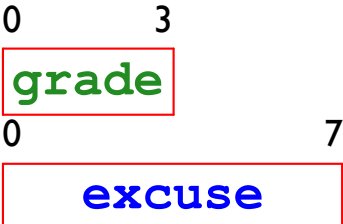
```
/* a number and a string */
struct t_shirt {
    int    size;
    char *color;
};
```



The diagram shows a memory layout for the struct t_shirt. It consists of three boxes: a box labeled 'size' from address 0 to 4, an empty box from 4 to 8, and a box labeled 'color' from 8 to 15. The addresses 0, 4, 8, and 15 are marked above the boxes.

A **union** is an *or* of field values:

```
/* a number or a string */
union homework_result {
    int    grade;
    char *excuse;
};
```

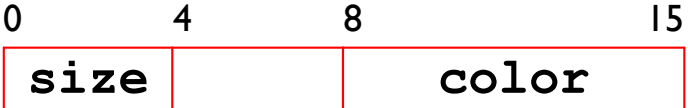


The diagram shows a memory layout for the union homework_result. It consists of two boxes: a box labeled 'grade' from address 0 to 3, and a box labeled 'excuse' from address 0 to 7. The addresses 0, 3, and 7 are marked above the boxes.

C Unions

A **struct** is an *and* of field values:

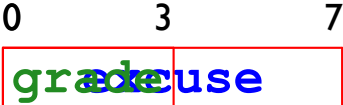
```
/* a number and a string */  
struct t_shirt {  
    int    size;  
    char  *color;  
};
```



A memory layout diagram for the struct t_shirt. It shows a horizontal bar divided into three sections. The first section, labeled 'size', starts at address 0 and ends at 4. The second section is empty and ends at 8. The third section, labeled 'color', starts at 8 and ends at 15. The labels 'size' and 'color' are centered in their respective sections.

A **union** is an *or* of field values:

```
/* a number or a string */  
union homework_result {  
    int    grade;  
    char  *excuse;  
};
```

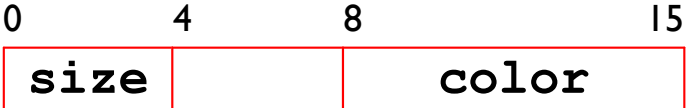


A memory layout diagram for the union homework_result. It shows a horizontal bar divided into two sections. The first section, labeled 'grade', starts at address 0 and ends at 3. The second section, labeled 'excuse', starts at 3 and ends at 7. The labels 'grade' and 'excuse' are centered in their respective sections.

C Unions

A **struct** is an *and* of field values:

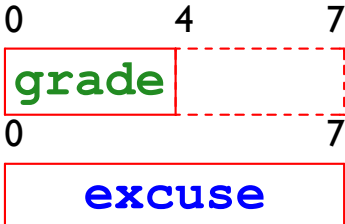
```
/* a number and a string */
struct t_shirt {
    int    size;
    char  *color;
};
```



0	4	8	15
size		color	

A **union** is an *or* of field values:

```
/* a number or a string */
union homework_result {
    int    grade;
    char  *excuse;
};
```



0	4	7
grade		
0		7
excuse		

Setting Union Fields

```
#include <stdio.h>

union homework_result {
    int    grade;
    char *excuse;
};

void got_doctor_note(union homework_result *h) {
    h->excuse = "illness";
}

int main() {
    union homework_result h;

    h.grade = 0;
    got_doctor_note(&h);
    printf("%d\n", h.grade);

    return 0;
}
```

Using Union Fields

Combine to combine **union** with **struct** field to select variant

```
struct homework_record {
    int graded;
    union homework_result r;
};

void got_doctor_note(struct homework_record *h) {
    h->graded = 0;
    h->r.excuse = "illness";
}

int main() {
    struct homework_record h;

    h.graded = 1;
    h.r.grade = 0;

    got_doctor_note(&h);

    if (h.graded)
        printf("%d\n", h.r.grade);
    else
        printf("%s\n", h.r.excuse);

    return 0;
}
```

[Copy](#)

Using a Union to Reinterpret Bytes

```
#include <stdio.h>

union i_or_f {
    int i;
    float f;
};

int main() {
    union i_or_f v;
    v.i = 0x24400000;
    printf("%g\n", v.f);
    return 0;
}
```

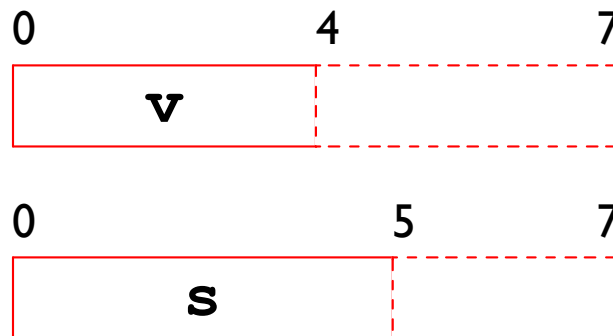
[Copy](#)

Union Alignment

Alignment is \geq max field size and multiple of alignment

```
union u {  
    float v;  
    char  s[5];  
}
```

`sizeof(union u)` is 8:



Controlling Struct Alignment

```
typedef struct step {  
    char mode;  
    double height;  
} step_t;
```

Controlling Struct Alignment

```
/* select 1-byte alignment for everything */  
#pragma pack(1)  
  
typedef struct step {  
    char mode;  
    double height;  
} step_t;  
  
/* resume default alignments */  
#pragma pack()
```