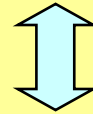


Structures

- 2
- defines a new name for an existing type
 - ◆ does not create a new type
 - ◆ useful for expressing intention
 - ◆ can aid portability too

```
    unsigned int at;  
typedef unsigned int size_t;
```



syntax mirrors a declaration

```
unsigned int at;  
size_t at;
```

equivalent definitions

```
void f(unsigned int variable);  
void f(size_t variable);
```



equivalent declarations
(allowed)

```
void f(unsigned int variable);  
...  
void f(size_t variable)  
{  
    ...  
}
```



declared without typedef
defined with typedef
(allowed)

- an enum definition introduces a new type
 - ◆ and a sequence of enumerators
 - ◆ each enumerator is not scoped to its enum
 - ◆ each enumerator has a constant int value
 - ◆ by default starts at zero and increments by one

```
enum suit  
{  
    clubs, diamonds, hearts, spades  
};  
  
enum suit trumps = clubs;
```

← type name

← enumerators

```
enum season  
{  
    spring=1, summer, autumn, fall=autumn, winter  
};
```

↑
enumerators with the same value are allowed

enum on declarations is just noise

```
enum suit { ... };
enum suit trumps = clubs;
```



better

why use a different tag name?

```
enum suit_tag { ... };
typedef enum suit_tag suit;
suit trumps = clubs;
```



better

```
enum suit { ... };
typedef enum suit suit;
suit trumps = clubs;
```



commonly compressed into this

```
typedef enum suit { ... } suit;
suit trumps = clubs;
```



tag name no longer required

```
typedef enum { ... } suit;
suit trumps = clubs;
```



enum typedef

- a thinly wrapped integer – not type safe
 - ◆ any int value can be converted to an enum
 - ◆ an enum can be converted to an int

```
const char * suit_name(suit s)
{
    switch (s)
    {
        case clubs      : return "clubs";
        case diamonds   : return "diamonds";
        case hearts     : return "hearts";
        case spades     : return "spades";
        default         : return NULL; // can happen
    }
}
```

```
int main(void)
{
    suit trumps = (suit)42;
    int value = (int)trumps;
    printf("%s\n", suit_name(trumps));
}
```

neither cast is required



enum ← → int

- useful for designators

```
enum { january, february, ...  
      november, december };  
  
const int days_in_month[] =  
{  
    [january] = 31,  
    [february] = 28,  
    ...  
    [november] = 30,  
    [december] = 31  
};
```

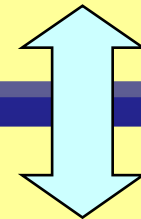
- alternative to #define for array sizes

```
#define MAX_LEN (1024)
```

```
char buffer[MAX_LEN];
```

```
enum { max_len = 1024 };
```

```
char buffer[max_len];
```

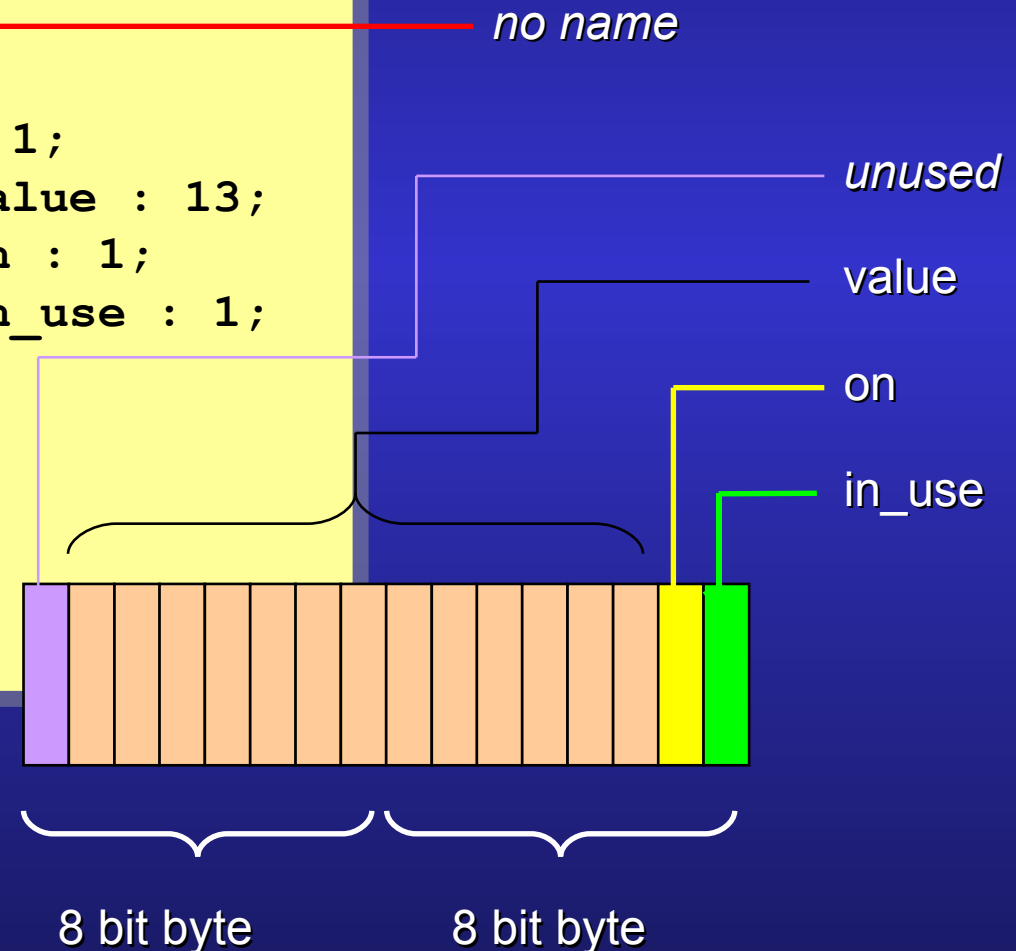


alternatives

- you can use bit-fields to control memory allocation right down to the bit level
 - ◆ machine dependent; not portable

```
typedef struct
{
    unsigned int : 1;
    unsigned int value : 13;
    unsigned int on : 1;
    unsigned int in_use : 1;
} fields;

fields widget;
...
if (widget.in_use)
    ...
```

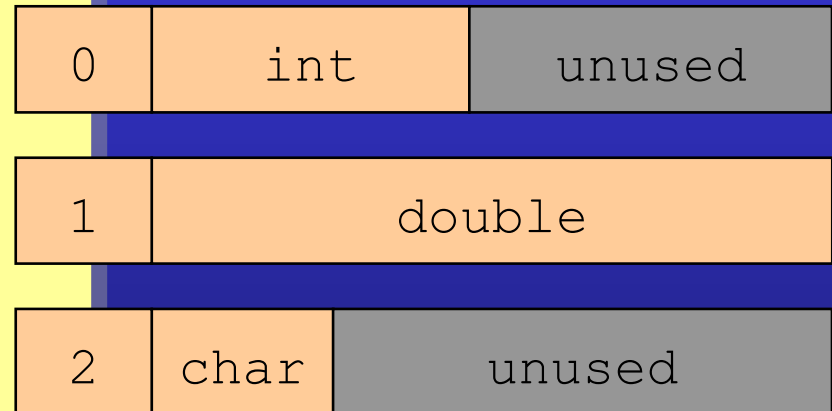


- the members of a union overlay one another
 - ◆ typically used to save memory
 - ◆ often accompanied by a discriminator

```
typedef enum { int_u, double_u, char_u } discrim;
```

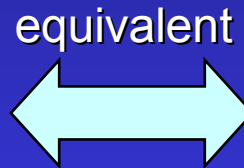
```
typedef union spring_tag  
{  
    int i;  
    double d;  
    char c;  
} spring;
```

```
typedef struct ure_tag  
{  
    discrim is_a;  
    spring value;  
} ure;
```



- a struct definition introduces a new type
 - ◆ aggregation of heterogeneous data members
 - ◆ structs may contain other structs

```
struct date_tag
{
    int year;
    int month;
    int day;
};
```



```
struct date_tag
{
    int year, month, day;
};
```

```
struct project_tag
{
    const char * name;
    struct date_tag deadline;
    ...
};
```

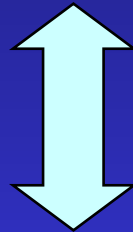
why not use `_t` as a suffix instead of `_tag` ?

struct typedef

struct on declarations is just noise

```
struct date_tag { ... };  
struct date_tag deadline;
```

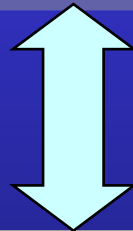
alternatives



```
struct date_tag { ... };  
typedef struct date_tag date;  
date deadline;
```

why use a different tag name?

equivalent



```
typedef struct date_tag { ... } date;  
date deadline;
```

- repeating struct/union in all declarations can create subtle problems
 - ◆ spell the tag name incorrectly and it still compiles
 - ◆ the new tag name introduces a new type

```
struct wibble { ... };  
void func(struct wibble_t *);
```



```
typedef struct { ... } wibble;  
void func(wibble_t *);
```



- types can have alignment restrictions
 - ◆ first struct member determines alignment
 - ◆ padding can be added between struct members and after the last struct member

```
typedef struct point_tag
{
    char x;
    int y;
} point;
```

0xBABC ← char x

0xBABD ← padding

0xBABE ← padding

0xBABF ← int y

0xBAC0 ← int y

0xBAC1 ← int y

0xBAC2 ← int y

0xBAC3 ← padding

0xBABC ← char x

0xBABD ← padding

0xBABE ← padding

0xBABF ← padding

0xBAC0 ← int y

0xBAC1 ← int y

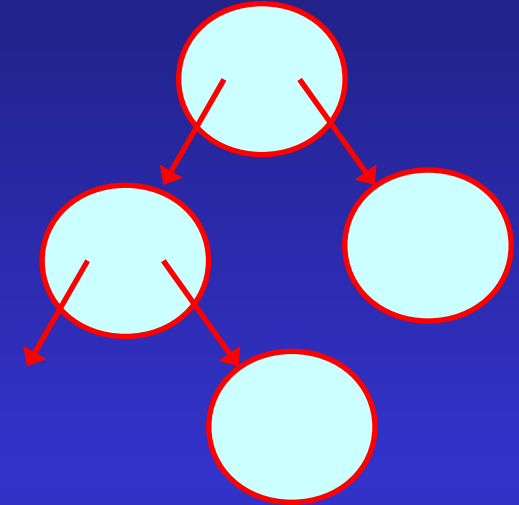
0xBAC2 ← int y

0xBAC3 ← int y

0xBAC4 ← padding

- a struct S member can be of type struct S *

```
struct tree_node
{
    int count;
    struct tree_node * left;
    struct tree_node * right;
};
```



- a struct S member can't be of type struct S

```
struct infinite
{
    struct infinite descent;
};
```




compile-time error


TODO: drawing

- **structs support a convenient initialisation**
 - ◆ allows const struct variables
 - ◆ not permitted for assignment
 - ◆ list cannot be empty
 - ◆ missing fields are default initialised

```
date deadline = { 2008, may, 1 }; 
```

```
const project fubar =  
    { "fubar", { 2008, may, 1 } }; 
```

```
date deadline;  
deadline = { 2008, may, 1 }; 
```

```
date deadline = { }; 
```

- the dot operator accesses struct members
 - ◆ left associative: $a.b.c \rightarrow (a.b).c$
 - ◆ initialisation/assignment from another struct

```
struct date_tag
{
    int year;
    int month;
    int day;
};
typedef struct date_tag date;
```

```
struct project_tag
{
    const char * name;
    date deadline;
    ...
};
typedef struct project_tag project;
```

```
date deadline;
deadline.year = 2008;
deadline.month = may;
deadline.day = 1;
```

```
project fubar;
fubar.name = "fubar";
fubar.deadline = deadline;
fubar.deadline.year++;
```

simple bitwise copy

- **structs support designator identifiers**
 - ◆ allows list elements to be reordered
 - ◆ missing members are default initialised

```
typedef struct date_tag
{
    int year;
    int month;
    int day;
} date;
```

```
date deadline = { .day = 1, .month = may, .year = 2008 };
```

equivalent

```
date deadline = { .month = may, .day = 1, .year = 2008 };
```

- **aggregate initialization list can be cast**
 - ◆ **cast type becomes type of expression**
 - ◆ **assignment works with the cast**

```
deadline =  
    (date){ 2008, may, 1 };
```

cast works for plain initialiser lists

```
deadline =  
    (date){ .year = 2008, .month = may, .day = 1 };
```

cast works for designators

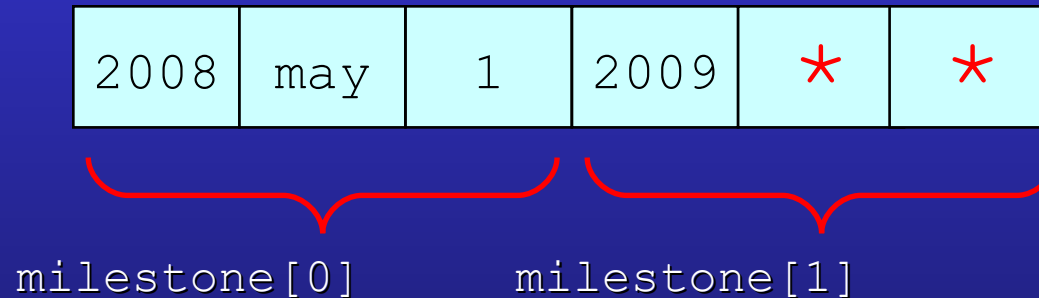
```
call((date){ .year = 2008, .month = may, .day = 1 });  
call((const date){ 2008, may, 1 });
```

also allows anonymous objects to be created

- arrays and struct may contain each other
 - ◆ [int] and .identifier designators can be combined

array of structs

```
date milestones[] =  
  {  
    [0] = { 2008, may, 1 },  
    [1].year = 2009  
  };
```



* default value

- last member may be incomplete array type
 - ◆ can't be a member of a struct
 - ◆ can't be an element in an array

```
typedef struct hack_tag
{
    size_t count;
    char message[];
} hack;
```

← incomplete array type

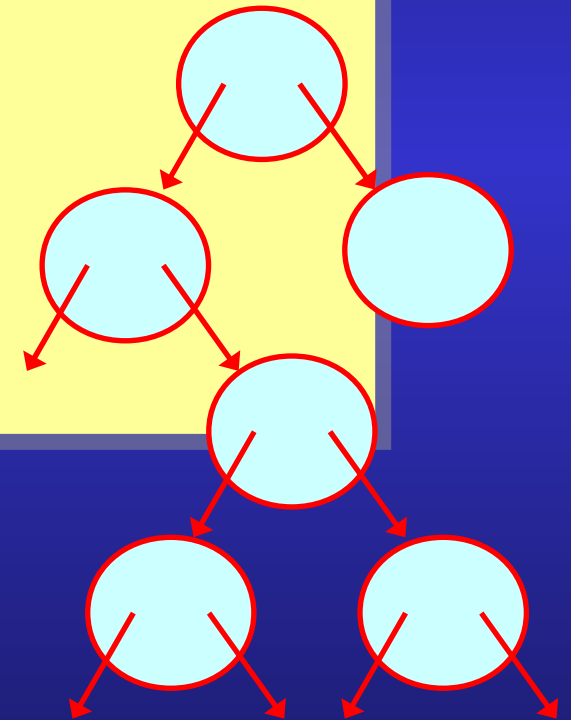
```
size_t n = get_size();
hack * ptr = malloc(sizeof(*ptr) + n);

ptr->count = n;
strcpy(ptr->message, "Hello world");
```



- the restrict pointer modifier can be used on struct pointer members
 - ◆ enables the same no-alias optimizations

```
typedef struct tree_node_tag tree_node;  
  
struct tree_node_tag  
{  
    int count;  
    tree_node * restrict left;  
    tree_node * restrict right;  
};
```



- $p \rightarrow m$ is an idiomatic shorthand for $(*p).m$
 - ◆ arrow has same precedence as dot
 - ◆ associates left to right: $p \rightarrow q \rightarrow r == (p \rightarrow q) \rightarrow r$

```
date deadline = { 2008, may, 1 };  
date * ptr = &deadline;
```

```
*ptr.year = 2008;
```

 compile-time error

```
(*ptr).year = 2008;
```

 non-idiomatic

```
ptr->year = 2008;
```

 idiomatic

- **creating new types is important**
- **typedef does not create a new type**
- **enums are thinly wrapped integers**
- **bitfields**
- **unions for saving memory or expressing mutually exclusive possibilities**
- **structs are the most common**
- **structs have a rich aggregate list syntax**
- **struct hack allows variable length struct**
- **struct dot operator for values**
- **struct arrow operator for pointers**
- **structs support assignment but not equality**

- This course was written by

Expertise: Agility, Process, OO, Patterns
Training+Designing+Consulting+Mentoring

{ JSL }

Jon Jagger

jon@jaggersoft.com

www.jaggersoft.com