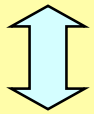
Structures

typedef

- 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 declaration

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

equivalent definitions
(compile time error)

```
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)

enum

- 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 integer value
  - by default starts at zero and increments by one

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

enum suit trumps = clubs;
```

type name

enumerators

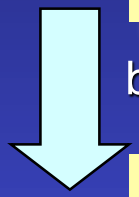
```
enum season
{
    spring=1, summer, autumn, fall=autumn, winter
};
```

enumerators with the same value are allowed

enum typedef

enum on declarations is just noise?

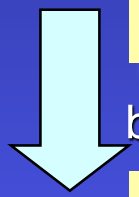
```
enum suit { ... };
enum suit trumps = clubs;
```



better

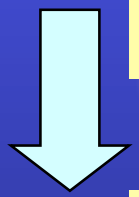
don't use a different tag name

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



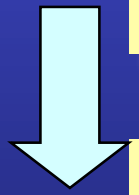
better

```
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 → int

- 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

?

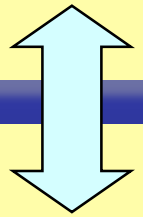
anonymous enums

- 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];
```



alternatives

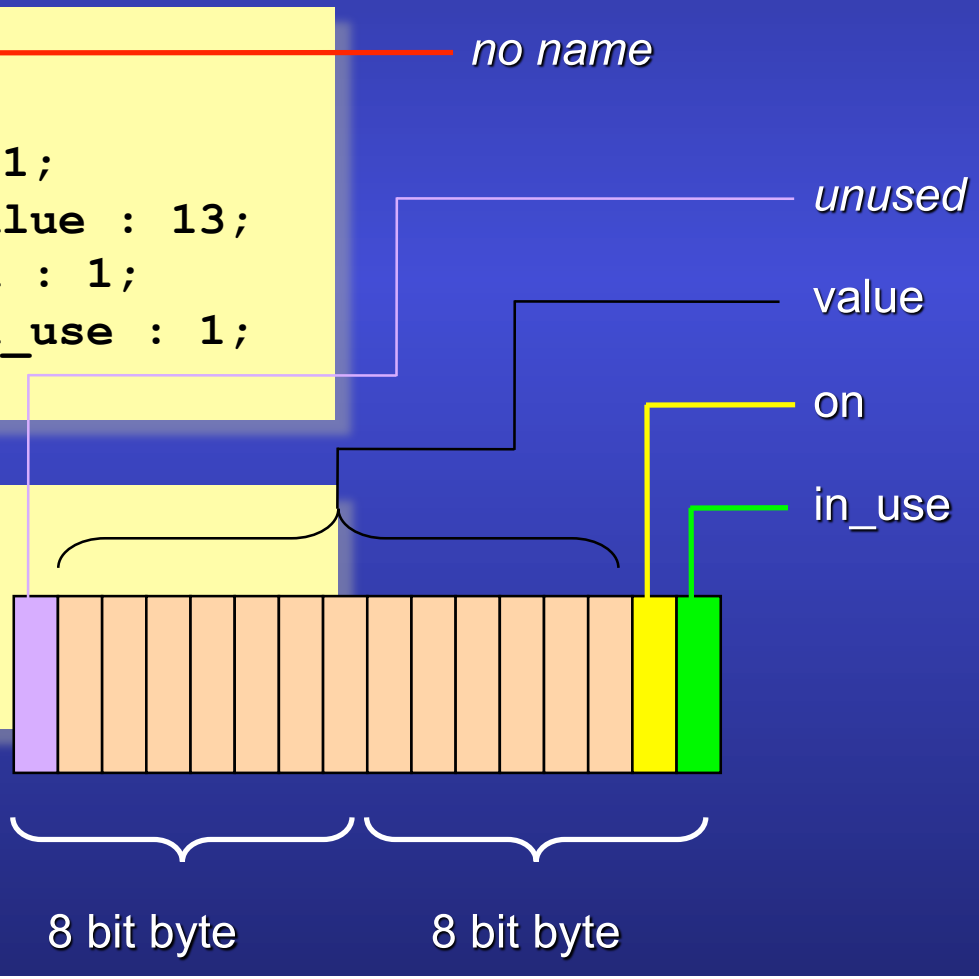
```
enum { max_len = 1024 };

char buffer[max_len];
```

bit fields

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

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

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

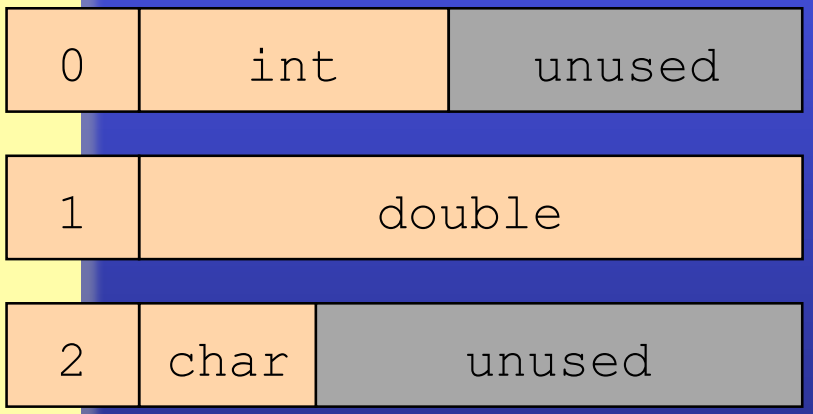


unions

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

```
enum descrim {  int_u, double_u, char_u };
```

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

struct ure
{
     enum descrim is_a;
     union spring value;
};
```



structs

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

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

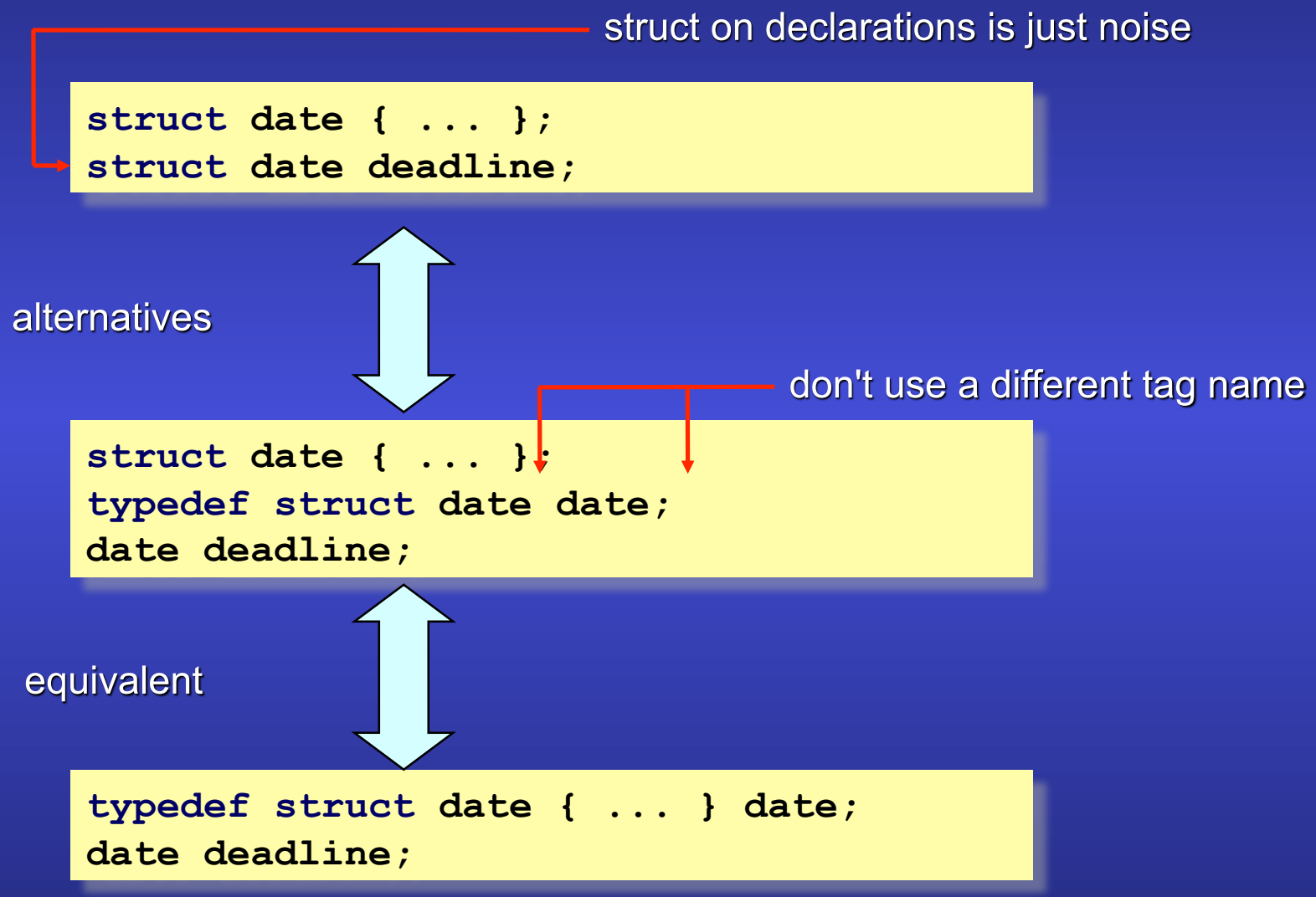
equivalent

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

?

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

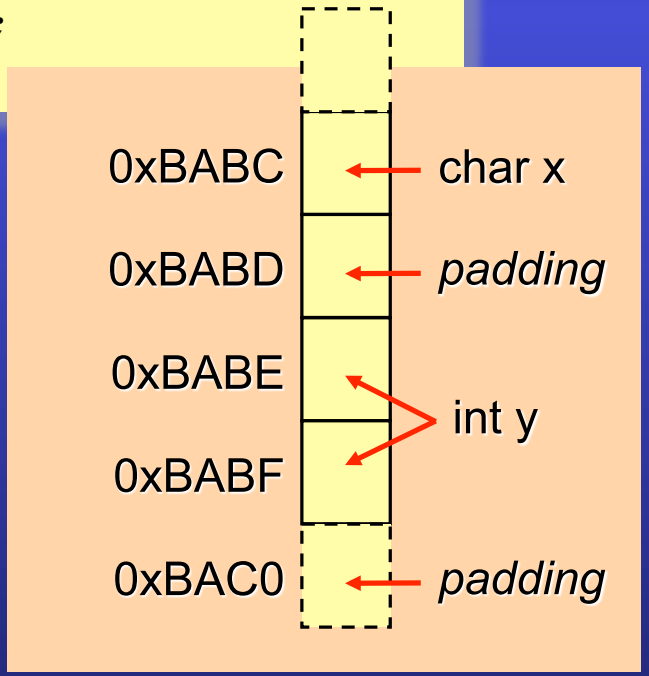
struct typedef

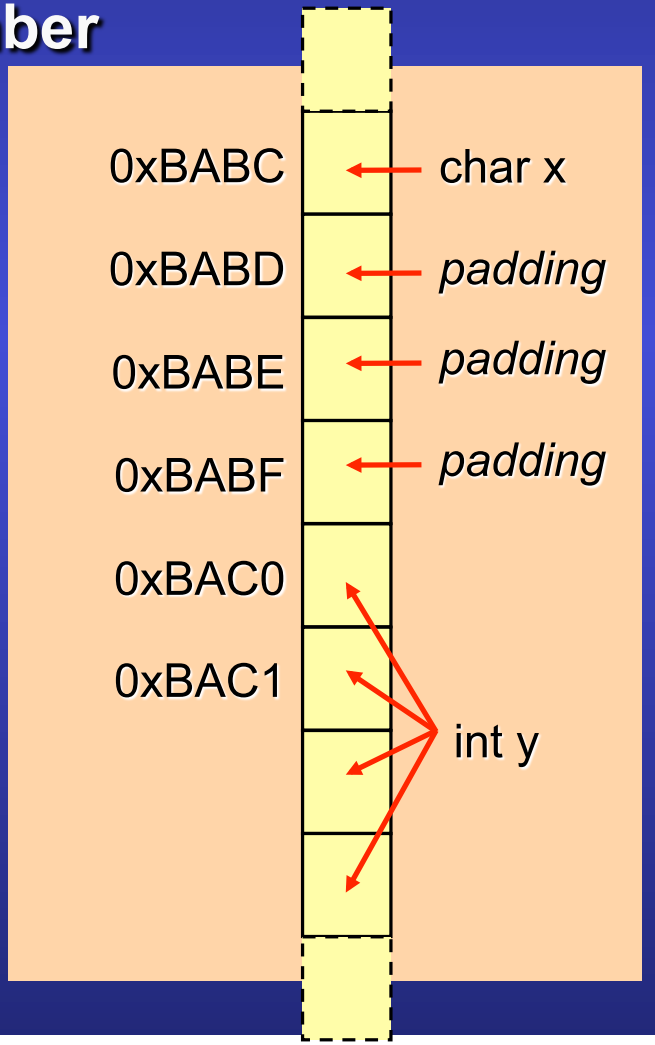


alignment

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

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

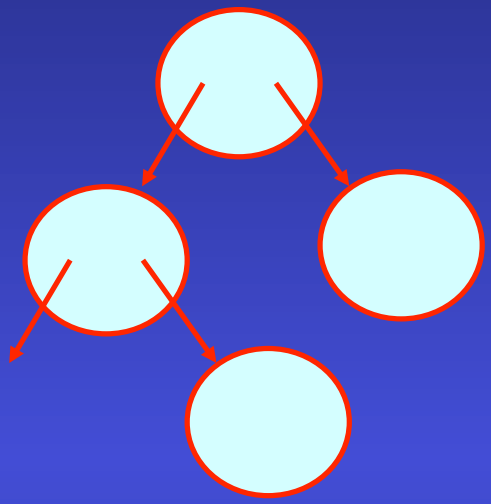




recursive structs

- 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

initialization

- 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 = { };
```

dot operator

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

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

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

simple bitwise copy

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

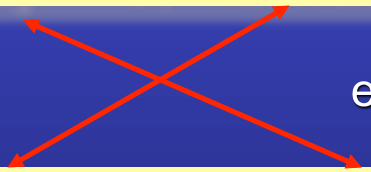
```
date deadline;
deadline.year = 2008;
deadline.month = may;
deadline.day = 1;
```

designators

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

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

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



equivalent

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

compound literals

- 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

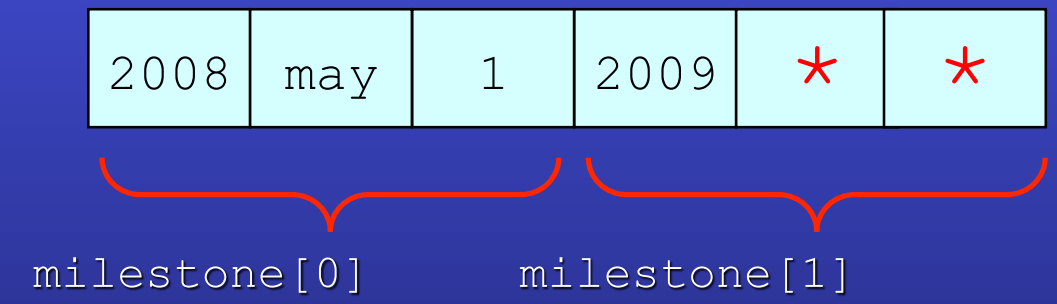
```
call((date){ .year = 2008, .month = may, .day = 1 });

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

struct hack

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

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

incomplete array type

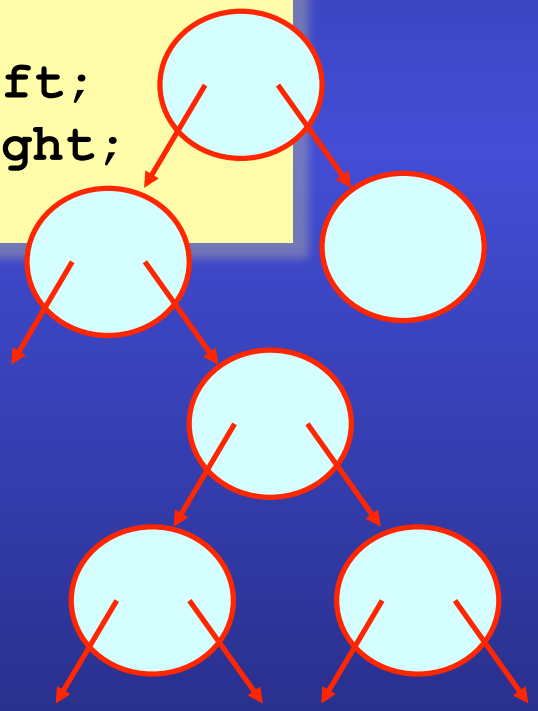
```
size_t n = get_size();
struct hack * ptr = malloc(sizeof(*ptr) + n);

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

?

restrict

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

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



arrow operator

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

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

date * ptr = &deadline;
```

```
*ptr.year = 2008;

(*ptr).year = 2008;

ptr->year = 2008;
```

✘ compile-time error

?? non-idiomatic

✓ idiomatic

summary

- 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