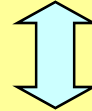
Structures

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)

- 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;
```

- 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

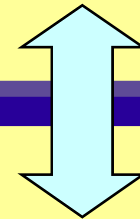
- useful for designators

```
enum { january, february,  ...
       november, december };

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

c99

- alternative to #define for array sizes

```
#define MAX_LEN (1024)

char buffer[MAX_LEN];
```



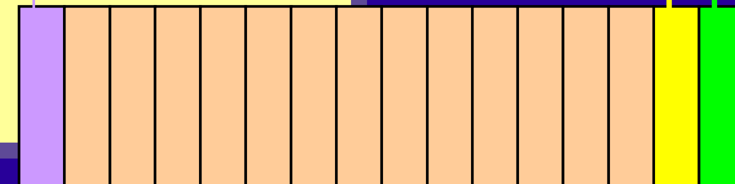
alternatives

```
enum { max_len = 1024 };

char buffer[max_len];
```

- you can use bitfields to control memory allocation right down to the bit level
  - compiler 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)
    ...
```

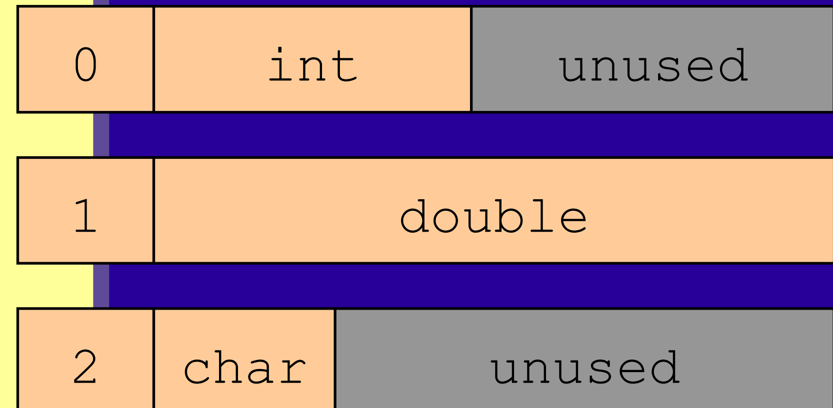


unions

- 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;
};
```



- 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;
};
```

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

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

struct typedef

struct on declarations is "noise"?

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

alternatives

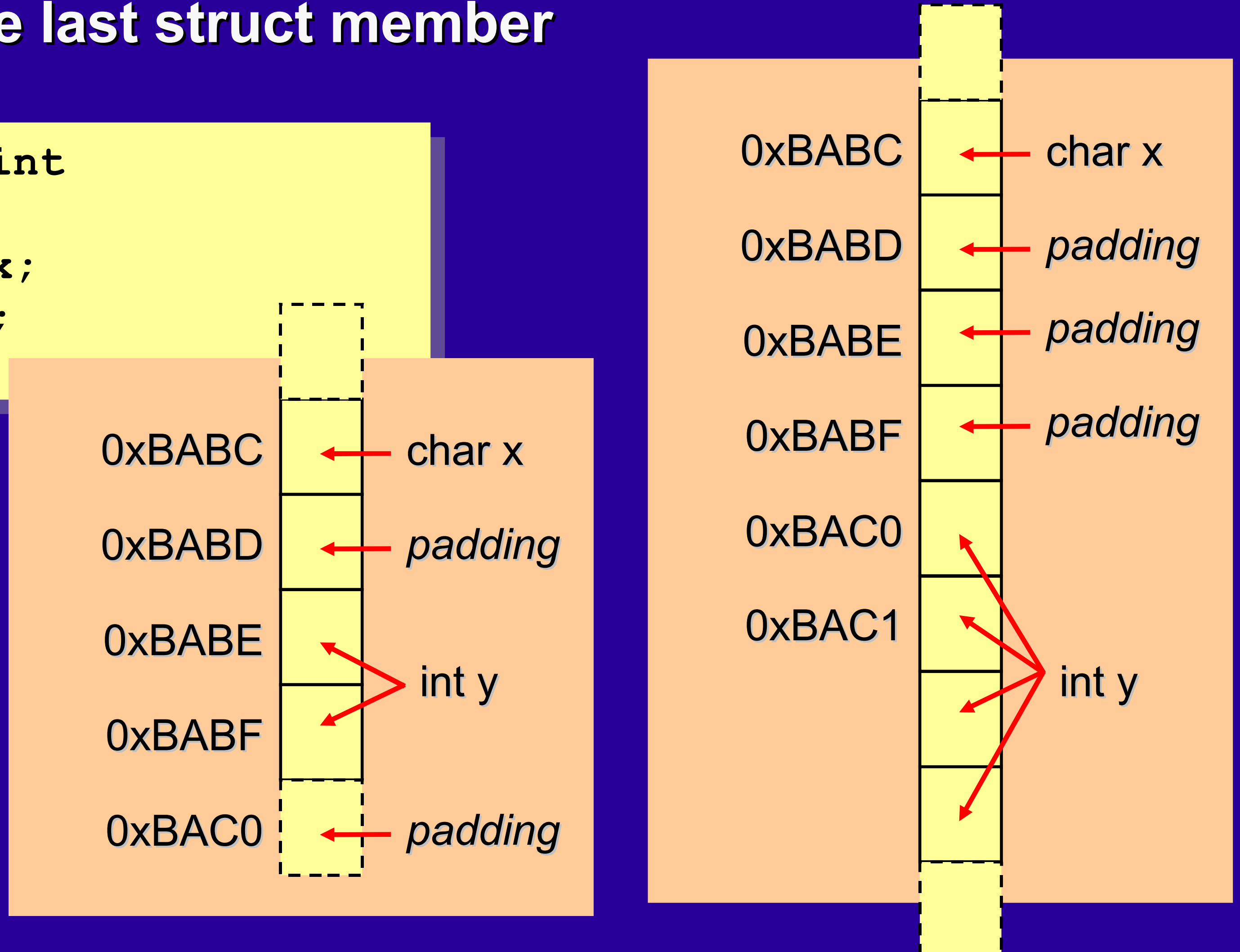
use the same tag name

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

equivalent

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

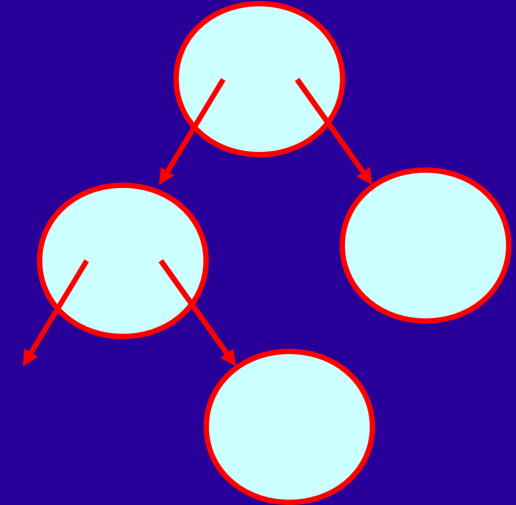
- types can have alignment restrictions
  - first struct member determines alignment
  - padding can be added between struct members and after the last struct member



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

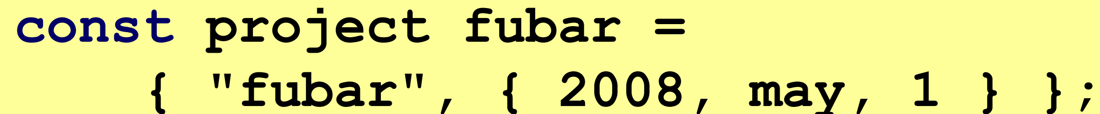
```
struct infinite
{
    struct infinite descent;
};
```

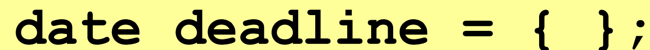
compile-time error

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





dot operator

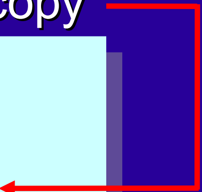
- the dot operator accesses struct members
  - left associative: a.b.c → (a.b).c
  - initialisation/assignment from another struct

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

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

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

simple bitwise copy



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

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

c99

```
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"

c99

```
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 });
```

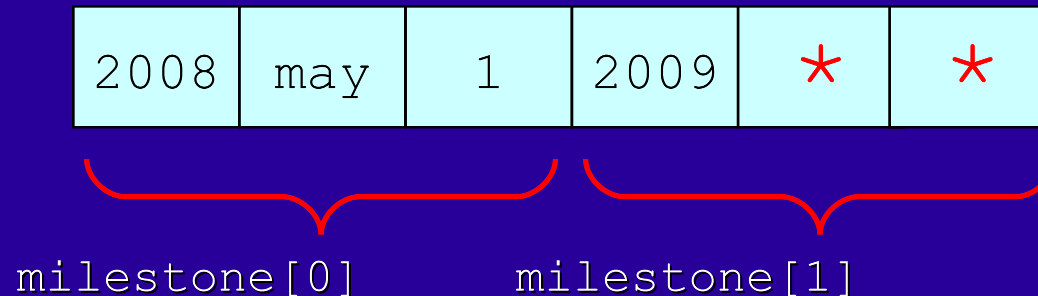
also allows anonymous objects to be created

- 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
    };
```

c99



* 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

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

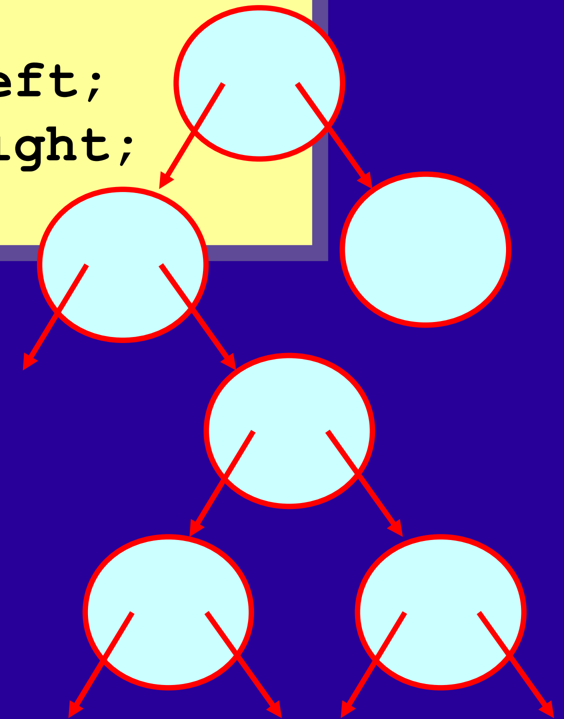
incomplete array type — c99

```
size_t n = get_size();
struct 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**

c99

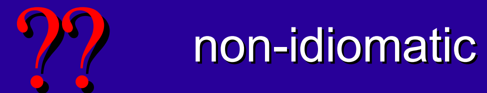
```
struct tree_node
{
    int count;
    struct tree_node * restrict left;
    struct tree_node * restrict right;
};
```



- **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**

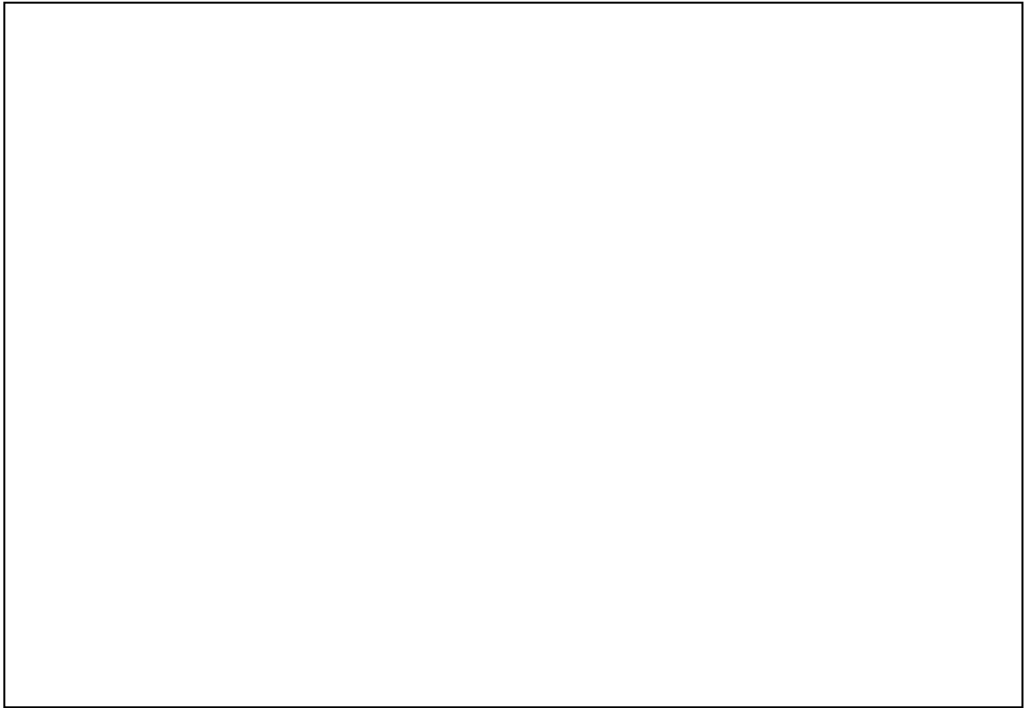
```
date deadline = { 2008, may, 1 };

date * ptr = &deadline;
```

```
*ptr.year = 2008;

(*ptr).year = 2008;

ptr->year = 2008;
```



summary

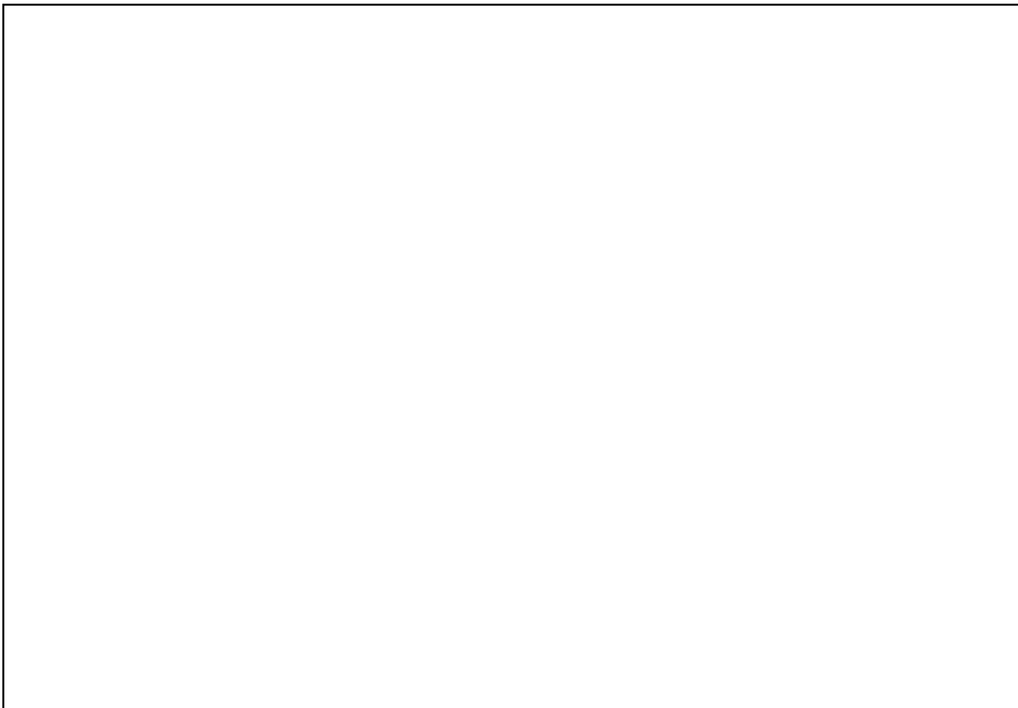
- creating new types is important
- typedef does not create a new type
- enums are thinly wrapped integers
- bitfields
- unions for saving memory or expressing mutually exclusive possibilities
- structs are the most common
- structs have a rich aggregate list syntax
- struct hack allows variable length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



Note that a duplicate typedef is an error in C.

```
typedef unsigned int size_t;
typedef unsigned int size_t; // error
```

A duplicate typedef such as this is allowed in C++ though.



An enumerator is not scoped to its enum. This means two enumerators in the same scope must be distinct from each other and from all other ordinary identifiers declared in that scope.

```
enum suit { … diamonds … };
enum precious_stones { … diamonds … }; // compile time error
int diamonds; // compile-time error
```

You can't forward declare an enum.

A trailing comma is permitted in the enumerator list (it isn’t in C++):

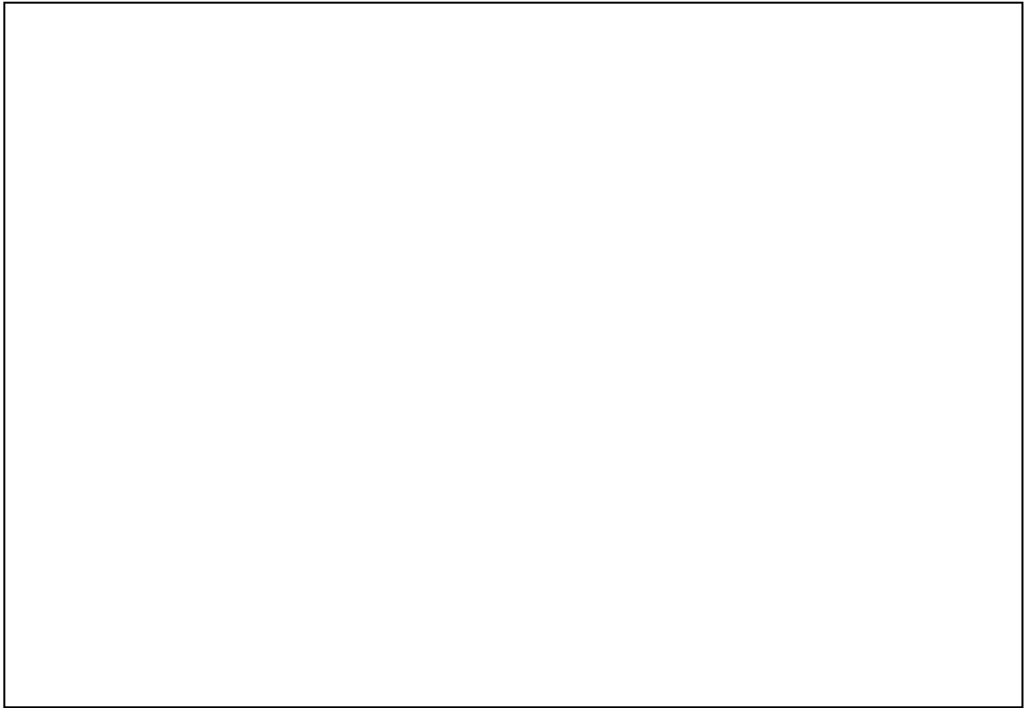
```
enum suit { clubs, diamonds, hearts, spades, };
```

This is useful mainly to generated code.

The names of enumeration constants (such as clubs) are not externally visible in C. If you build a C program from two source files, one containing the identifier clubs as an enumeration constant and the other containing the identifier clubs as the name of a function (and the latter does not #include the former) then the build will succeed; it will not fail at link time because of a duplication definition since only the name of the function is externally visible to the linker. It may fail if built in C++ though.

The int → enum cast is required in C++.

The C standard does not guarantee that the underlying integer type of the enum will be int.

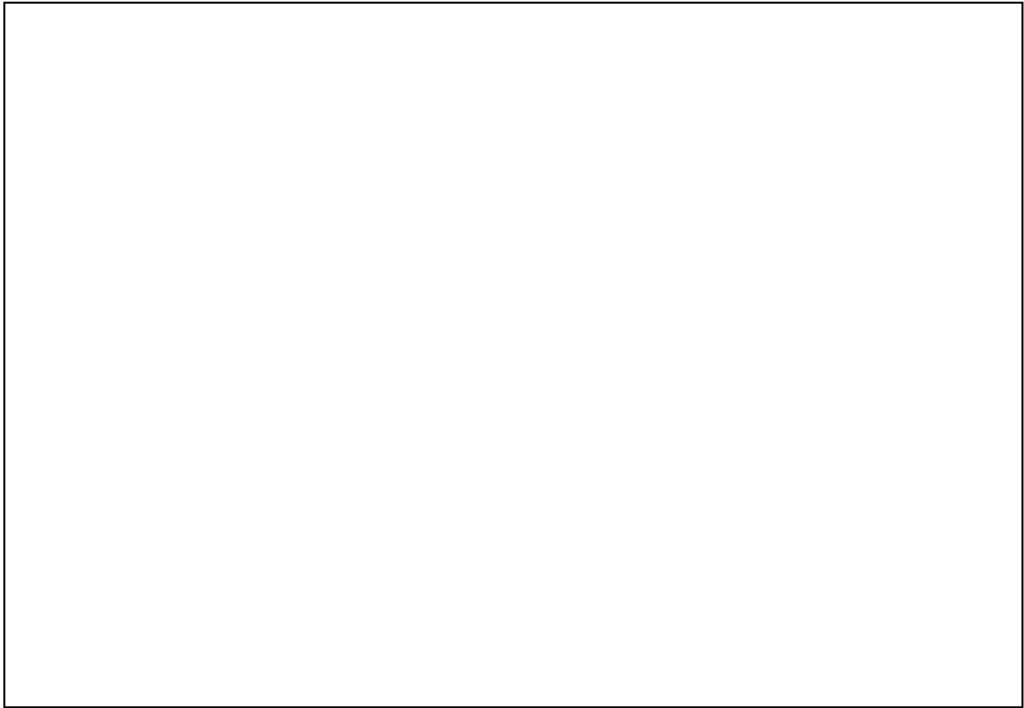


The size of a bit field has to be declared as a compile time constant value.

A bit field can be unnamed, which indicates an unused number of bits used solely to align the following bit field to a specific bit.

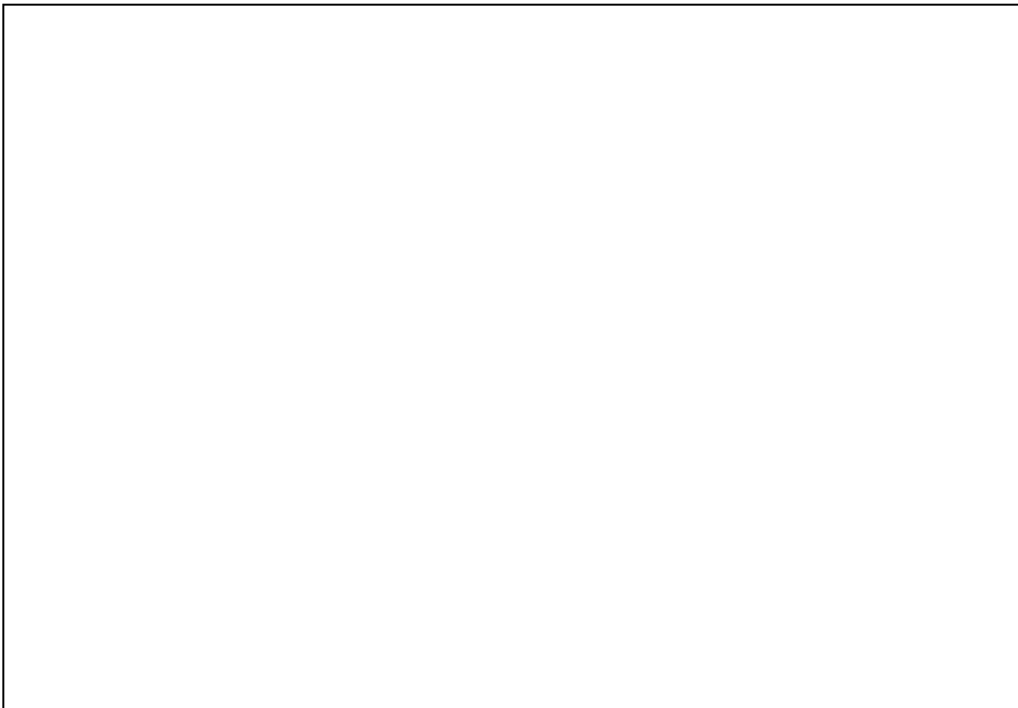
A bit field cannot have its address taken.

Bit fields are not portable. For example, they have no effect on the endianess of the machine.



The two date structs are syntactically identical. However, you can choose to use them to expresses the ideas that year, month, and day are coincidentally the same type (by writing each one in its own declaration), or that year, month, and day are necessarily of the same type (by writing all three in the same declaration). This is quite subtle though and the former is probably preferred.

One reason for not using _t as a suffix is that POSIX reserves _t as a suffix.



One reason for using a different tag name is to maintain C++ compatibility. For example, in C++ the following will not compile:

```
typedef struct date * date;
```

However, this is not particularly compelling and should be treated as a counterexample rather than an example! As detailed later in the course, this form of `typedef` (with a hidden pointer) is not recommended.

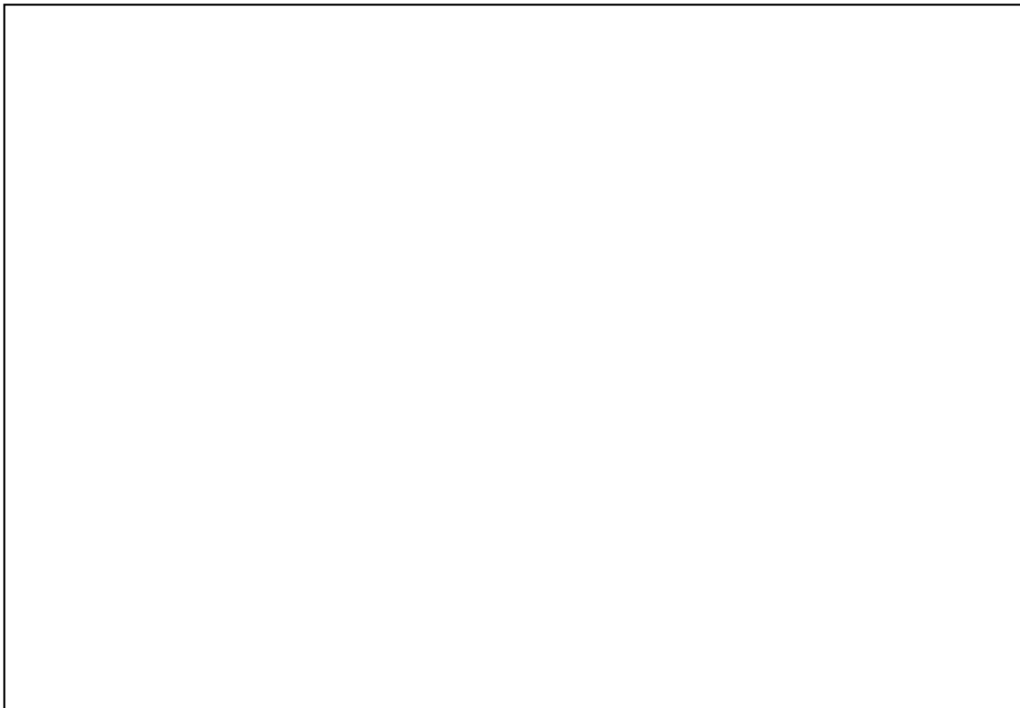
So, looked at from another point of view, a strong case for using the same tag name is for C++ compatibility and simplicity. In C++ `struct date` allows both `date` and `struct date` to be used as names.

And perhaps the simplest reason for not introducing a different tag name is simplicity: avoid gratuitous and needless differences. Don't adopt coding conventions that (1) highlight things that need not be highlighted and (2) solve imagined rather than real problems.

You can use the offsetof macro in <stdlib.h> to find the offset (in bytes) of a struct member from the start of its struct.

Note that you are guaranteed that struct fields are aligned in memory in the order declared and at increasing addresses:

```
point p;
assert(&p.x < &p.y);
```



An alternative arrangement of the top code fragment is as follows:

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

Before C99 initialisers for auto aggregates were required to be constant expressions.

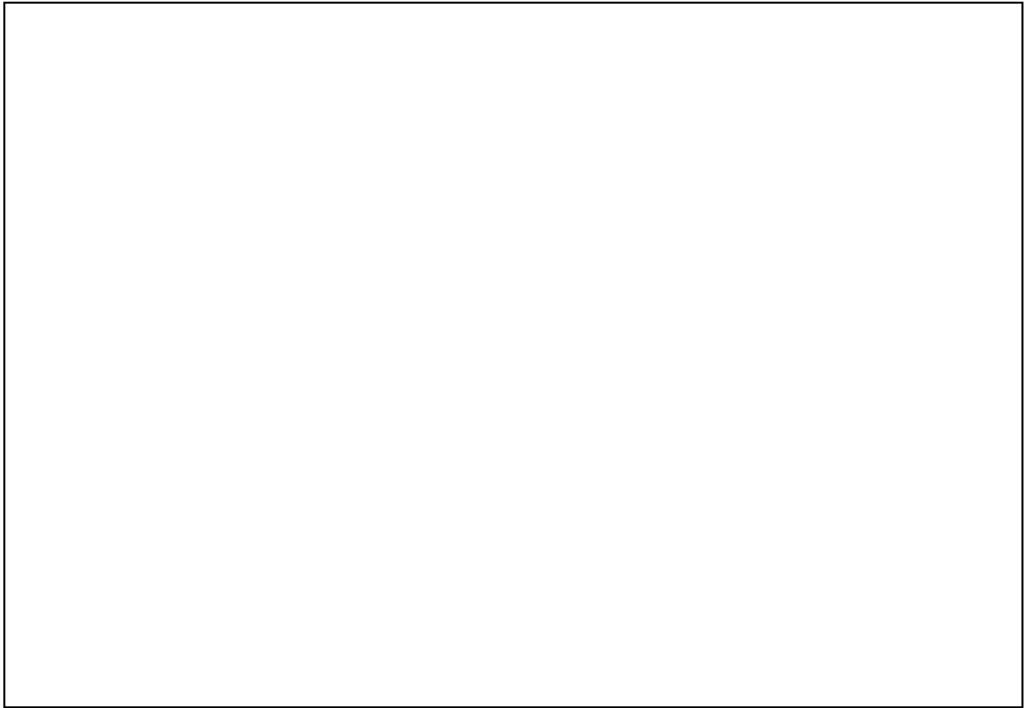
A trailing comma is allowed in the initializer list (a new feature of C99):

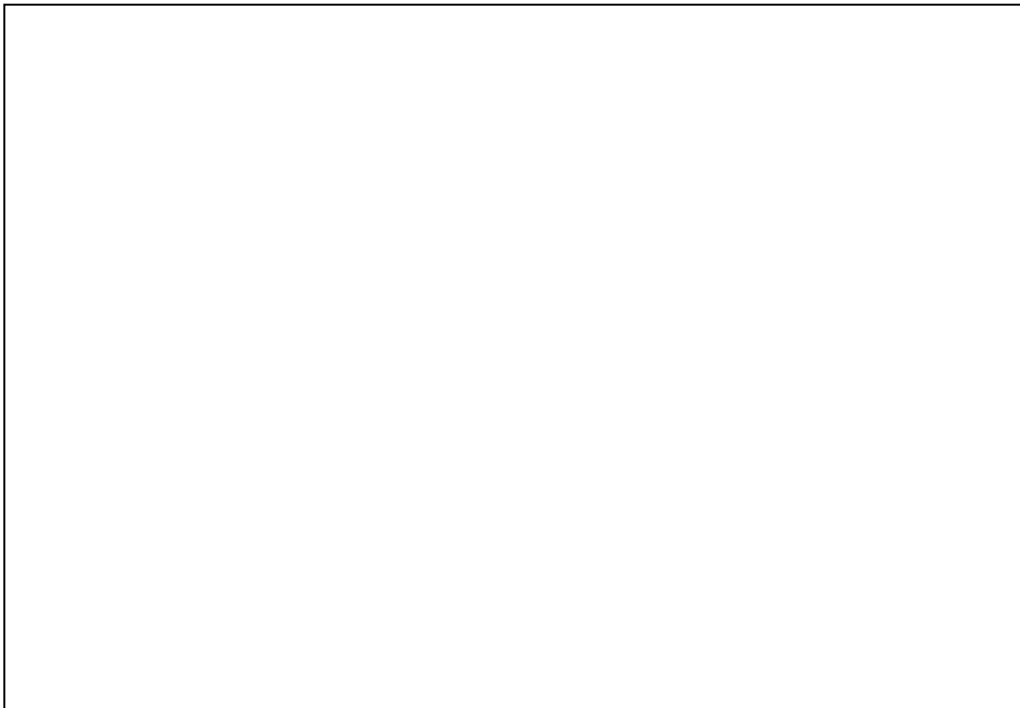
```
date deadline = { 2008, may, 1, };
```

Again, this is only really useful for machine generated code.

An empty initializer list is *not* allowed (it is in C++).

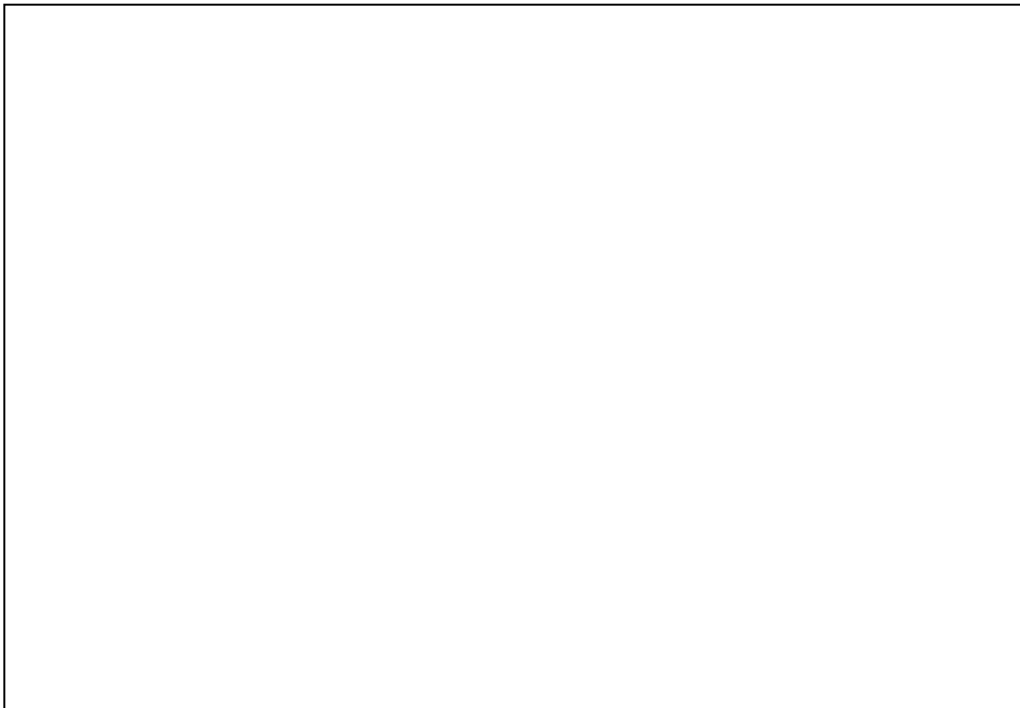
```
date deadline = { }; // constraint violation
```





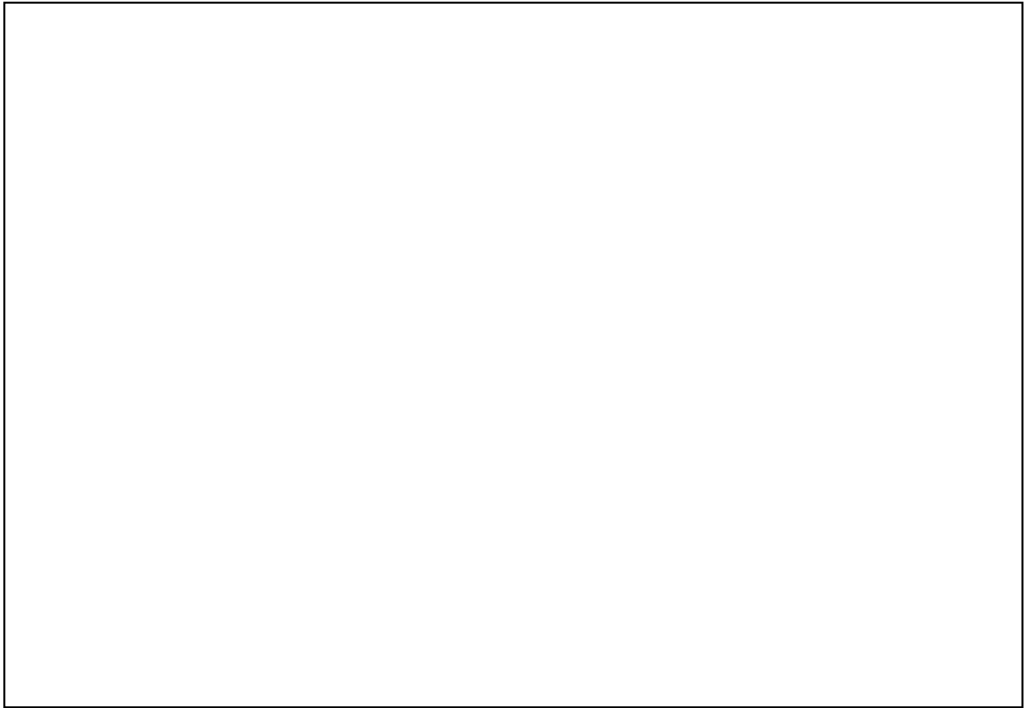
This syntax was added in C99.

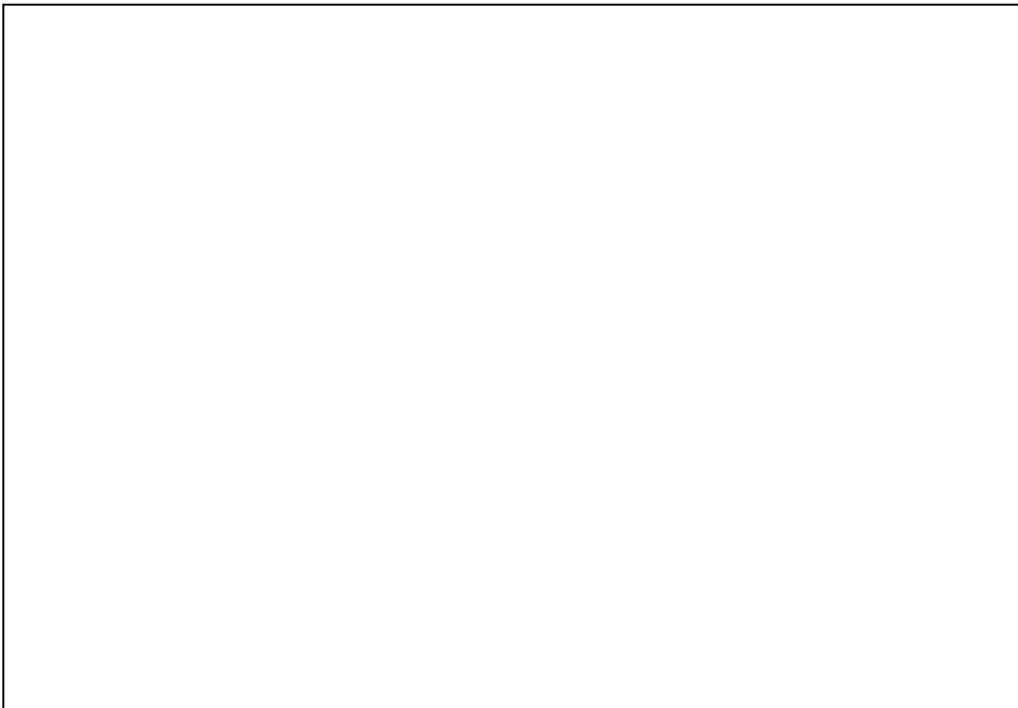
Designated initialisation occurs in the initialisation list order and not in the order the named fields are declared in the struct.



Once again members may be omitted and will be default initialized:

```
deadline = (date){ .month=may }; // year = 0, day = 0

deadline = (date){2008}; // month = 0, day = 0
```





This is a new feature in C99. It is commonly known as the struct-hack but it is now officially part of the language.

The incomplete array member cannot be the only member of its struct.

```
struct wont_work
{
   char message[];
};
```