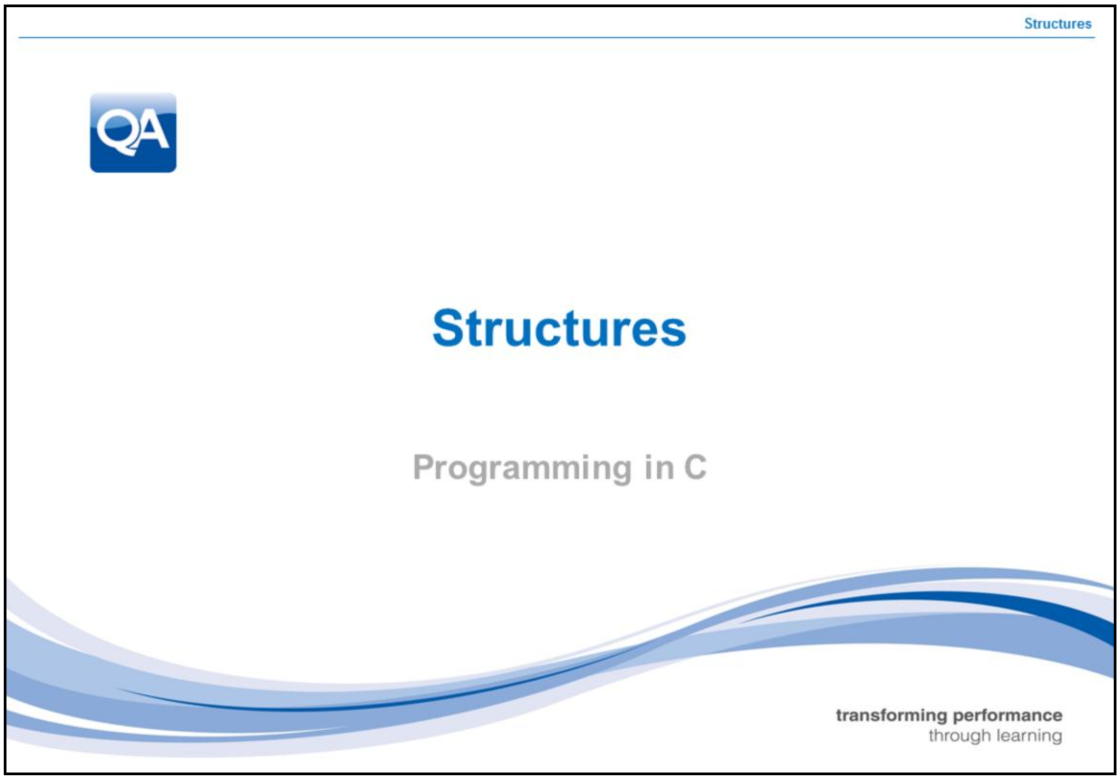
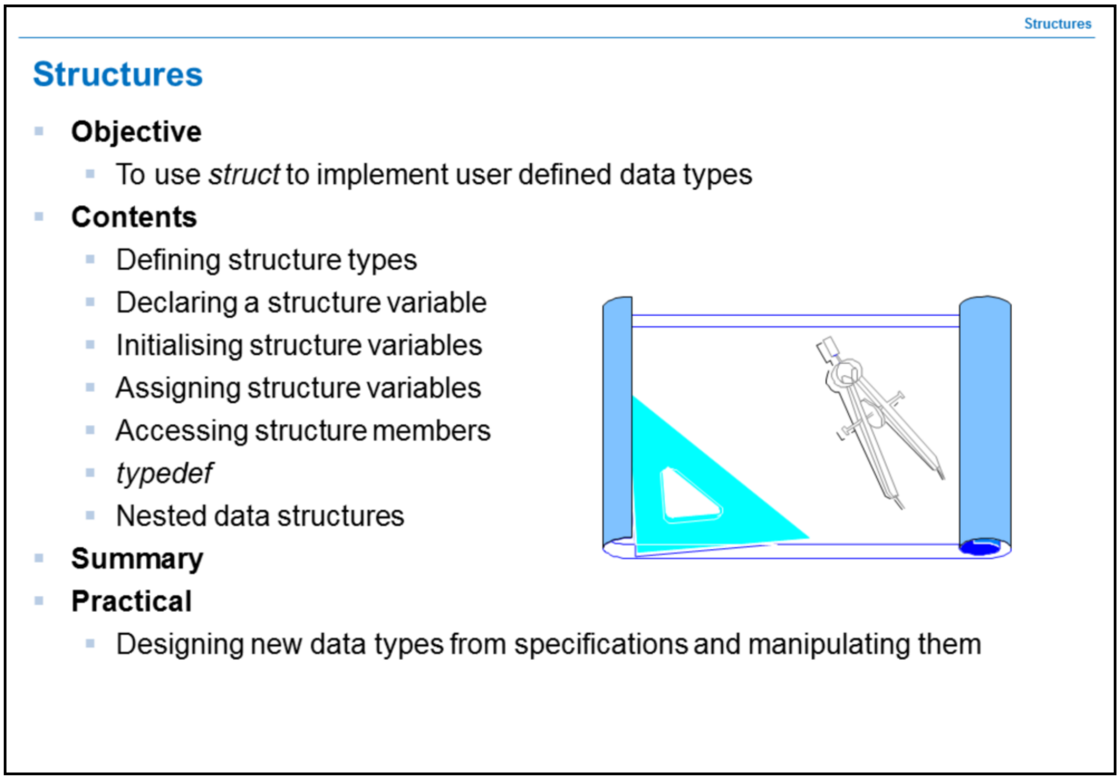


Structures

Structures

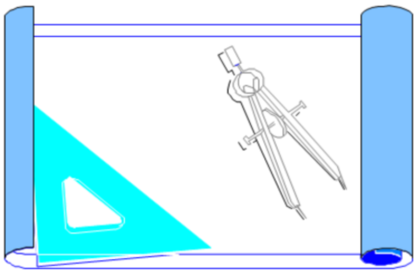
Programming in C

transforming performance
through learning

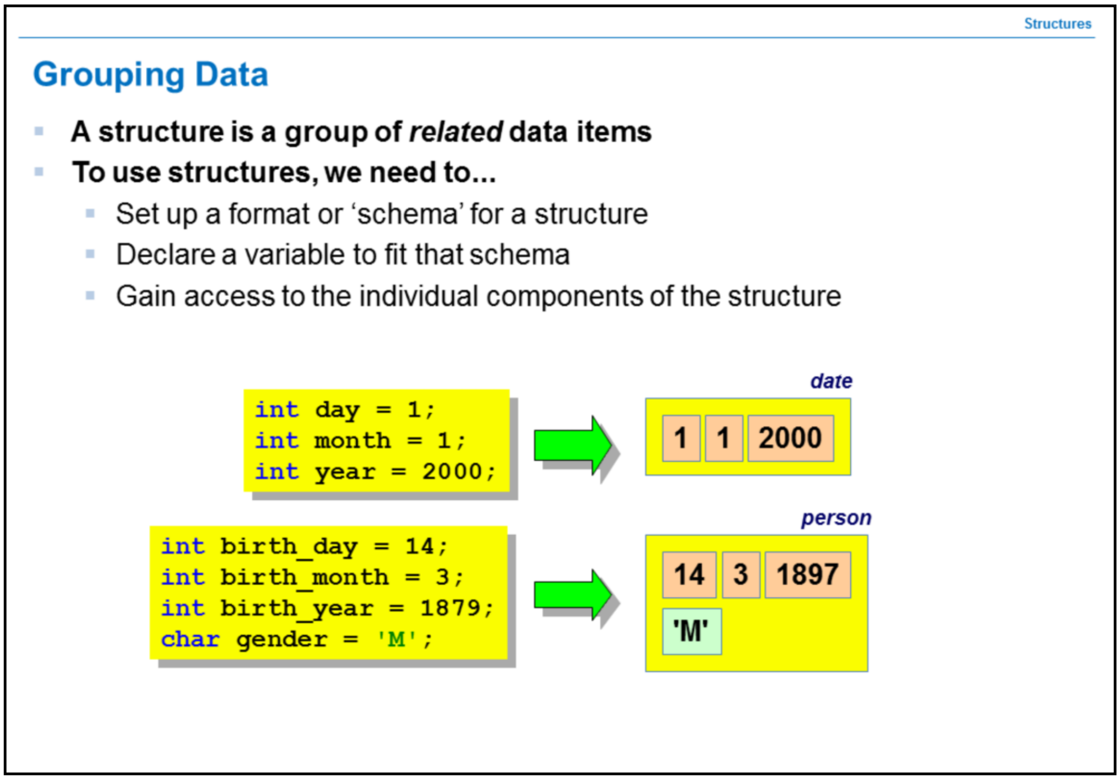


Structures

- **Objective**
  - To use *struct* to implement user defined data types
- **Contents**
  - Defining structure types
  - Declaring a structure variable
  - Initialising structure variables
  - Assigning structure variables
  - Accessing structure members
  - *typedef*
  - Nested data structures
- **Summary**
- **Practical**
  - Designing new data types from specifications and manipulating them



This is the second of the two chapters covering the C's compound (or aggregate) data types. The objective of this chapter is to enable the programmer the freedom to represent and implement the more complicated and flexible data in the specification. Arrays only allow the definition of groups and sequences of like data, i.e. tables, vectors, matrices, etc. The struct takes the idea further.



Structures

Grouping Data

- **A structure is a group of *related* data items**
- **To use structures, we need to...**
  - Set up a format or 'schema' for a structure
  - Declare a variable to fit that schema
  - Gain access to the individual components of the structure

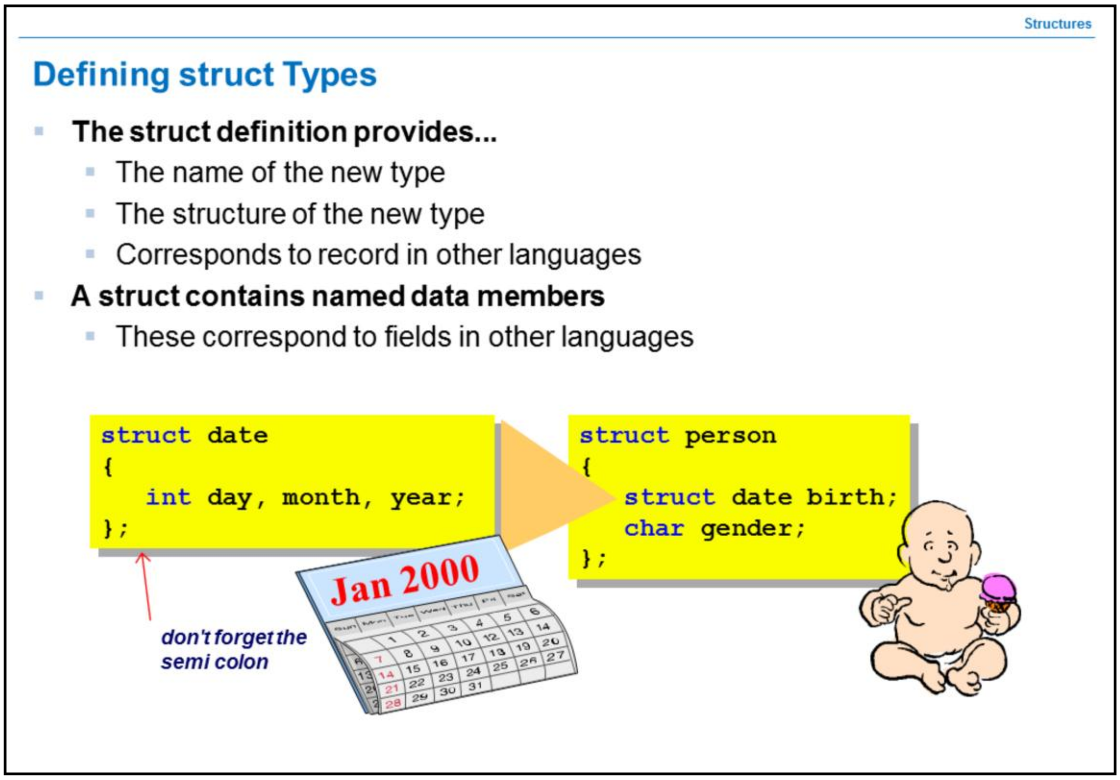
```
int day = 1;
int month = 1;
int year = 2000;
```

```
int birth_day = 14;
int birth_month = 3;
int birth_year = 1879;
char gender = 'M';
```

date
1
1
2000
person
14
3
1897
'M'

The structure, like the array, is a data item that contains a collection of logically-connected data items called members, components or elements. Unlike the array, the members can be of differing type and are not ordered sequentially; they are accessed by name, not index.

Because this aggregate has a richer structure, there is a step that the programmer needs to perform before declaring variables is possible. The shape of the structure needs to be defined. The template, as it is called, informs the compiler of the data type of the members and their names. The other two stages, declaration of variables and accessing the data members, are equivalent to those of the array.



Structures

Defining struct Types

- **The struct definition provides...**
  - The name of the new type
  - The structure of the new type
  - Corresponds to record in other languages
- **A struct contains named data members**
  - These correspond to fields in other languages

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

```
struct person
{
    struct date birth;
    char gender;
};
```

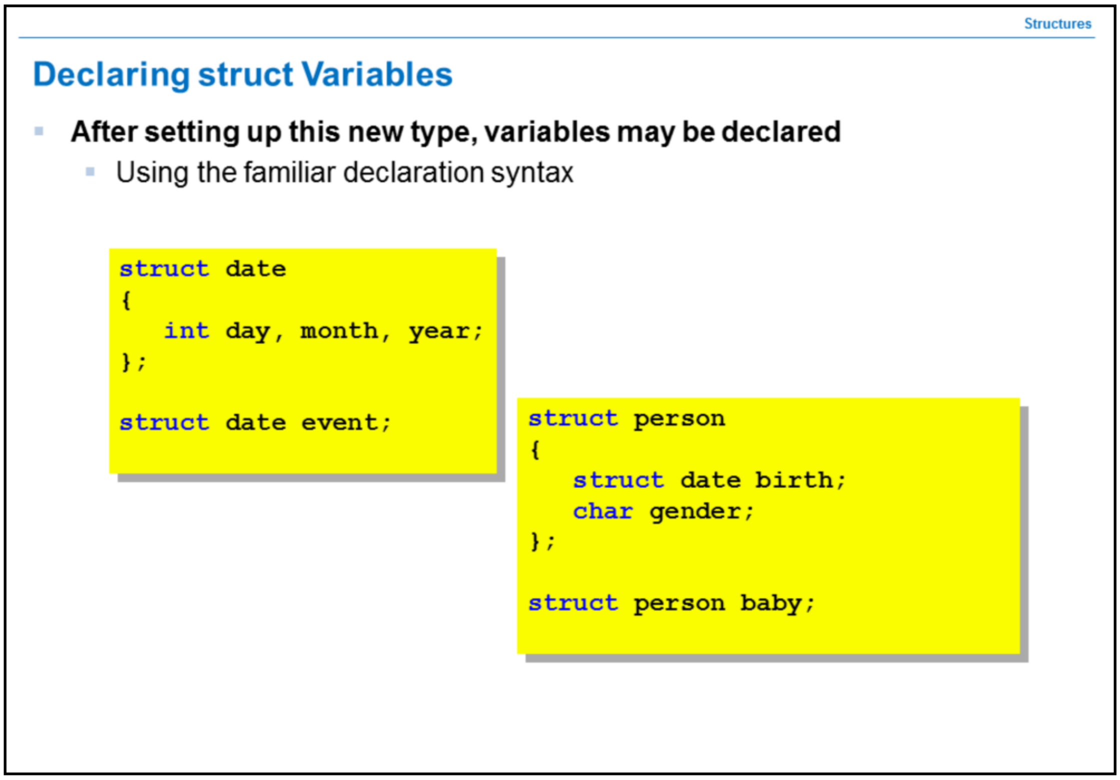
don't forget the semi colon
Jan 2000

Data is often grouped together, for instance a point has an x and a y coordinate. It is more convenient to have a type to represent such grouping than to deal with the attributes separately, e.g. dates can be represented as three integers, but it would be inconvenient to pass these around separately. The content of a struct consists of ordinary declarations, so a date type could be defined as above, or as follows:

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

In C we can define a new type to implement this idea. The `struct` keyword is followed by an identifier, which is called the structure type or tag name. A `struct` definition is a statement so there is a terminating semi-colon.

The type name is then used to declare variables of that type. `struct` - short for *`structure`* - is used for defining such groupings in C. `struct` definitions normally live in header files as they are typically used by many other source files in a system.



Declaring struct Variables

- **After setting up this new type, variables may be declared**
  - Using the familiar declaration syntax

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

struct date event;
```

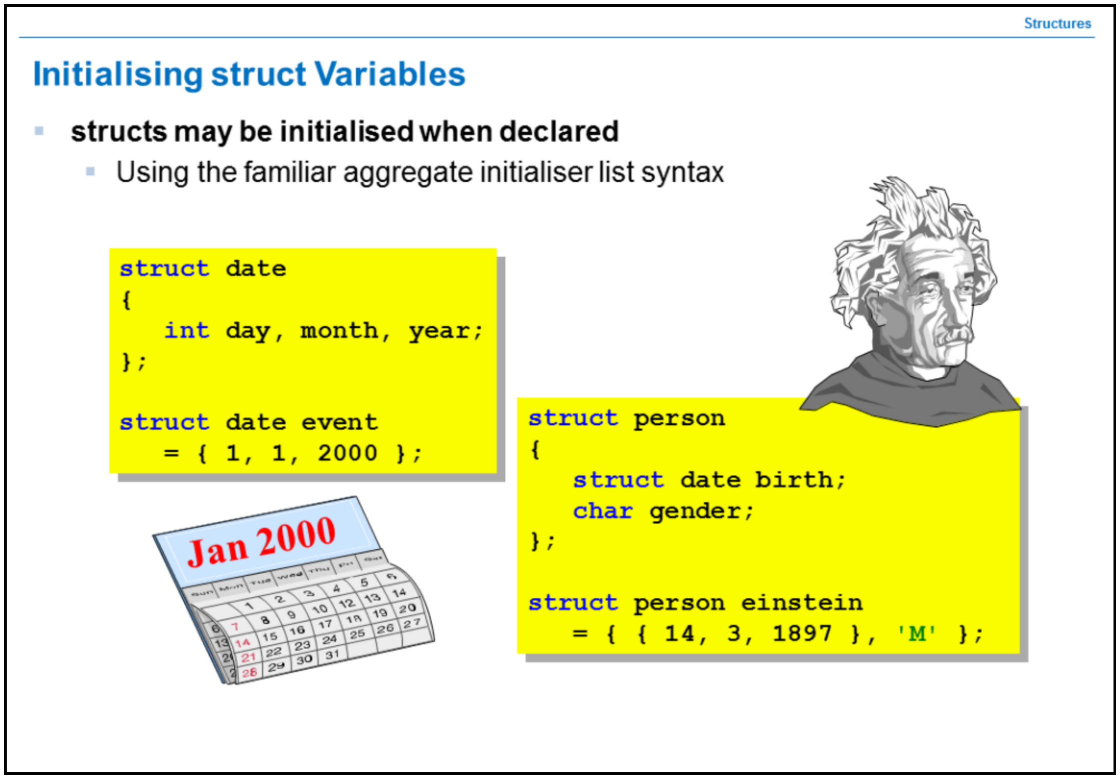
```
struct person
{
   struct date birth;
   char gender;
};

struct person baby;
```

The declaration of variables simply uses the keyword `struct` and the name of the struct type and the name of the variable to be declared:

```
struct person me;         /* one person variable */
struct person you;        /* another person variable*/
struct person a,b;        /* two in one go */
```

The data member declarations within a `struct` definition are like ordinary variable declarations, but without any initialisers. These data members may be of any type: fundamental types, arrays, other `struct` types, etc, etc. Note that data member names do not interfere with the names of variables declared elsewhere: they are enclosed in the *namespace* of the `struct` variable:

```
struct alpha { int value; };
struct beta  { int value; }; /* OK */
```



Initialising struct Variables

- **structs may be initialised when declared**
  - Using the familiar aggregate initialiser list syntax

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

struct date event
   = { 1, 1, 2000 };
```

```
struct person
{
   struct date birth;
   char gender;
};

struct person einstein
   = { { 14, 3, 1897 }, 'M' };
```

An array is an aggregate data type and is initialised using an *aggregate initialiser*. A struct is also an aggregate data type so it should come as no surprise that a struct can also initialised using an *aggregate* initialiser.

The members of a `struct` type variable are initialised by the aggregate initialiser in the same order as their appearance in the `struct` definition. This allows a `struct` variable to be initialised all at once. Alternatively, a `struct` variable may be initialised from another struct variable of the same type.

Where initialisers are omitted the compiler 'right fills' the `struct` with default values, which in the case of numeric types is zero. The compiler will flag an error if there are more initialisers in the initialiser list than there are declared members:

```
struct date oops = { 1,1,2000,42 };
/* compiler error */
```

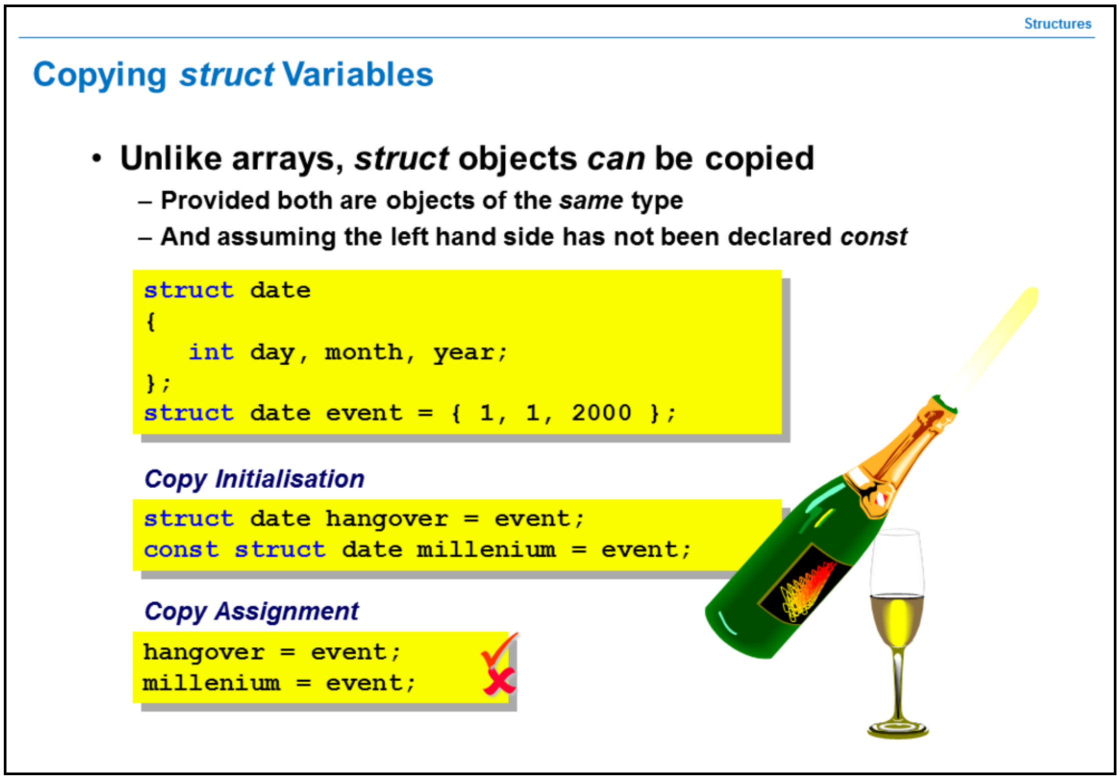
It is not possible to initialise a struct using an empty initialiser list:

```
struct date first = {}; /* compiler error */
```

If the empty list is omitted then the initial values of the struct are undefined, i.e. garbage:

```
struct date when;
```

In this case there is a 50% chance that the value of each `int` member will be negative!



Copying *struct* Variables

- **Unlike arrays, *struct* objects *can* be copied**
  - Provided both are objects of the *same* type
  - And assuming the left hand side has not been declared *const*

```
struct date
{
   int day, month, year;
};
struct date event = { 1, 1, 2000 };
```

Copy Initialisation

```
struct date hangover = event;
const struct date millenium = event;
```

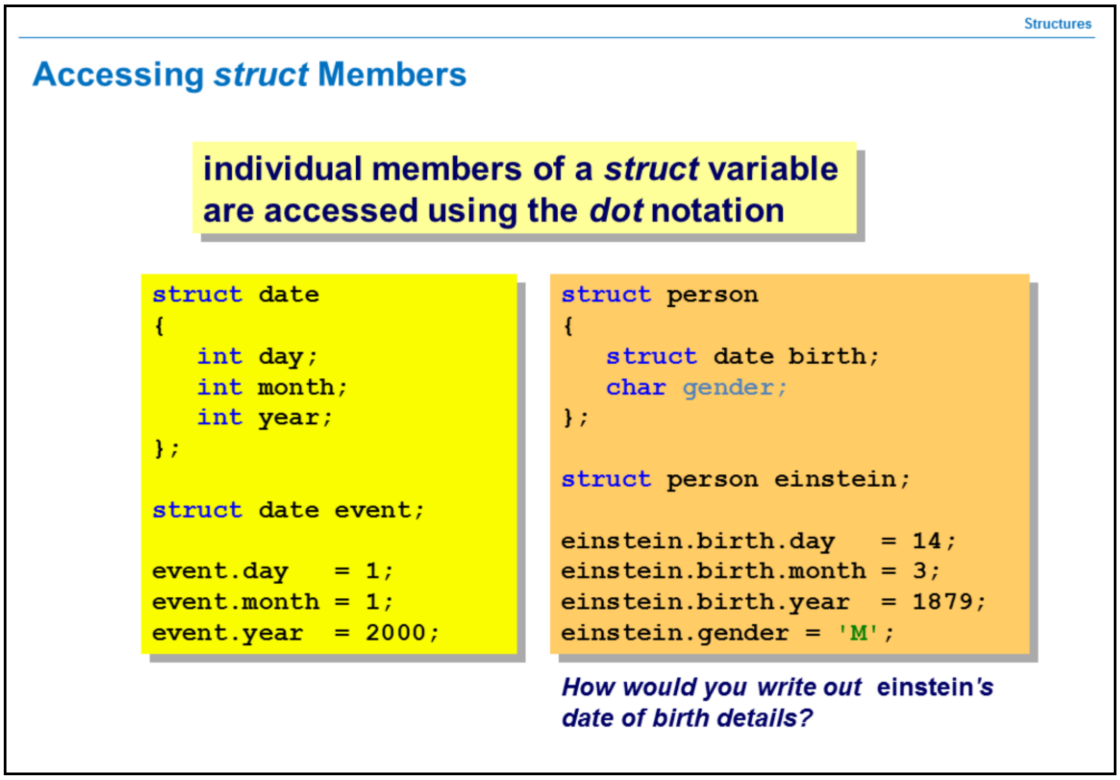
Copy Assignment

```
hangover = event;       ✓
millenium = event;      ✗
```

Variables of `struct` types may also be assigned to and from other variables of the same type. Members may be accessed for assignment individually using dot notation (see next slide). This provides a way of selectively accessing parts of a `struct`.

Sometimes `structs` are used to hold reference data that should not be modified. As with other constants, a `struct` variable can be qualified as `const` when it is declared and initialised. This prevents subsequent code changing either the whole variable or any of its members.

Note the difference between arrays and `structs`: although they are both *aggregates*, and may be initialised with the aggregate initialiser syntax, `structs` can be assigned or copied one to another whereas arrays cannot.



Structures

Accessing *struct* Members

individual members of a *struct* variable are accessed using the *dot* notation

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

struct date event;

event.day   = 1;
event.month = 1;
event.year  = 2000;
```

```
struct person
{
   struct date birth;
   char gender;
};

struct person einstein;

einstein.birth.day   = 14;
einstein.birth.month = 3;
einstein.birth.year  = 1879;
einstein.gender = 'M';
```

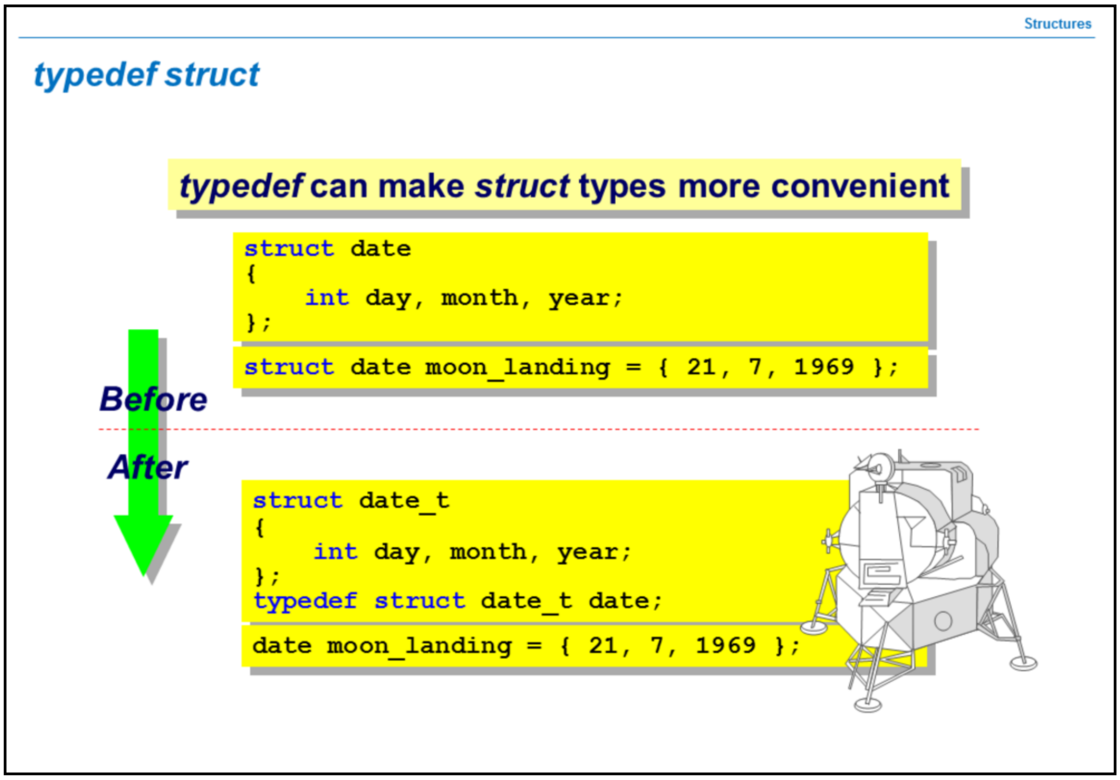
How would you write out* einstein's *date of birth details?

To access a member within a `struct` variable the dot notation is used. The name of the containing variable is followed by a dot and then the name of the member being accessed. This applies in turn to any members nested inside members.

As with arrays, there is no implicit I/O for `struct` types. This must be performed by the user:

```
printf("%d:%d:%d",
          einstein.birth.day,
          einstein.birth.month,
          einstein.birth.year);
```

In practice we might well factor this detail into a purpose built function. See the *structs and Functions* slide a few pages ahead.



typedef struct

***typedef* can make *struct* types more convenient**

```
struct date
{
    int day, month, year;
};
struct date moon_landing = { 21, 7, 1969 };
```

Before
After

```
struct date_t
{
    int day, month, year;
};
typedef struct date_t date;
date moon_landing = { 21, 7, 1969 };
```

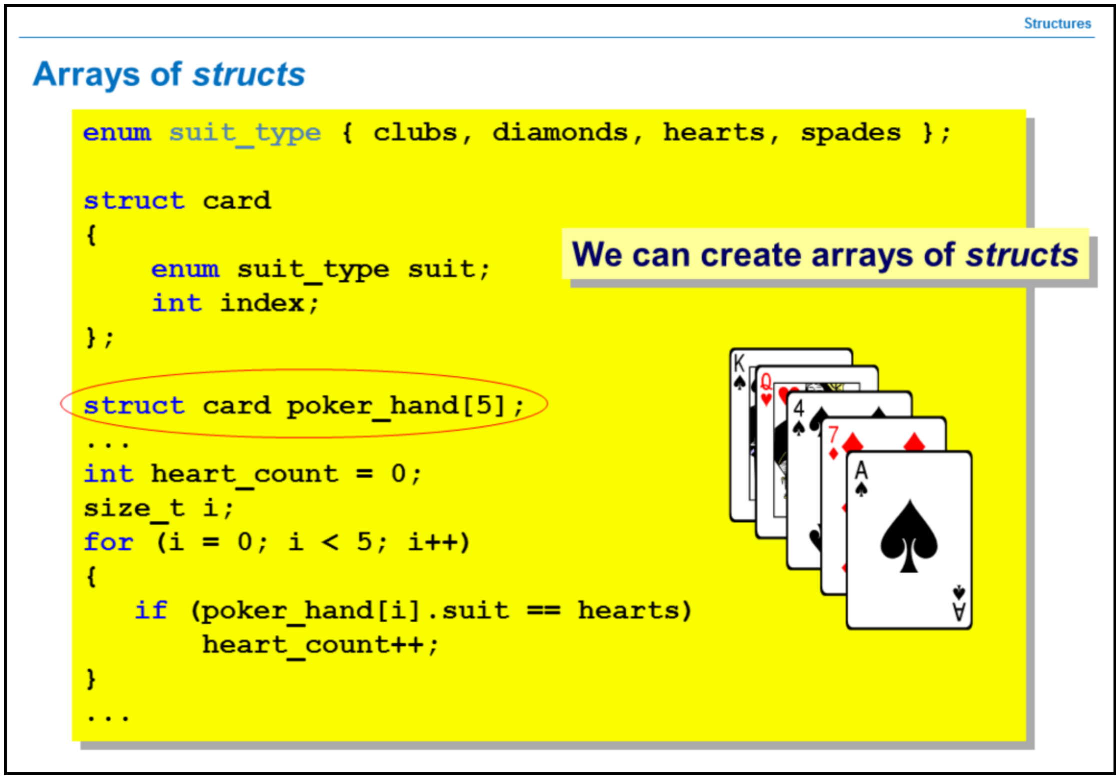
The syntax of a `typedef` declaration is like that of a normal declaration prefixed with the keyword `typedef`.

```
        struct date wibble; /*1*/
typedef struct date wibble; /*2*/
```

Line one declares a variable called wibble of type struct date. Line 2, which is exactly the same except it is prefixed with the typedef keyword does not declare any variables at all. Line 2 declares wibble to the typename alias for struct date.

In other words, when writing a typedef, instead of declaring a variable, the identifier becomes another name or synonym for the declaring type. The declaring type must be an existing type name: either a built-in type, such as `int`, another `typedef` declared name, or a new user defined type.

Type definitions normally live in header files as they are typically used by many other source files in a system. By placing types in header files their definitions can be shared and easily accessed using the `#include` preprocessor directive.



Arrays of *structs*

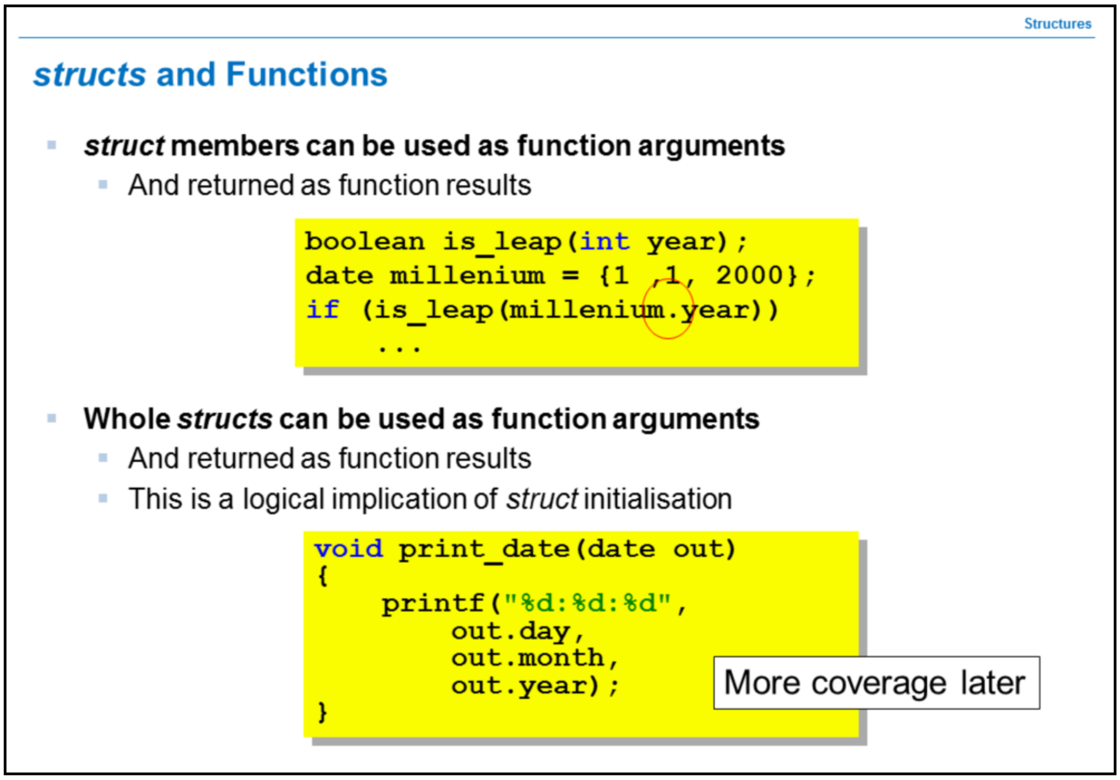
```
enum suit_type { clubs, diamonds, hearts, spades };

struct card
{
    enum suit_type suit;
    int index;
};

struct card poker_hand[5];
...
int heart_count = 0;
size_t i;
for (i = 0; i < 5; i++)
{
   if (poker_hand[i].suit == hearts)
       heart_count++;
}
...
```

We can create arrays of *structs*

This, the second example of nested structures, illustrates an array of structures. As in the other example, access is made into the outside aggregate first, i.e. the array. Access operators are then applied to the array element, itself an aggregate, in order to get to the nested item.

Another example of an array of structures is given below.

```
struct student course[10];
...
course[0].married_flag = 'Y';
...
course[2].enrolment_date.day = 1;
course[2].enrolment_date.month = 1;
course[2].enrolment_date.year = 1994;
...
for (i = 0; i < size; i++)
{
    printf("Exam mark %d is %d%%\n", i,
            course[9].exam_marks[i]);
}
```



structs and Functions

- *struct* members can be used as function arguments
  - And returned as function results

```
boolean is_leap(int year);
date millenium = {1 ,1, 2000};
if (is_leap(millenium.year))
    ...
```

- Whole *structs* can be used as function arguments
  - And returned as function results
  - This is a logical implication of *struct* initialisation

```
void print_date(date out)
{
    printf("%d:%d:%d",
        out.day,
        out.month,
        out.year);
}
```

More coverage later

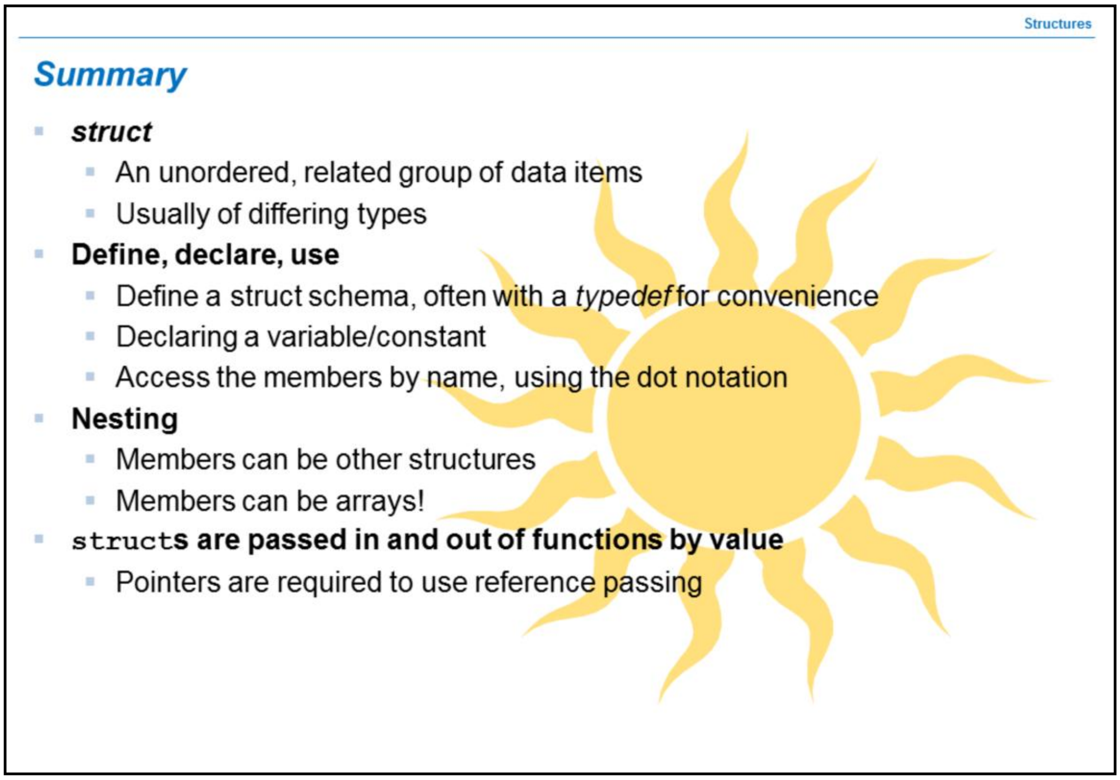
Functions have not been mentioned in this chapter because you are encouraged to use a call by reference (using pointers) for all function communication involving structures. Efficiency is the reason. Structures can be very large, and call by value could be expensive in terms of time and space.

In the slide a date variable is passed by value to the print_date function. This then prints the three members of the date. If we wished to print the name of the month we could use an array of string literals as a lookup table which we saw in the Arrays chapter. The code for this might be:

```
const char * const month_names[] =
    {
        "January",
        ...
        "December"
    };
    printf("%d:%s:%d",
            copy.day,
            month_names[copy.month - 1],
            copy.year);
```

The -1 is because the month value is one-based whereas arrays are zero-based. There are two consts in the declaration of month_names. The pointer elements inside the array are const and the elements they point to (strings) are also const. Pointers will be covered in detail in later chapters. An experienced C programmer might well print a date using the strftime library function in <time.h>.

Like all non-pointer based data types, the `struct` is passed in and out of functions by reference. These items could get large, so passing by reference is preferable. The Pointers and Structures chapter will cover this in more detail.

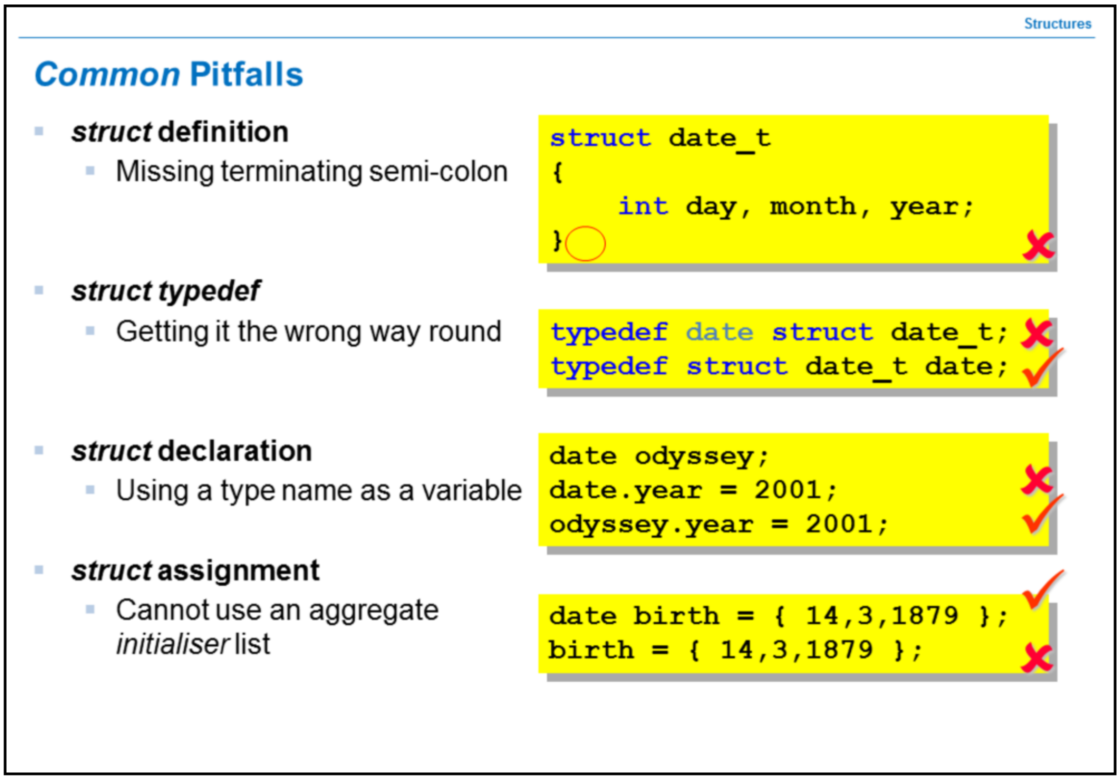


Structures

Summary

- ***struct***
  - An unordered, related group of data items
  - Usually of differing types
- **Define, declare, use**
  - Define a struct schema, often with a *typedef* for convenience
  - Declaring a variable/constant
  - Access the members by name, using the dot notation
- **Nesting**
  - Members can be other structures
  - Members can be arrays!
- `struct`**s are passed in and out of functions by value**
  - Pointers are required to use reference passing

Structures will be discussed again in the *Pointers and Structures* chapter. Nothing new in the concept of the structure will be introduced. The major objective will be the introduction of call by reference and the operator that eases the task of accessing the member using a pointer.



Structures

Common Pitfalls

- *struct* definition
  - Missing terminating semi-colon
- *struct typedef*
  - Getting it the wrong way round
- *struct* declaration
  - Using a type name as a variable
- *struct* assignment
  - Cannot use an aggregate *initialiser* list

```
struct date_t
{
    int day, month, year;
}                              ✗
```

```
typedef date struct date_t;  ✗
typedef struct date_t date;  ✓
```

```
date odyssey;
date.year = 2001;            ✗
odyssey.year = 2001;         ✓
```

```
date birth = { 14,3,1879 };  ✓
birth = { 14,3,1879 };       ✗
```

Omitting semi-colons – or including them where they are unnecessary – is one of the most common errors that a newcomer to C will make. A semi-colon is required on the end of a struct definition:

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

A common mistake is to get the typedef declaration reversed. Remember, the syntax of a `typedef` declaration is like that of a normal declaration prefixed with the keyword `typedef`.

A struct definition provides a new type name and a structural description of that type. It does not provide a variable that you can use. For this a separate declaration is needed:

```
        struct date special;
        special.year = 1905;
```

The aggregate initialiser syntax can only be used for initialisation. It cannot be used for assignment.

Intentionally left blank