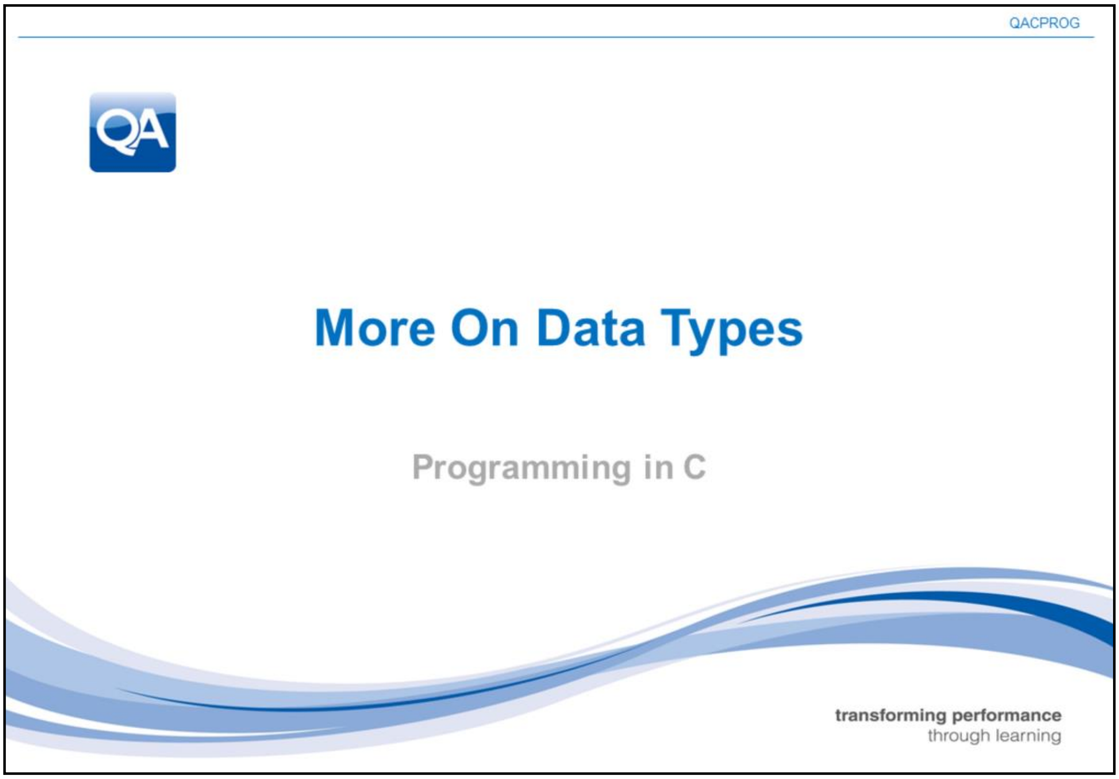
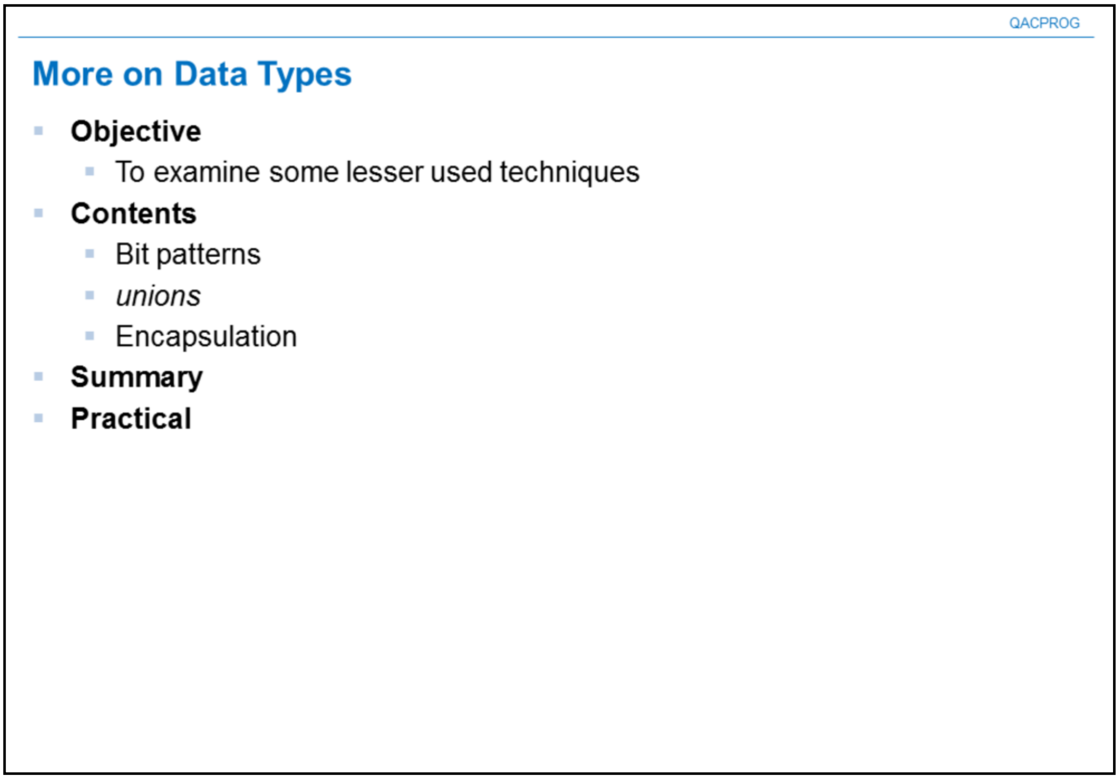
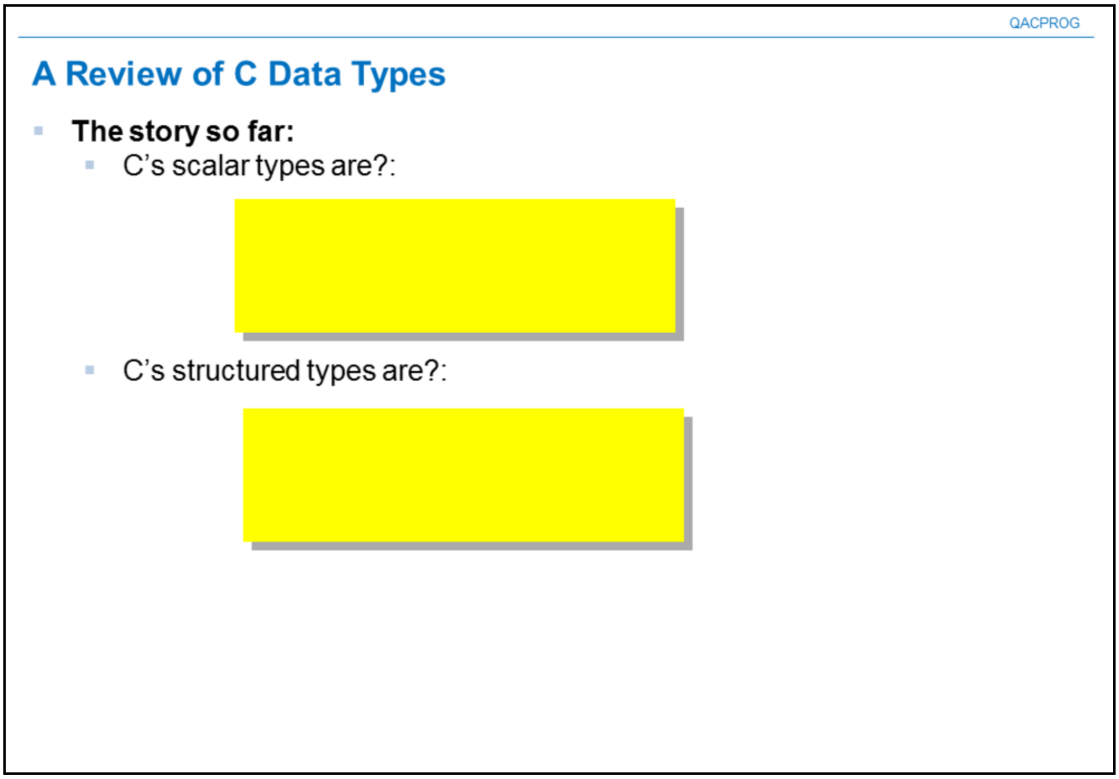

QACPROG
More On Data Types
Programming in C
transforming performance
through learning



The objective of this chapter is to introduce techniques using the standard types already covered in the course. No new data types are introduced, only new ways of dealing with them.

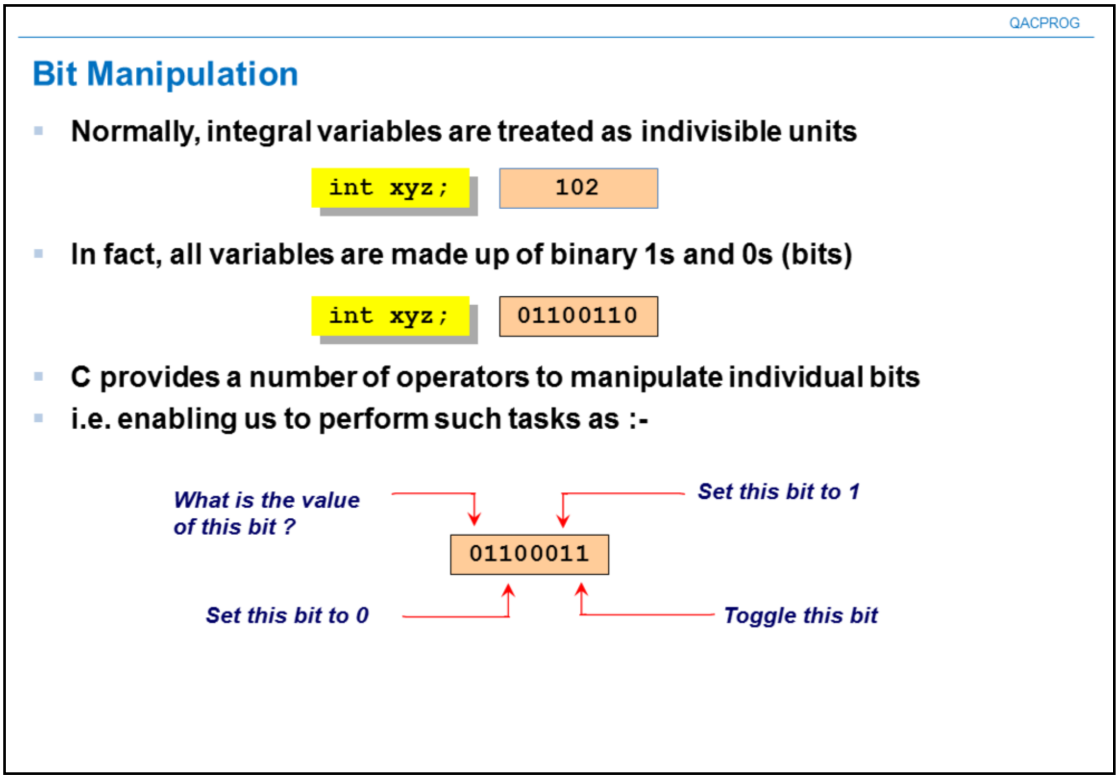


There are only three really fundamental types, i.e. the `char`, the `int` and the `double`.

- There are three varieties of char: plain char, signed char and unsigned char. By definition, the size of a char is a single byte.
- There are three varieties of int: short, int, long. All three of these can be signed or unsigned.
- There are three varieties of floating point numbers: float, double and long double.

Strictly speaking, we can add const and volatile to all these type to create yet further types.

The three structured types encountered so far are: the array, the enum, and the struct.



Bit manipulation is only available for `int`s and `char`s. As with all bit manipulation, experience is required in handling the operators and in interpreting the results. In order to achieve bit-manipulation operations like the ones mentioned in the slide, it is necessary to combine one or more of the bit operators described on the next page.

Examples in the chapter use 8-bit patterns.

QACPROG

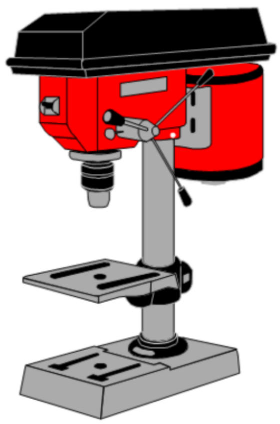
Bit Operators

- C's operators work with integral types interpreted as 'bit bags'
- Unary operator

```
~    Complement
```

- Binary operators

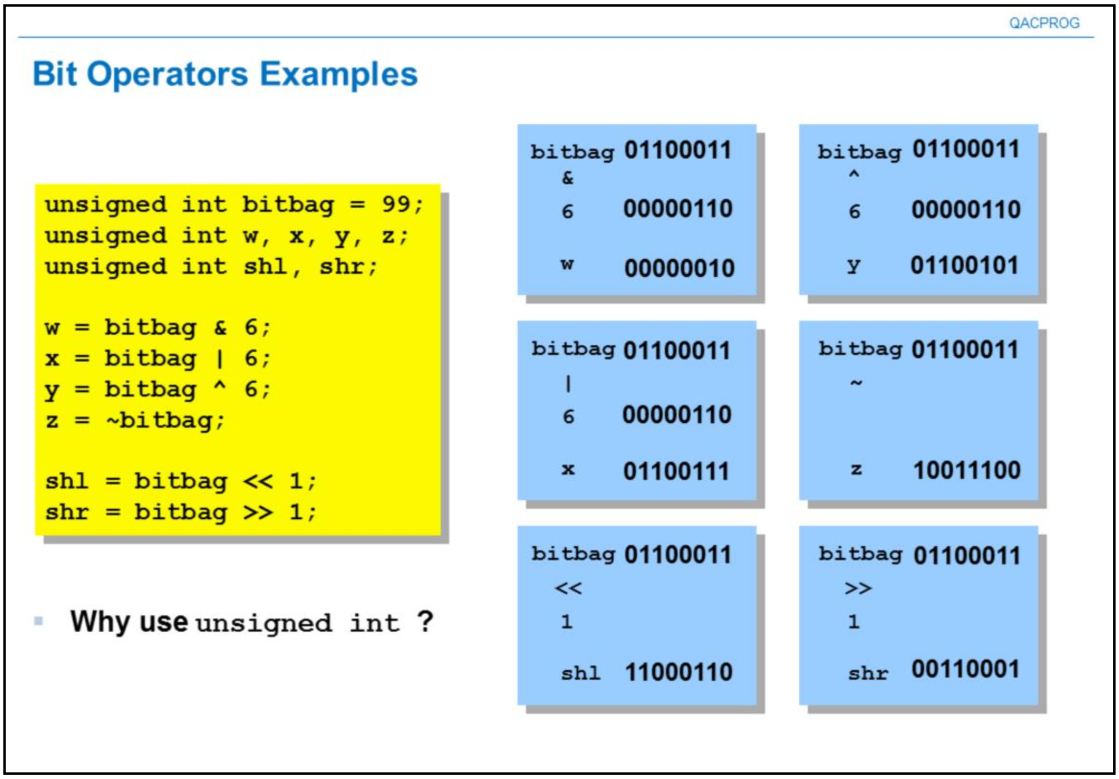
```
<<   Left Shift
>>   Right Shift

&    AND
|    OR
^    XOR (Exclusive OR)
```



All the standard basic operators are available. Arithmetic and rotational shifts have to be performed using the basic shift operators (which are logical), although there are exceptions to this rule.

The shifts take their type from the left expression. This cannot be promoted by the size of the right-hand expression, which represents the number of places to shift, e.g.:

```
int x = 5 ;
long int big = 0L;
...
big = x << 12;        /* x does not get promoted to a long */
```

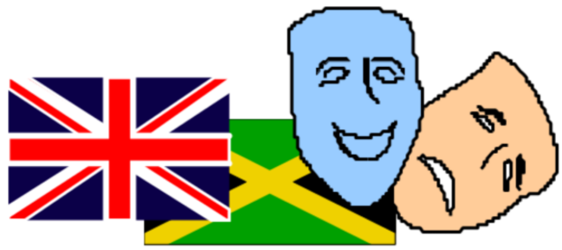


The use of the `unsigned int` is encouraged. It overcomes some portability and representation problems.

QACPROG

Notes on Bitwise Operators

- **Main use of bit manipulation...**
  - Masking
  - Status flags
- **Words of warning...**
  - Beware of low precedence of operators (especially << and >>)
  - The logical operators && and || are not the same as the bitwise & and |
  - Right shifting may be arithmetic on some architectures, so use unsigned whenever possible



The code shown below illustrates some of the problems.

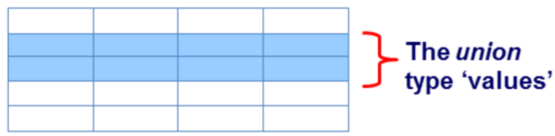
```
void problem_bits(void)
{
      unsigned int x, y;
      ...
      x = y * 8;
      x = y << 3;         /* Same ? */
      ...
      x = y / 16;
      x = y >> 4;         /* Same ? */
      ...
      x = y * 8 + 4;      /* Using multiply */
      x = y << 3 + 4;     /* No! + has higher */
                          /* priority than  < */
      x = (y << 3) + 4;    /* Correct  - use brackets */
      ...
}
```

QACPROG

unions

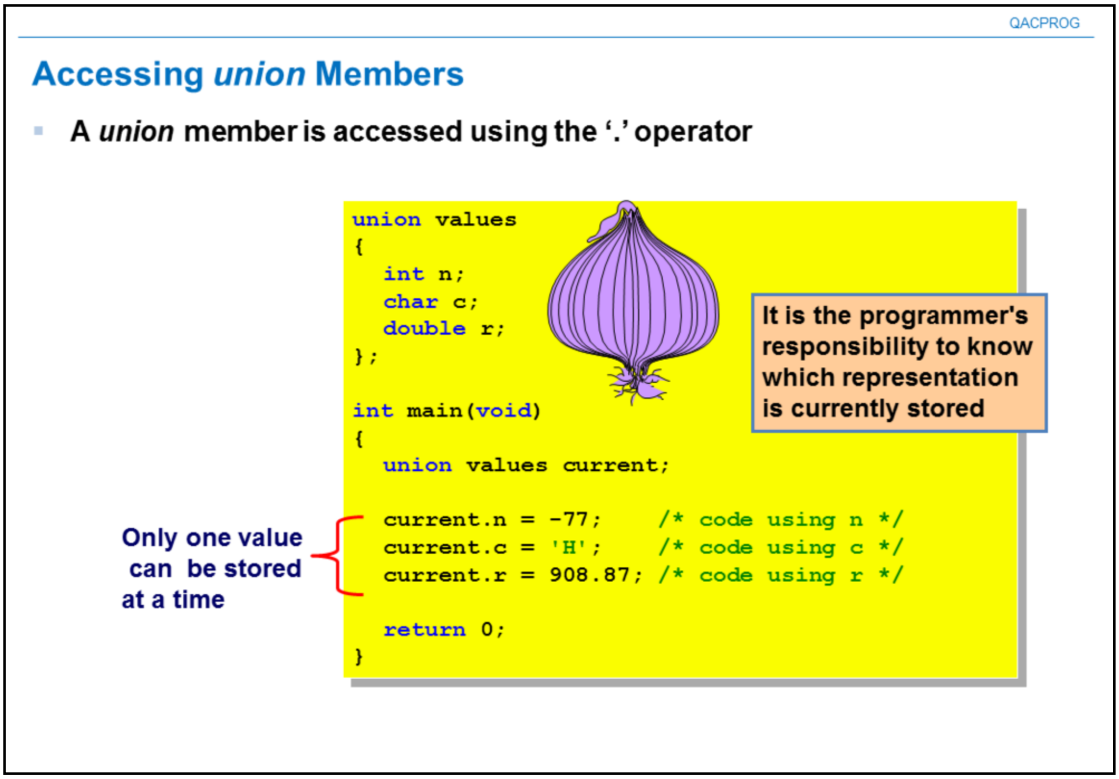
- A union is a variable that may hold data of different types and sizes at different times
- Unions follow the same syntax as structures, but have members that share storage

```
union values
{
     int    n;
     char   c;
     double r;
};
```



- The C compiler keeps track of size and alignment requirements and allocates memory that accommodates the largest of the specified members
- How much space would be allocated for a variable of type 'union values'?

The `union` is a structure in which all the data members have zero offset. It is dangerous to assume any internal representation, because this is compiler specific. The programmer is guarantied that, if used correctly, the `union` will provide an efficient mechanism for simple data overlay when storage must be used efficiently.



Accessing the `union` members is the same as accessing structure members; the dot operator is used. Problems will arise if access is made to a data item that is currently overlaid with one of its fellows. For example, the following sequence will give indeterminate results:

```
char ch ;
...
/* Assign a double - the largest data member */
current.r = 123.45;
/* Careful - grab hold of the first byte from current */
ch = current.c;
...
```

The programmer who is knowledgeable about the underlying memory allocation and data representations can use unions for masking, bit manipulation on non-integers and other low-level techniques. `unions` are inherently non-portable if these techniques are used. A portable example is given on the next page.

Notes: ISO does support the initialisation of `unions`, using data matching the first member. Although it is legal to define `const` unions, for obvious reasons, any attempt to modify the contents will result in undefined behaviour.

QACPROG

union Example

- Consider the problem of recording a customer's postal address in a customer record system. A full postal address might be needed for personal customers, while business customers' addresses might simply be a reply box number and town
- Only one customer record type is required for ease of processing
- Note that the union is wrapped up in a struct. This has a member that helps to identify which union member is present

```
/* Business address format */
struct business
{
    int  box_no;
    char town[30];
};
```

```
/* Personal address format */
struct personal
{
    int  house_no;
    char street[25];
    char town[30];
    char post_code[8];
};
```

```
/* Holds either address format */
union address
{
      struct business reply_box;
      struct personal full_address;
};
```

```
struct customer_address
{
      int cust_no;
      int address_type;
      union address postal;
};
```

As long as access to the appropriate `union` members, themselves structures, is made in a sensible manner, the example shown above could be implemented in a portable way. This is helped by wrapping the union up in a structure like `customer_address`, which contains a data member, i.e. `address_type`. The data member can keep track of which of the two `union` members is currently held. Accessing the appropriate address is then made by a simple check on this integer.

QACPROG

Encapsulation in C

- **True encapsulation, i.e. as defined in the Object Model, is not possible**
- **C does its best using typedef and 'dedicated' functions**
- **Best example is FILE**
  - '`typedef`'ed `struct`
  - All functions take a `FILE *` as first argument.

```
#include <stdio.h> /* Defines FILE  */

int main(void)
{
    FILE * file_ptr;
    int firstChar;
    file_ptr = fopen("myfile.ext", "r");

    if (!file_ptr)
    {
          firstChar = fgetc(file_ptr);
          ...
    }
}
```

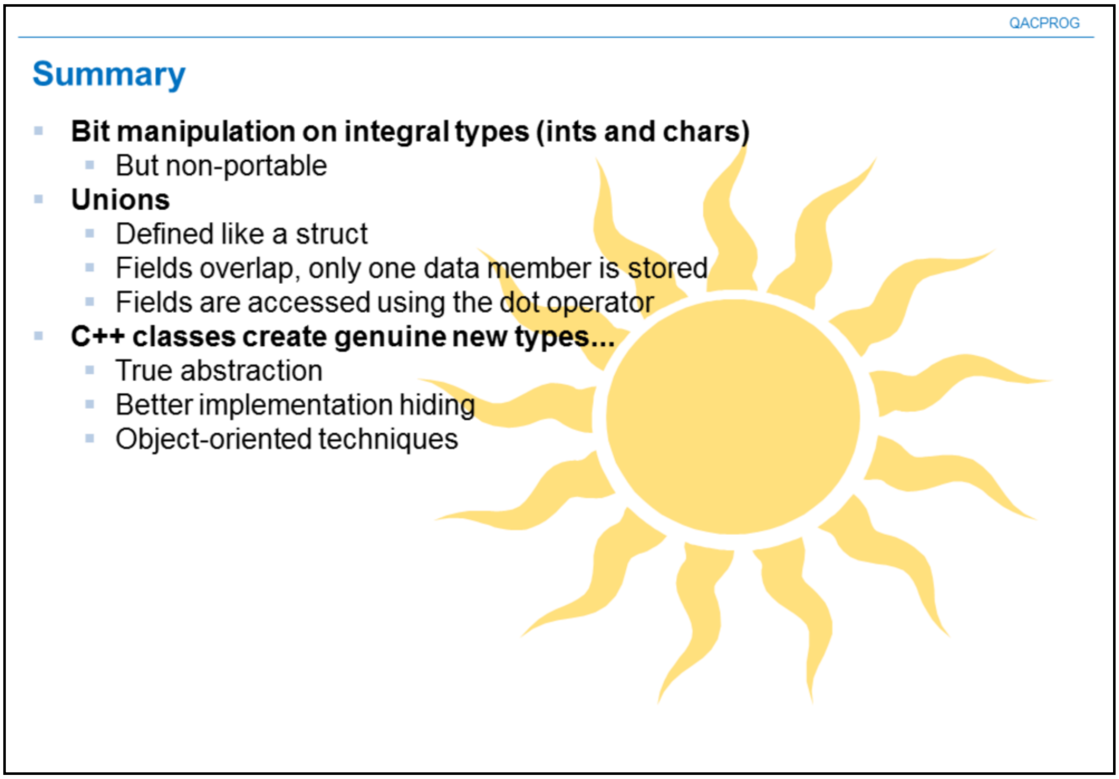
There is no true encapsulation in C. In the strict definition of the Object Model, on which OO is based, encapsulation implies privacy and information hiding. There is no `private` keyword, so all data in scope is 'up for grabs'.

However, we have the C scope rules and information 'obscuring' using `typedef`. The `FILE` type is a `typedef`ed `stuct _iobuf`. All th efile manipulation functions take a `FILE *` as its first argument, in some cases the only argument (see `fget`() above). The value of the `FILE *` is kept from us as it is returned from the `fopen`() function and released by the `fclose`() function.

The only time we need to check on the `FILE *` is to check that it is not `NULL/0`, the value returned from `fopen()` if there is any problem.

Physically, we can envisage the `struct` and its `typedef` together with any other relevant tokens residing in file.h header file and all the file functions residing in file.c. As it happens, the `FILE` type is 'bundled in' with the other IO functionality.

This is the basis for C++ encapsulation which uses a struct-like mechanism (the class) and public and private keywords to implement true encapsulation.



Bits are manipulated within bit patterns. A bit pattern could be an 8, 16, 32, etc. integral data item. Manipulation requires expertise and experience. The underlying representations should be known, and any representation is implementation specific.

`typedefs` are aliases for data type names. These could be the C built-in types or they could be names built up from previous `typedefs`. The compiler restricts the names to be used only with types, which is more restricting than `#define`. However, the `typedef` is more flexible, since arrays and functions can be defined as 'new' types. Abstraction, using `typedef`, requires heavy investment.

C++ performs the task of data abstraction much more gracefully using classes. The language is far richer than C. It is virtually a superset of C, but requires an insight into object-oriented techniques to ensure that the full benefit is gained from the language.