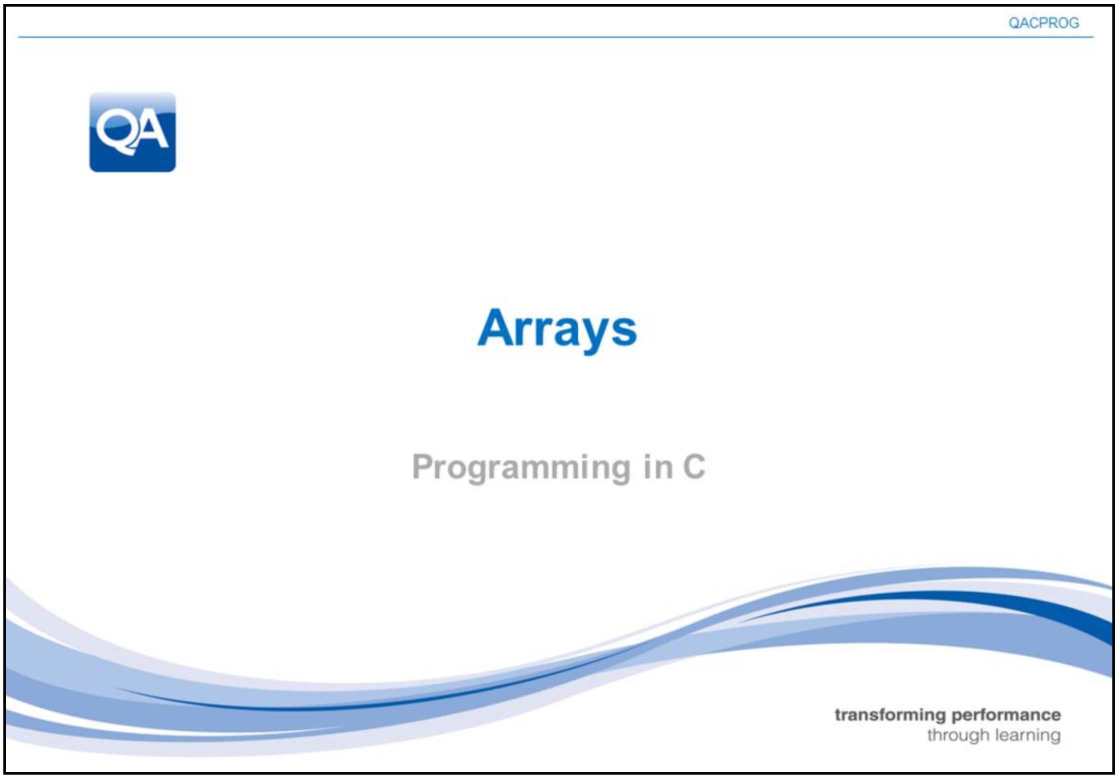

QACPROG
Arrays
Programming in C
transforming performance
through learning

QACPROG

Arrays

- **Objective**
  - To design and represent compound information
- **Contents**
  - Data types - a review
  - Declaring an array
  - Accessing array elements
  - Arrays with functions
  - Strings - character arrays
  - Multidimensional arrays
- **Summary**
- **Practical**
  - Simple manipulation of general arrays and strings

This is the first of two chapters covering C's compound types, or aggregates. The objective of both chapters is to enable the programmer to design and implement a representation of data in a specification. This, the first chapter, deals with groups of like data components which in C are implemented as arrays.

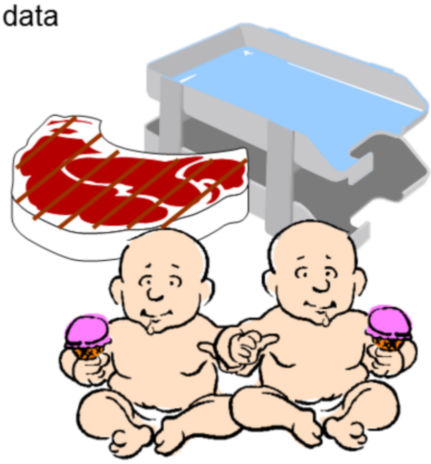
The chapter contains a overview of array concepts and how C implements the array. This is followed by general array manipulation and the way C implements strings. Although not covered in detail, there is a discussion on two-dimensional arrays at the end of the chapter.

QACPROG

Review of *Data* Types

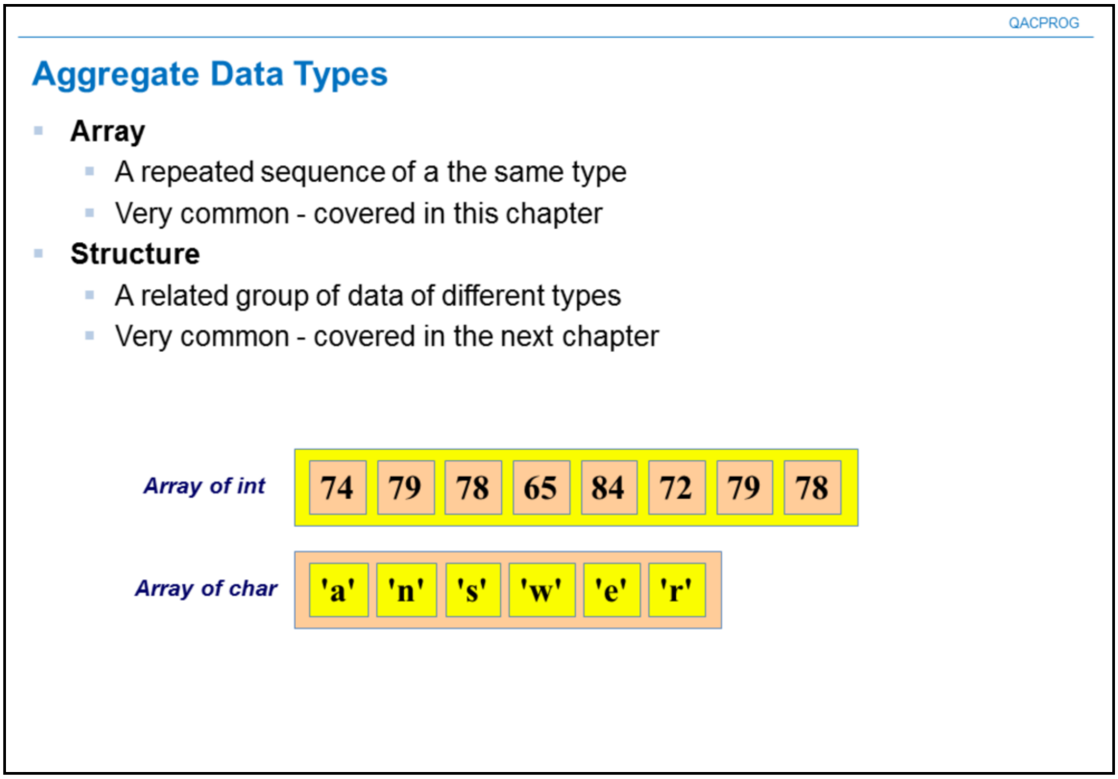
- **Built in data types create *single* data items**
  - Also called scalar data types
- **But data almost never exists in isolation**
  - Data is often aggregated with related data

```
int main(void)
{
    int    ray;
    char   grilled;
    double trouble;
    ...
}
```



Before one can appreciate the effective use of an array, it is necessary to review what is known about individual data items. All of C's basic scalar data types have been covered. A scalar data type contains a single entity that represents a direct piece of data, e.g. an `unsigned long int`, a `double`, etc. The scalar data type is declared by the programmer, who supplies two pieces of information, i.e. the type and the name. The data type could be qualified by the keywords `const`, `auto`, `static`, `register`, `volatile` and `extern`.

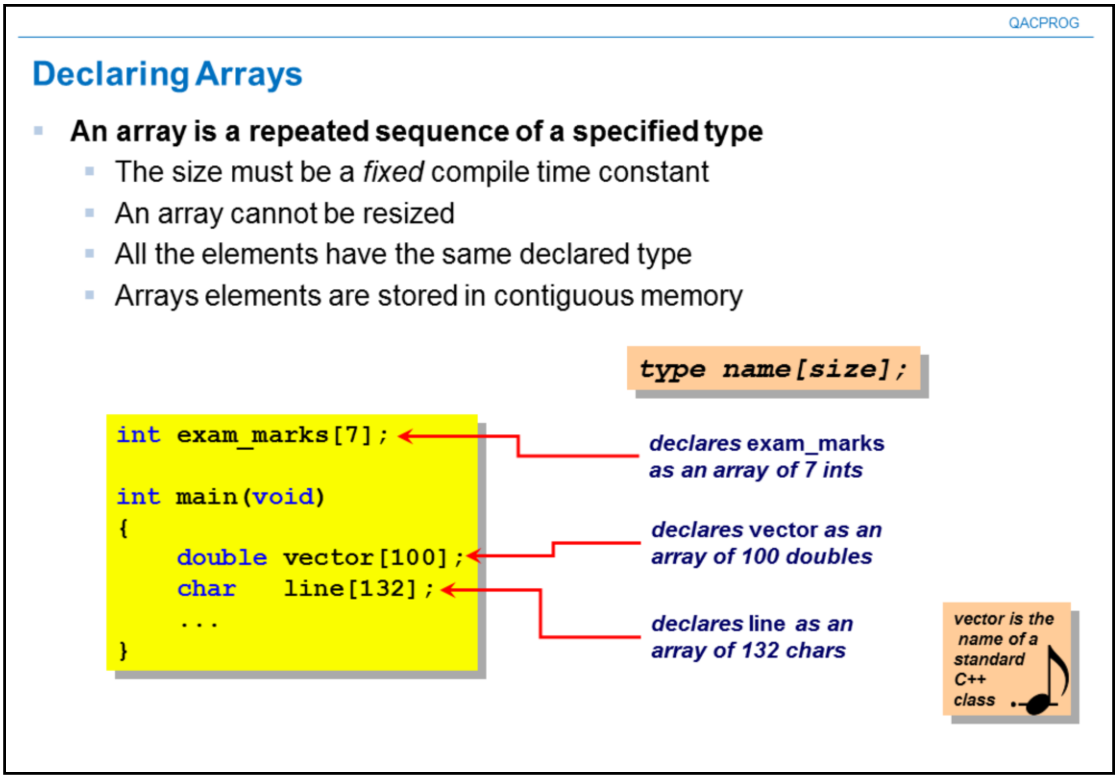
Other scalar types are pointer types. Pointers also hold a single entity, which is not actual data, but an address that will give indirect access to data. Pointers are covered in a dedicated chapter later on in the course.



Aggregates contain collections of logically-connected data items. There are two categories of aggregates:arrays contain an ordered sequence of data items of the same type; structures contain an unordered group of data items of possibly differing type. Unions are a rarer, third kind of aggregate data type. Unions contain a single data item, but the data item can be chosen from a group of data items. Unions are not covered in this course.

The data item could refer to scalar or aggregate, i.e. the following are all possible:

An array of any scalar type (including pointers).
An array of arrays.
An array of structures.
An array of unions.
A structure containing scalars, arrays, other structures and unions.
A union containing scalars, arrays, structures and other unions.



An array is a collection of elements all of the same type. The declaration of an array must specify this type and the size of the array. The size must be a compile time constant and is specified between square brackets, [and], and not parentheses, (and), as in some other languages. This means the following will compile:

```
int scores[12];
```

but the following will not compile (class_size is not a compile time constant):

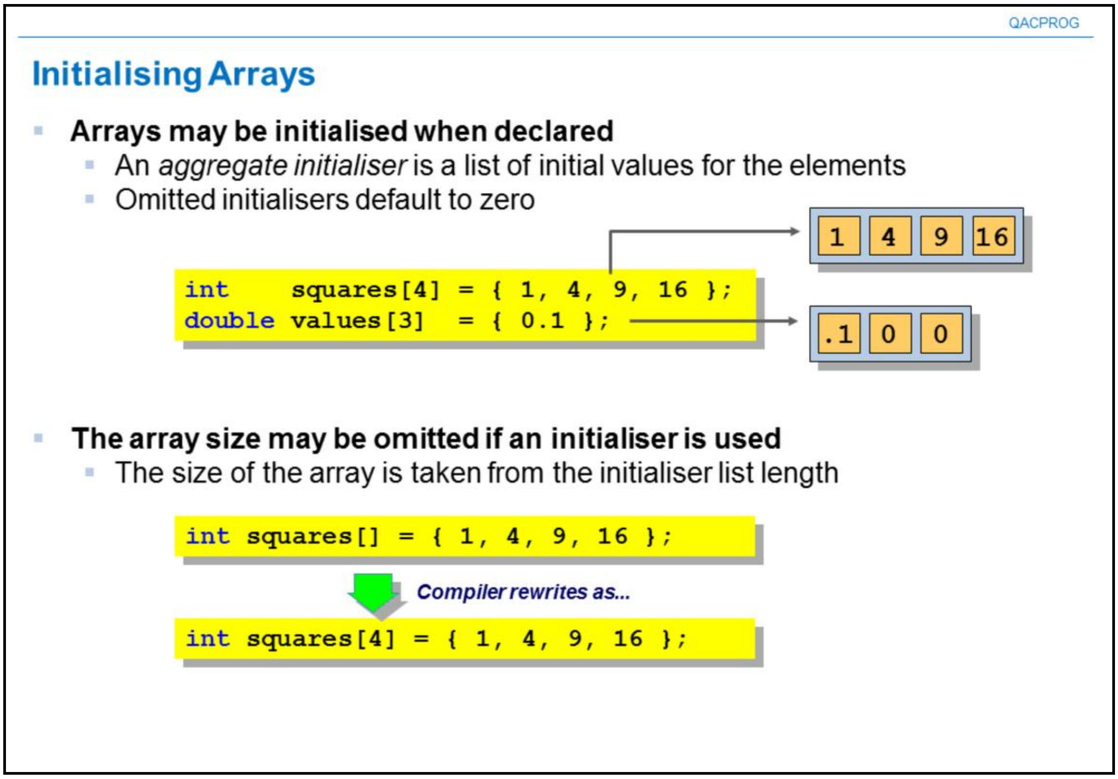
```
size_t no_of_students = 12;
int scores[no_of_students];
```

Surprisingly, an integer declared const is not considered to be a compile time constant. Hence, this will also not compile (although it will in C++):

```
const size_t no_of_students = 12;
int scores[no_of_students];
```

This appears to force you to declare an array using an integer literal. This would not be good since such literals (so called magic numbers such as 12) are not very declarative and are hard to maintain. Fortunately there is a way to declare an array using an expressive identifier for the array size - we can use the name of a enum constant:

```
enum { no_of_students = 12 }
int scores[class_size];
```



Although they have a slightly cumbersome name, aggregate initialisers offer a simple syntax of providing an array with initial values. Aggregate initialisers are executed from left to right. Where initialisers are omitted the compiler 'right fills' the array 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 elements.

```
int overflow[2] = { 23, 11, 1966 }; // compiler error
```

In C it is not possible to have an empty initialiser list (it is in C++).

```
int all_zeroed[5] = {}; // compiler error
```

If the empty list is omitted the initial values of locally declared array (that is, declared inside the scope of a function) are undefined, i.e. garbage. A particular compiler may initialise local arrays to all zero, but portable code cannot rely on it.

```
void f(void)
{
   int uninitialised[5];
   ...
}
```

If a complete initialiser list is provided the array size can be omitted and an empty [] used in the declaration: the compiler deduces the size from the number of elements provided in the initialiser list.

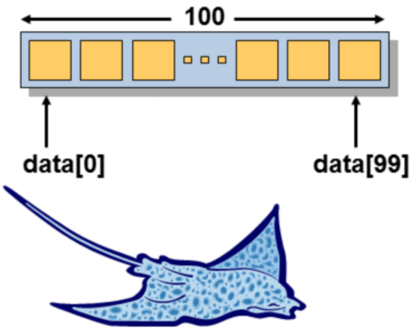
QACPROG

Accessing Array *Elements*

- **Arrays are indexed from *0* using the subscript operator**
  - An array with *size* elements has valid subscripts from *0* to size-*1*
  - Subscript indexes are *not* checked

```
array_name[integer_expression]
```

```
...
int index;
double data[100];
...
data[0] = 0.0;
data[1] = data[0];
...
scanf("%d", &index);
if (0 <= index && index < 100)
{
    printf("%lf", data[index]);
}
```

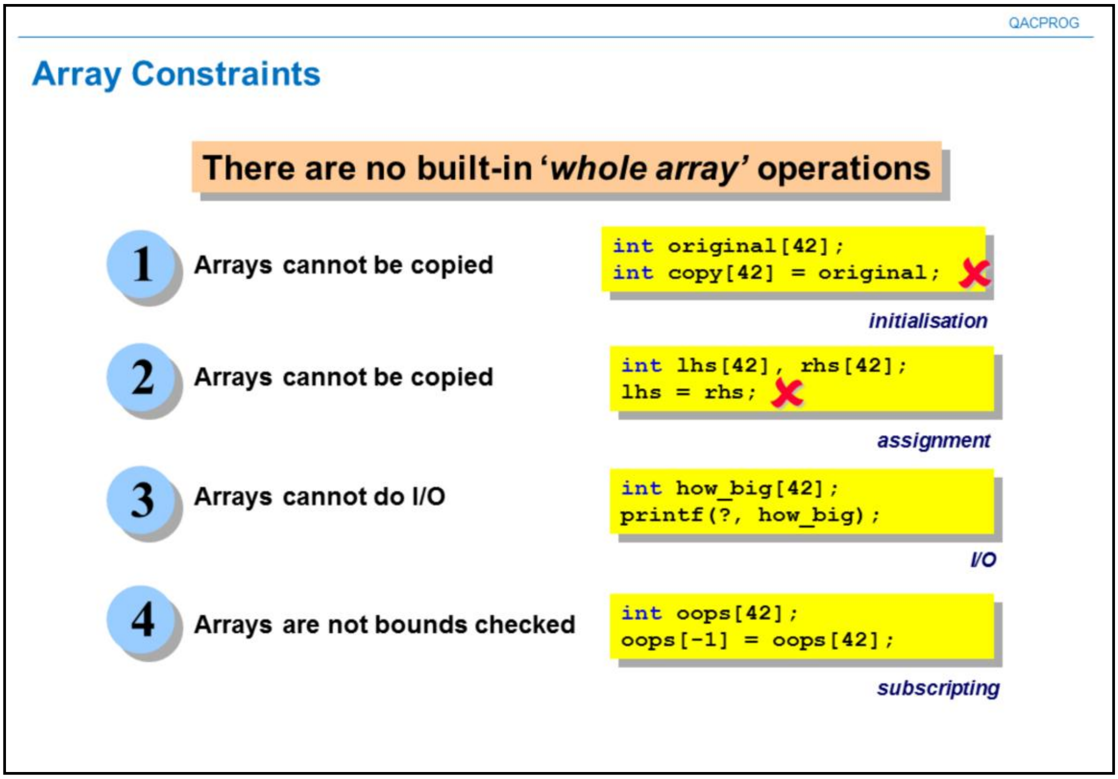


C is very precise about indexing an array. An array of `N` elements are indexed using the integral values `0` to `N-1`. There is no alternative. The syntax uses the `[` and `]` brackets, but care must be taken. The compiler will generate code using the base address of the array, i.e. the location of element `0`. The expression in the brackets must be integral. The value is used as a direct offset into the array.

Following on from the example shown above, the following code is dangerous, but is accepted by the compiler without any warnings:

```
...
d = data[100];      /* No! - bounds are [0..99] */
data[100] = 10.2;   /* Again, da[100] does not exist */
d = data[-5];       /* Silly, but possible */
data[-5] = 10;      /* Even this is possible */
...
```

It is vital to build bound checking into your code. The onus is on you!



There are no whole array operations in C. The only time the name of an array refers to the whole array is when the array is being declared in a *declaration* (and sometimes an argument to the `sizeof` operator). In an *expression* the name of an array always refers the the address of the initial element of the array.We will see this in more detail shortly. Realising this we can see why none of the above operations makes sense.

In the first case we cannot initialise `copy` from `original` because `original` refers not to the whole array called `original`, but the address of the initial element of `original`. In other words we are trying to initialise an array of 42 integers (called `copy`) as a copy of an address of a *single* integer! The types are not the same.

The second case a similar. However in the statement `lhs = rhs;` *both* identifiers refer to the address of their respective initial elements. The types are the same in this case. However, what *is* the address of the initial element of `lhs`? It is the *fixed* address where the compiler has decided to allocate the space for the array. And arrays cannot be relocated (they can't even be resized). In other words, in an expression the name of an array is constant.

In the third case, once again the use of the name of the array `how_big` refers to the address of the initial element of `how_big`. Which has type address of a *single* int. The problem is that `printf` cannot know how many ints live at this address.

The fourth case has a similar explanation. The name of an array when used in an expression (in this case combined with the subscript operator [ ] and an integer) refers to the address of the initial element of that array. Once again, this has type address of a *single* int. The problem is that the subscript operator cannot know how many ints live in the array starting at this address.

QACPROG

Exercise: Array Operations

```c
int a[10] = { 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

int main(void)
{
    int b[10];
    size_t i;
    for (i = 0; i < 10; i++)
    {
        b[i] = 0;
    }

}
```

What initial value will elements of b have?

❶ How can "b=0" be achieved? – See Code!

❷ How can "b=a" be achieved?

❸ How can "printf(a)" be achieved?

❹ How can "scanf(a)" be achieved?

Unlike some versions of Basic, C does not provide any support for arrays as units of data. Everything has to be achieved through individual element access. It soon becomes second nature to a programmer to write a for loop to access the elements sequentially, especially as the initialisation and test parts encourage the discipline for bound checking.

The loop shown below increments the elements of the integer array b:

```c
for (i = 0; i < 10; i++)
    b[i]++;
```

Note that the strict '<' is used in preference to <=. This is preparing the way for maintenance. The value 10 can be easily changed throughout if necessary. Note also the precedence of the array element access operator []. It has a higher precedence than the ++ operator, so no bracketing is required.

The for loop construct, as used above, virtually guarantees correct bounds checking.

Solutions:

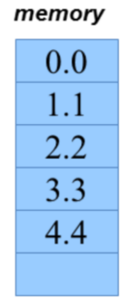
```c
for (i = 0; i < 10; i++)          /*  b = a */
    b[i] = a[i];

for (i = 0; i < 10; i++)          /*  printf(a) */
    printf("%d ", a[i]);

for (i = 0; i < 10; i++)          /*  scanf(a) */
    scanf("%d", &a[i]);
```

QACPROG

Array Arguments

- ***Whole* arrays *cannot* be used as function arguments**
  - Copying a whole array is a whole array operation!
  - Array arguments decay into the address of the initial array element
  - The called function *can* access and alter any element

```
void change(double []);
int main(void)
{
      double vector[] =
            { 0.0, 1.1, 2.2, 3.3, 4.4 };
      change(vector);
      printf("%f %f", vector[0], vector[3]);
      return 0;
}
void change(double da[])
{
      da[0] = 3.14;
      da[3] = da[2] + da[4];
}
```



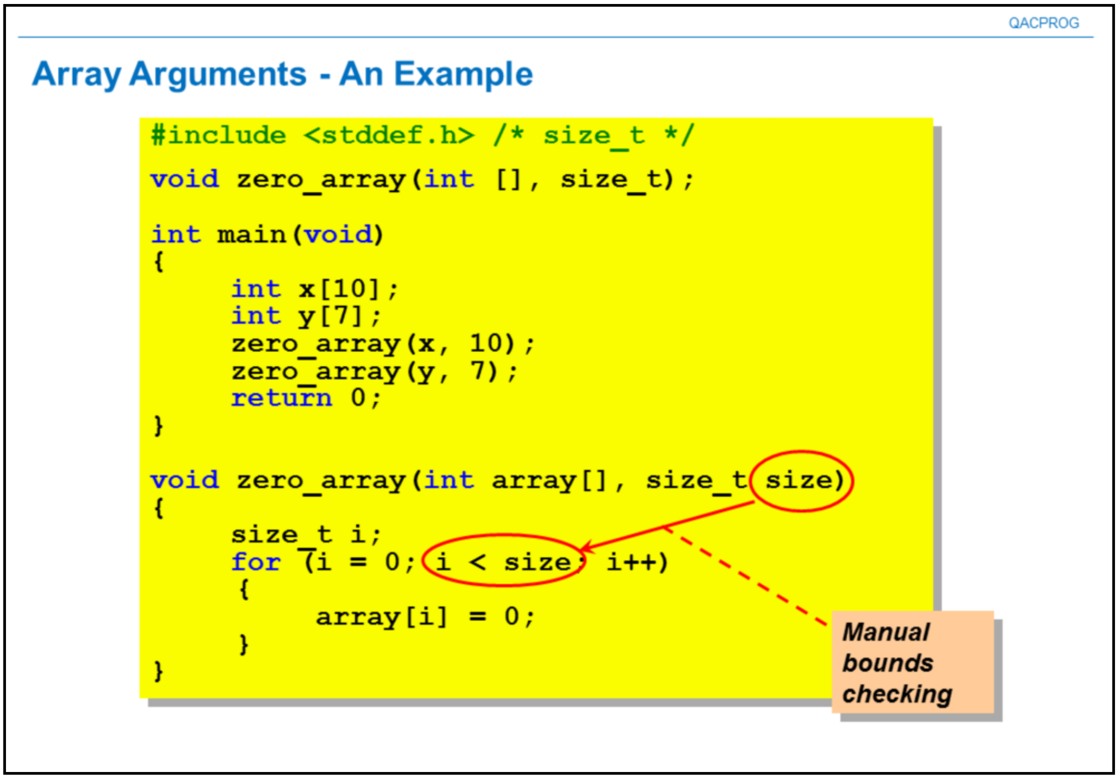
The way arrays are passed into functions in C is a major anomaly of the language. Arrays are passed by reference. This is due to C's mechanism of naming the array. The array name, as declared by the programmer, is not organised in the same way as the scalar data names. It is an address. When passed to a function, it is passed as a reference, i.e. the function is given indirect access to the data held in that array.

Note the prototype. Its single argument is of type `double[]`. Some programmers prefer to include a dummy parameter name to give, for example, `double table[]`. This is possibly a more readable expression, which can be read as 'table is an array of `doubles`'.

The function call is straightforward enough; the name of the array is supplied as the argument.

The definition implies that we are passing an array of `doubles` into a local `double` array called `da`. This local array is then accessing its elements, but this is not the true story. `da` is certainly local, but it is not an array. However, using the array-element access operator `[]`, the elements we are accessing are those of the array passed into the function, i.e. `vector`. The example shown above does not indicate the true nature of what is going on; a knowledge of pointers and how they work with arrays and functions is required. This is covered in the *Pointers and Arrays* chapter.

The example shown above is not very robust and is potentially very dangerous. What if vector had three and not five elements? There are no bound checks to control the potentially dangerous last statement. Sizing information should be made available to this function. The best way is to provide a second argument...



This example illustrates the way in which bound checking can be achieved. Integrity is now in the hands of the caller, i.e. the owner of the data.

We could generalise this solution further by passing the value to set the array elements to as another function parameter.

```
void set_array(int array[], size_t size, int new_value)
{
    size_t index;
    for (index = 0; index < size; index++)
    {
        array[index] = new_value;
    }
}
```

Having done this we could replace all calls to zero_array with calls to set_array and pass an extra zero argument. However, a much better solution is to leave zero_array as an available specialised version of set_array:

```
void zero_array(int array[], size_t size)
{
    set_array(array, size, 0);
}
```

Finally, we can avoid duplicating the size of the arrays in the function calls:

```
zero_array(x, sizeof(x) / sizeof(x[0]));
zero_array(y, sizeof(y) / sizeof(y[0]));
```

QACPROG

Exercise: Array Manipulation

Implement the missing function body

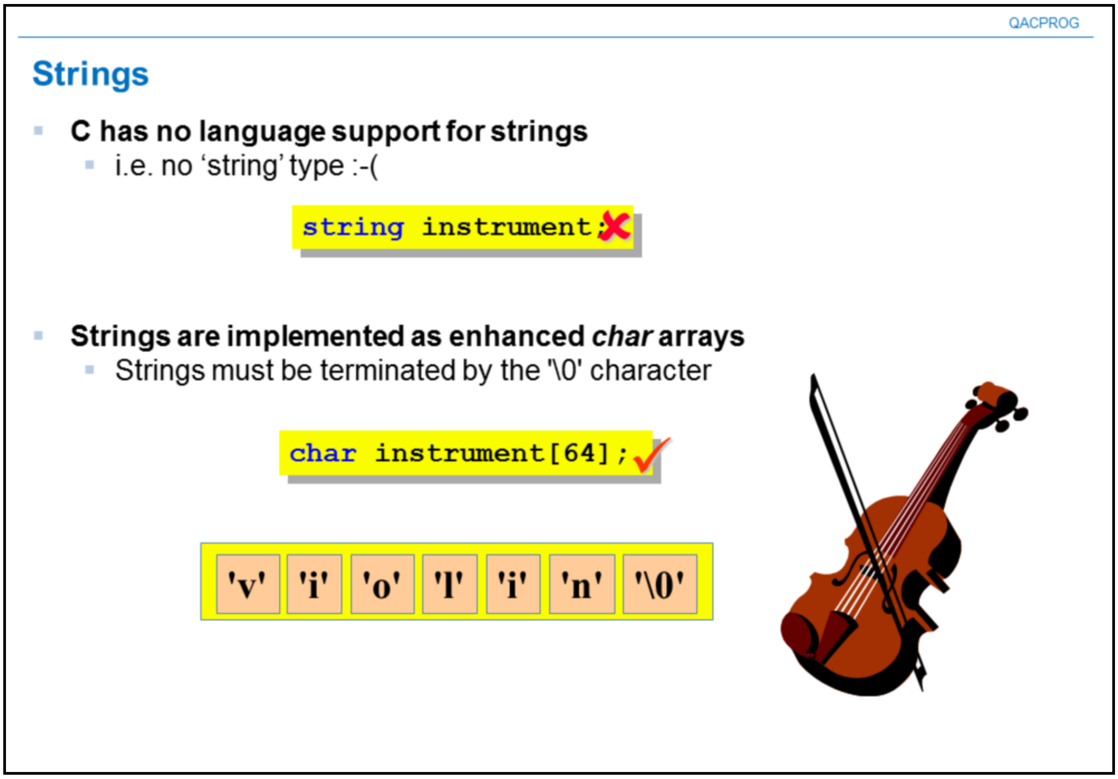
```
long add_up_int_array(int a[], size_t size);

int main(void)
{
      int squares[] = { 1, 4, 9, 16, 25 };
      long total = 0;
      total = add_up_int_array(squares, 5);
      printf("Total is %ld\n", total);
      return 0;
}

long add_up_int_array(int a[], size_t size)
{



}
```

The function `add_up_int_array` is designed to take a reference to an array of integers together with sizing information. It returns the sum of the integer elements in the array. Here is a possible implementation:

```
long add_up_int_array(int a[], size_t size)
{
      long result = 0;
      size_t index;
      for (index = 0; index < size; index++)
      {
            result += a[index];
      }
      return result;
}
```

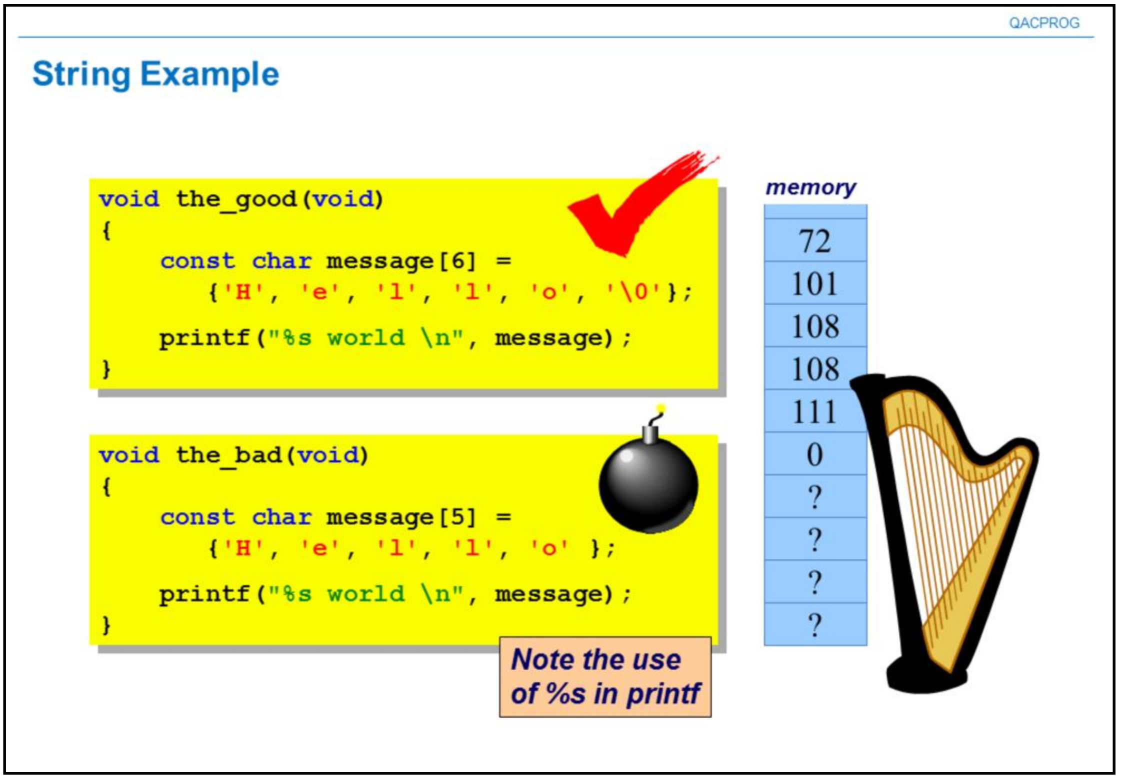
There are many ways to implement this simple algorithm but only one sensible interface. Note that the definition of add_up_int_array does not modify any of the elements inside the array. We can specify this in the code using `const`:

```
long add_up_int_array(const int a[], size_t size)
{
      ...
}
```



In order to use strings in C, the array must terminate with the special sentinel character, `'\0'`. This was mentioned in the chapter covering the `char` data type. Internally, it is the zero code; the character with ASCII value `0`.

When a string is declared, it *has* to be declared as an array of `char`. The size information must take the `'\0'` char into account, i.e. one extra char is required.



A string can be initialised just like any other array, but remember to put the `'\0'` in at the end.

Some terminology is included here:

> The null character or null terminator is the char `'\0'`.
> A string is a null-terminated char array.
> The length of a string is the number of characters up to, but not including, the null character.

The `printf` function supports strings; `%s` can be used to output a string; `scanf` provides the corresponding support for string input.

QACPROG

String Literals

- **C provides a string literal syntax**
  - The literal form is delimited with double quotes
  - A string literal may be used for initialisation

```
char message[] = { 'H','e','l','l','o','\0' };
```
cumbersome

```
char message[] = { "Hello" };
```
compiler adds trailing '\0' character

```
char message[] = "Hello";
```
braces are optional

- **Treat string literals as read-only character arrays**

```
void the_ugly(char a[])
{
    a[0] = 'J';
}
the_ugly("Hello");
```

```
void the_safe(const char a[])
{
    a[0] = 'J';
}
the_safe("Hello");
```

Initialising a `char` array with a long sequence of chars in an *aggregate initialiser list* is tedious. It is also error prone, since *you* have to remember to add the trailing null character. Fortunately there is a shorthand - constant strings (or string literals) can be created using double quotation marks. The characters are simply grouped together between the opening and closing double quotes. The resulting syntax is very similar to regular prose (like this) and we are relieved from two error-prone chores: typing in all those cumbersome and repetitive single quotes and commas; and typing in the trailing nul character.

When initialising an array of characters from a string literal the braces in the aggregate initialiser list are optional and are commonly omitted. Hence:

```
char message[] = "Hello"
```

This declares `message` as an array of characters, which is initialised as a copy of the constant string literal `"Hello"`. Remember, `"Hello"` contains an implicit, invisible trailing nul character. In other words, the string literal `"Hello"` contains *six* characters. This means that `message` is an array of *six* characters and contains the characters H,e,l,l,o, *and* a terminating nul character. If you accidentally forget about this terminating nul character and write:

```
char message[5] = "Hello";
```

it will *still* compile! In this case the array is being initialised as a copy of the first *five* characters of the *six* characters in the string literal "Hello". The trailing nul character is *not* copied. Debug time...

String literals have already been seen in the `printf` and `scanf` functions.

QACPROG

String Library Support - <stdio.h>

<stdio.h>

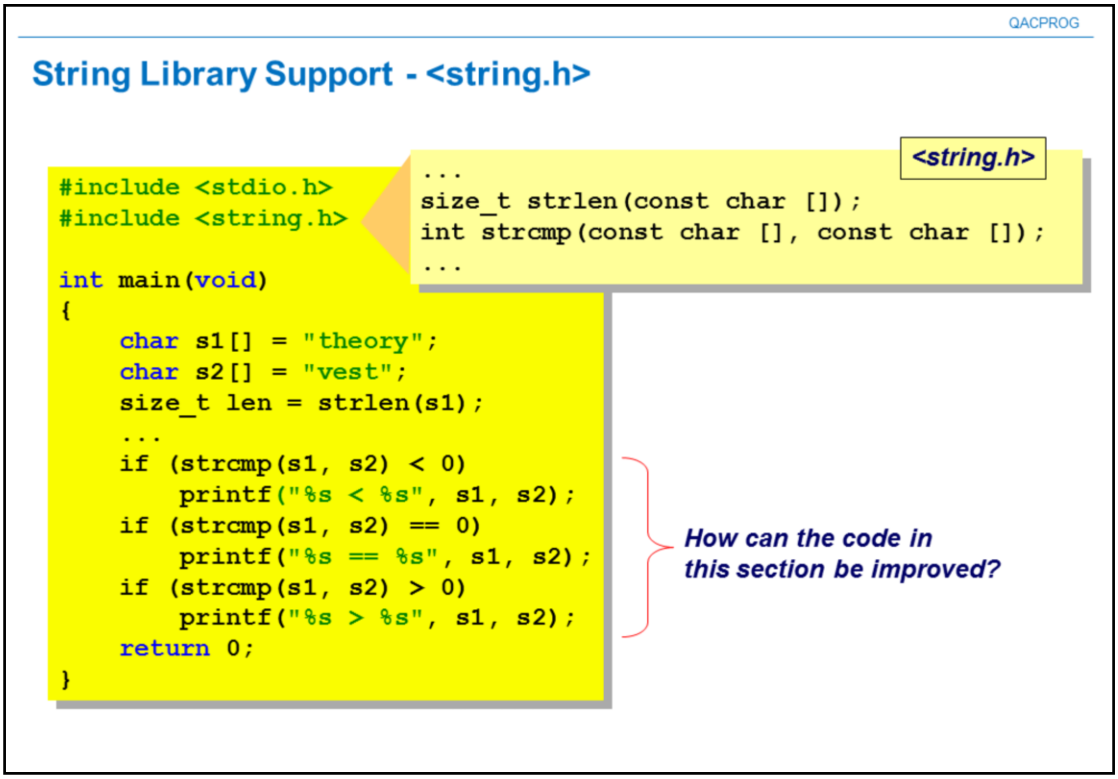
```
int sprintf(char [], const char [], …);
int sscanf(const char [], const char [], …);
```

```
#include <stdio.h>

int main(void)
{
    char source[32 + 1] = "20-10-2010";
    char dest[32 + 1];
    char dFormat[8 + 1]="%d-%d-%d";
    int d = 19, m = 11, y = 1999;

    sprintf(dest, dFormat, y, m, d);

    sscanf(source, dFormat, &d, &m, &y);

    return 0;
}
```

`dest` is now "1999-11-19"

sets 20 to d, etc …

We have already seen that `printf()` and `scanf()` functions support string output (to `stdout`) and input (from `stdin`) using the `%s` format specifier. The `'s'` versions perform the same task but with `stdout/stdin` replaced by strings. The string(s) in question being supplied as the first argument. One UNIX criterion is that 'everything' is a device, e.g. `stdout` and `stdin` our pre-set devices representing the screen and keyboard respectively (more in the I/O chapter later). So, here we can output to any sink and inputting from any source, including string buffers. All the format specifiers from `printf/scanf` may be used with these two functions.

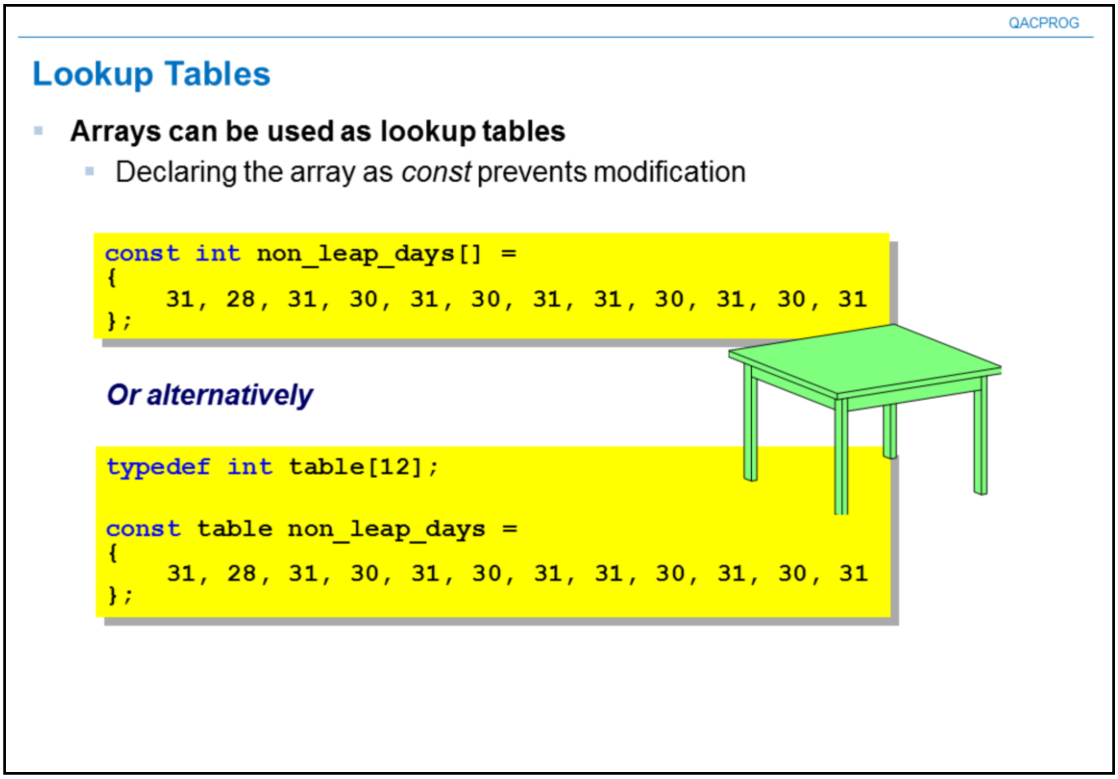


By far the greatest support for strings comes from the library. The library provides fifteen standard functions which manipulate strings and five standard functions which indirectly handle strings as arrays of characters. You might like to find `<string.h>` and view the function declarations it contains.

The two functions illustrated above are probably the simplest: `strlen` returns the length of a string (not including the terminating nul character), `strcmp` provides the functionality of string comparison.

The code section in the slide can be improved by avoiding the recalculation of the result of `strcmp` and by noticing that the three outcomes are mutually exclusive:

```
int cmp = strcmp(s1, s2);

if (cmp < 0)
    printf("%s < %s", s1, s2);
else if (cmp == 0)
    printf("%s == %s", s1, s2);
else
    printf("%s > %s", s1, s2);
```



Consider this code which calculates the number of days in a month using explicit control flow.

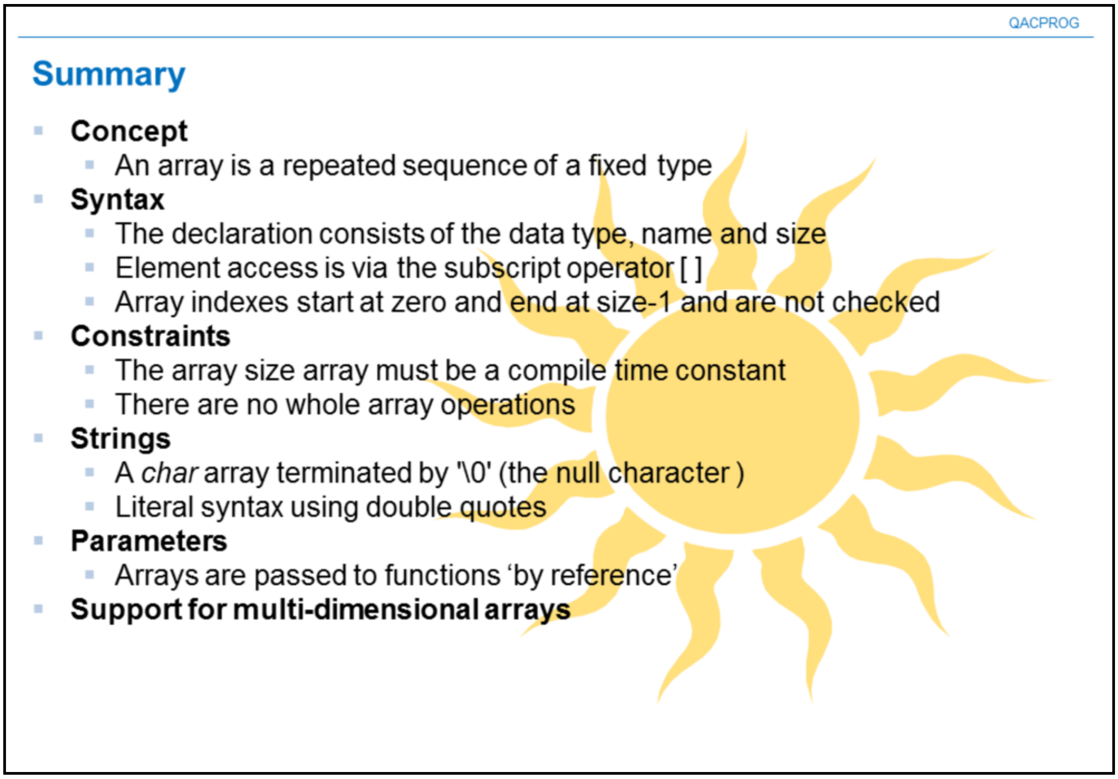
```
enum month_type { jan,feb,mar, ... ,oct,nov,dec };
if (month == jan)
   days_in_month = 31;
else if (month == feb)
   days_in_month = 28;
else if (month == mar)
   days_in_month = 31;
else ...
```

It is hard to see that the literal values are related because they are not grouped together. This separation makes the code hard to maintain. Lookup tables provide a much cleaner, more elegant solution. Using the table from the slide above we can simply write:

```
days_in_month = non_leap_days[month];
```

In an earlier chapter it was stated that there is no way to automatically print `enum` values. As `enum` constants have integer values these could be used to lookup a printable string in an array (this will become clearer when we have covered pointers):

```
const char * month_names[] =
{
   "january", "february", ... ,"november", "december"
};
printf("%s", month_names[month]);
```



The story of the array is by no means over. The course reveals the 'true' story behind the array in a later chapter. However, this chapter is essential, because it describes the concept of an array as seen by the outside world. Only the C programmer needs to know any different.

QACPROG

Common Pitfalls

- **Array size**
  - Must be a true compile time constant

```
const int size = 4;
double no[size];
double yes[4];
```

- **Array assignment**
  - Won't work
  - Can't use aggregate initialiser list

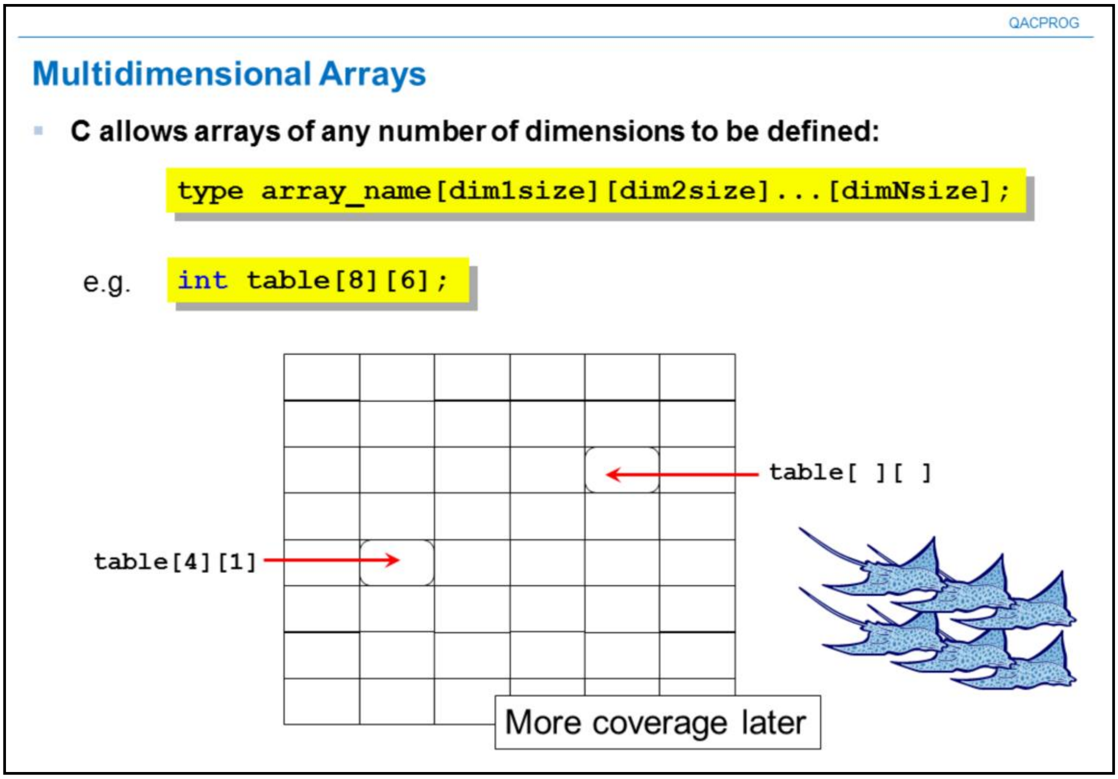
```
int lhs[4], rhs[4] = { 0,1,2,3 };
lhs = rhs;
lhs = { 0,1,2,3 };
```

- **Array access**
  - Index is from 0 not 1

```
int array[42];
array[42];
```

- **String literals**
  - Syntax uses double not single quotes

```
char one[] = 'Hello';
char two[] = "Hello";
```



A two-dimensional array is easy to illustrate. However, it is probably worthwhile thinking of an `N`-dimensional array of ints in terms of an (`N-1`)-dimensional array of arrays of `int`s. The example shown above can be thought of as:

> An array of 8 items, each of which is an array of 6 `int`s.

As mentioned earlier in the chapter, the syntax does not help us to think in this way. It is advisable to return to this topic once pointers have been mastered. The majority of applications tend to use pointers to declare multidimensional arrays and to access the data.

QACPROG

Initialising Multidimensional Arrays

- **Multidimensional arrays can also be initialised**

```
int results[100][5] =
{
    { 75, 95, 90, 84, 80 },    /* 5 marks for student 0 */
    { 100, 99, 100, 98, 99 }, /* 5 marks for student 1 */
    ...
    ...
    ...
    { 80, 67, 85, 79, 75 }     /* 5 marks for student 99 */
};

double coordinates[][2] =      /* coordinates of 3 points */
{
    { -2.5, 2.71 },            /* (x,y) point 0 */
    { 0.0, 0.0 },              /* (x,y) point 1 */
    { 4.0, 12.03 }             /* (x,y) point 2 */
};
```

More coverage later

Initialisation of multidimensional arrays is not easy to portray. Again, we have opted for the easy example of two dimensions.

Note the use of the braces to clarify the process. It also illustrates the ‘array within an array’ viewpoint.

Note also that the compiler can calculate the row count for the second array from the initialisation.