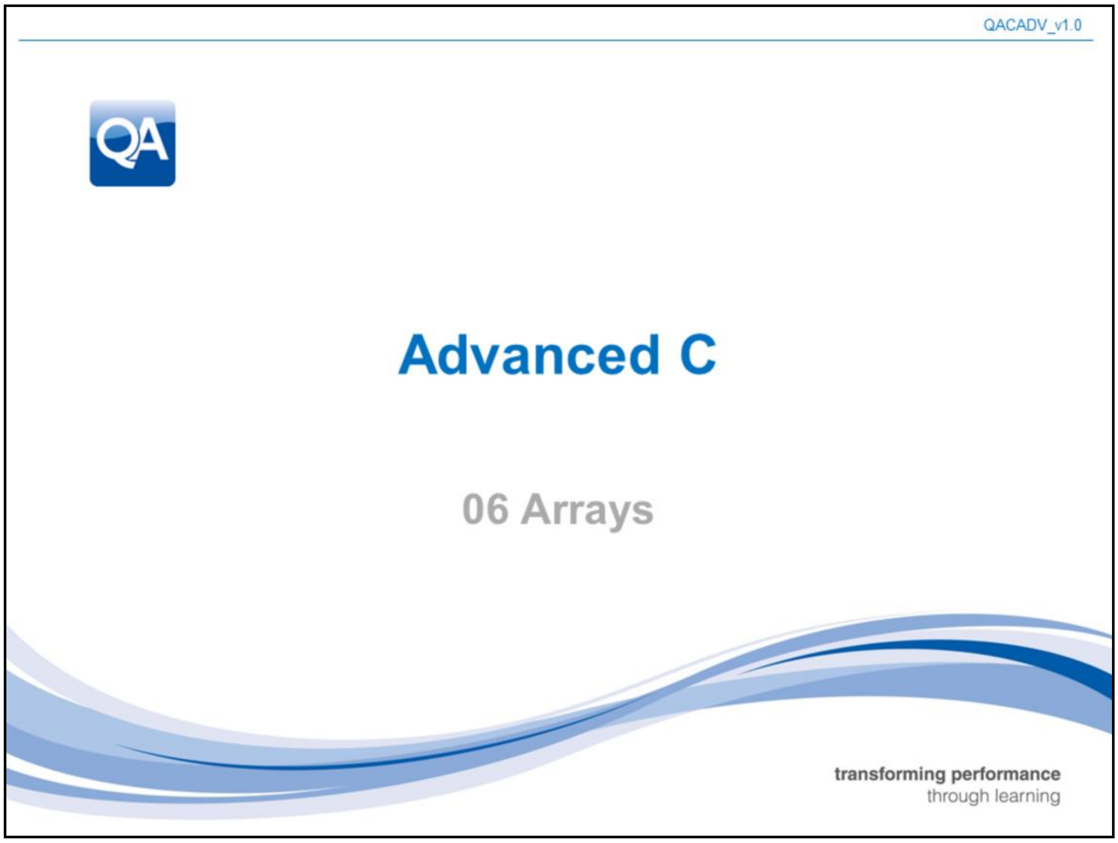
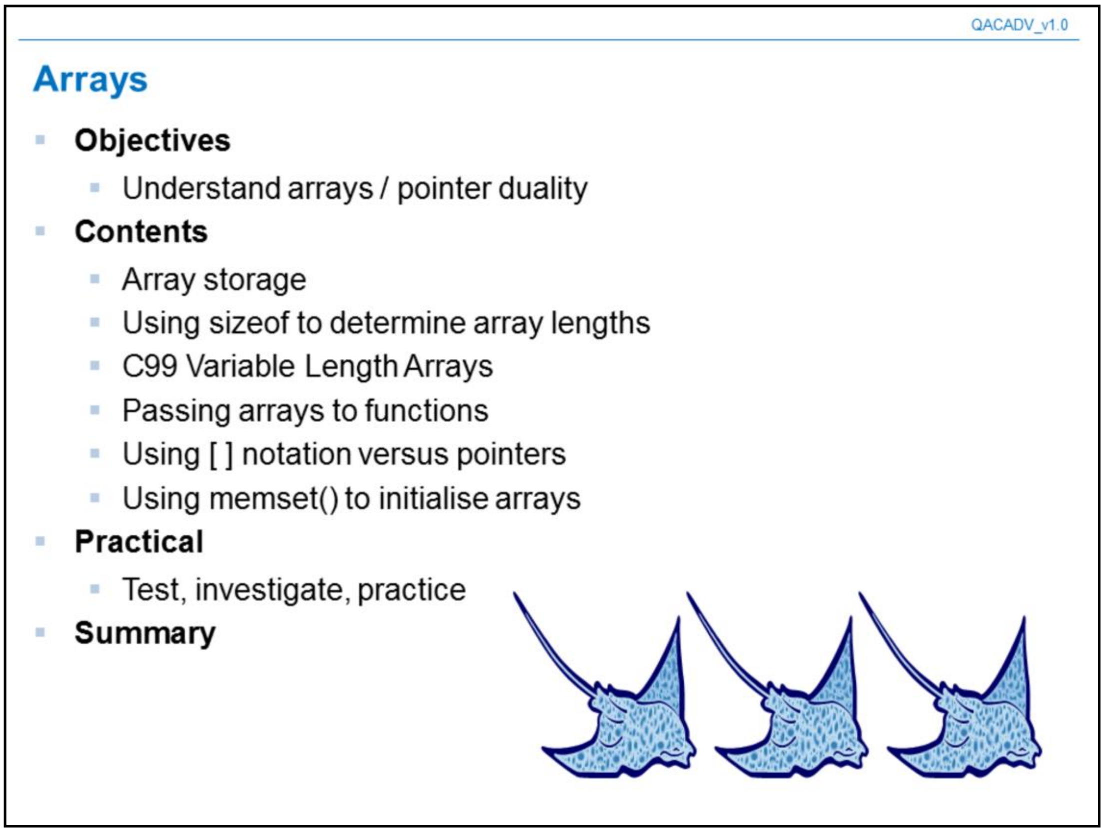


QACADV_v1.0

Advanced C

06 Arrays

transforming performance
through learning


QACADV_v1.0

Arrays

- **Objectives**
  - Understand arrays / pointer duality
- **Contents**
  - Array storage
  - Using sizeof to determine array lengths
  - C99 Variable Length Arrays
  - Passing arrays to functions
  - Using [ ] notation versus pointers
  - Using memset() to initialise arrays
- **Practical**
  - Test, investigate, practice
- **Summary**

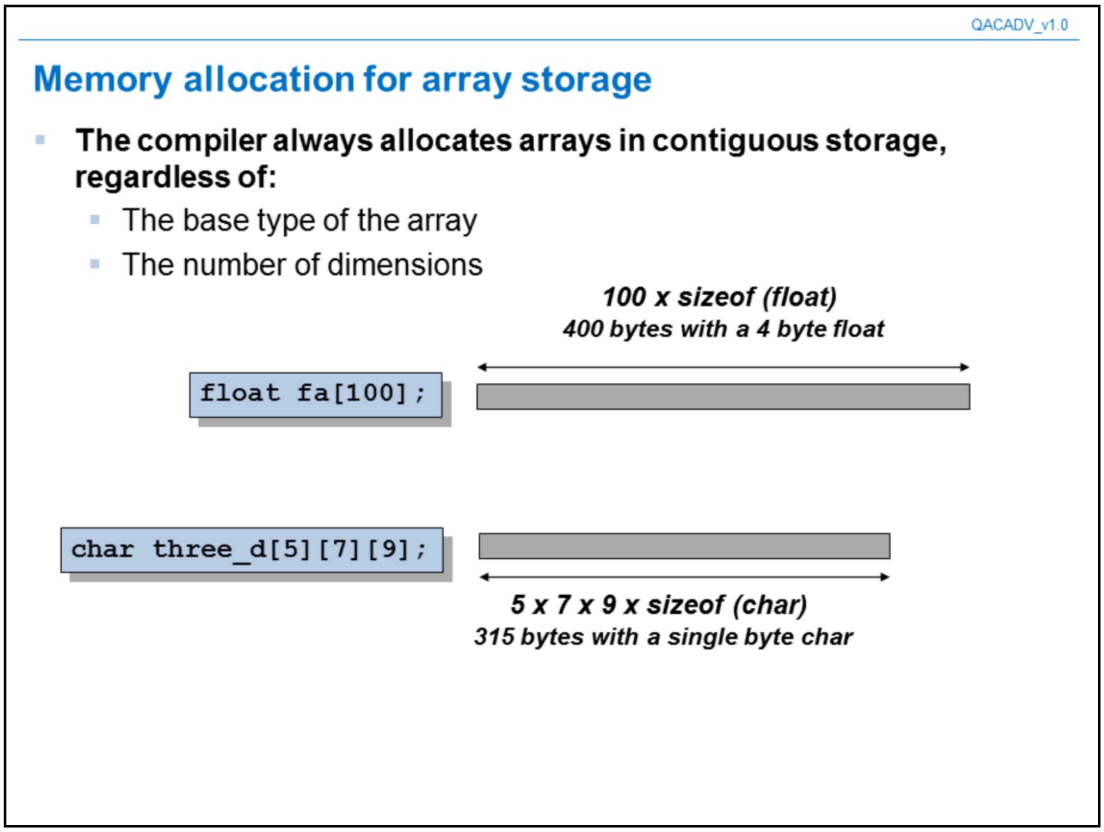
Arrays in C are very closely related to pointers, and many program errors stem from misconceptions and incorrect assumptions. The aim of this chapter is to describe the rules clearly and unambiguously, introducing new techniques and useful tricks as we do so.

We start with the `sizeof` operator. We show how it can be used to determine the number of elements in an array, and illustrate cases where the technique isn't quite adequate.

The chapter goes on to discuss the way in which arrays are passed into a function and the reasons why C does it this way.

A good part of the chapter is dedicated to the way the compiler converts array notation into pointer and offset notation. We give the rules first of all, then present a number of alternative ways of processing arrays. We discuss the pros and cons of each way in terms of performance.

The chapter closes with a look at the `memset()` function, an extremely useful and fast function for initialising arrays to zero. There are some limitations to this function, however, and we shall mention these during our discussion.



QACADV_v1.0

Memory allocation for array storage

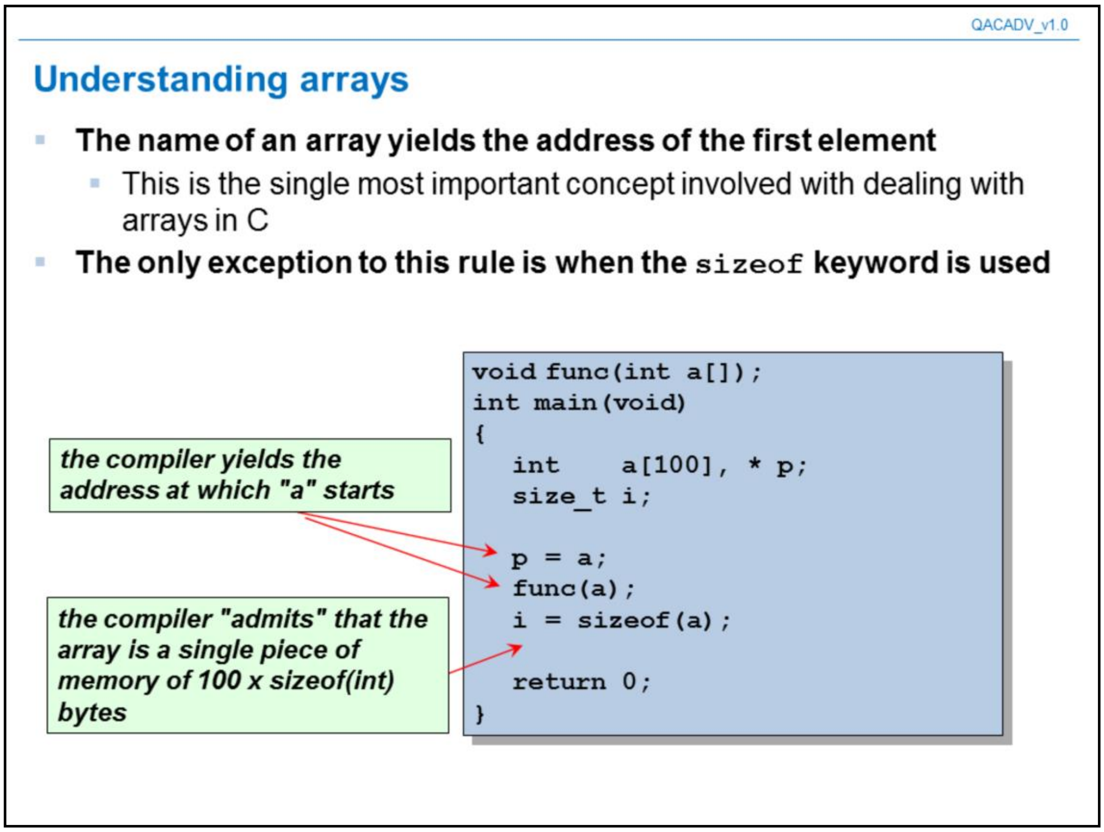
- **The compiler always allocates arrays in contiguous storage, regardless of:**
  - The base type of the array
  - The number of dimensions

100 x sizeof (float)
400 bytes with a 4 byte float
float fa[100];
char three_d[5][7][9];
5 x 7 x 9 x sizeof (char)
315 bytes with a single byte char

Whenever the compiler allocates the storage for an array, it always does so using a single contiguous area of memory.

Thus, an array of 10 integers would have a size of 10 times `sizeof(int)`, and an array of 10 floating point numbers would have a size of 10 times `sizeof(float)`.

The same is true of, for example, a 5 by 12 by 9 by 7 array of `double`s. This would be allocated within a single piece of memory whose size is 3780 times `sizeof(double)`.



QACADV_v1.0

Understanding arrays

- **The name of an array yields the address of the first element**
  - This is the single most important concept involved with dealing with arrays in C
- **The only exception to this rule is when the `sizeof` keyword is used**

```
void func(int a[]);
int main(void)
{
   int    a[100], * p;
   size_t i;

   p = a;
   func(a);
   i = sizeof(a);

   return 0;
}
```

the compiler yields the address at which "a" starts
the compiler "admits" that the array is a single piece of memory of 100 x sizeof(int) bytes

The single most important thing to understand about arrays in C is that when the name of an array is used the compiler yields the address at which the array starts. Thus for the declarations:

```
int  a[10];
int* p;
```

the two statements: `p = a;`

and `p = &a[0];`

are both identical, placing the address of the first (zeroth) element into the pointer p. If the array is passed down to a function, again the address at which the array starts is passed down. Also, the statements:

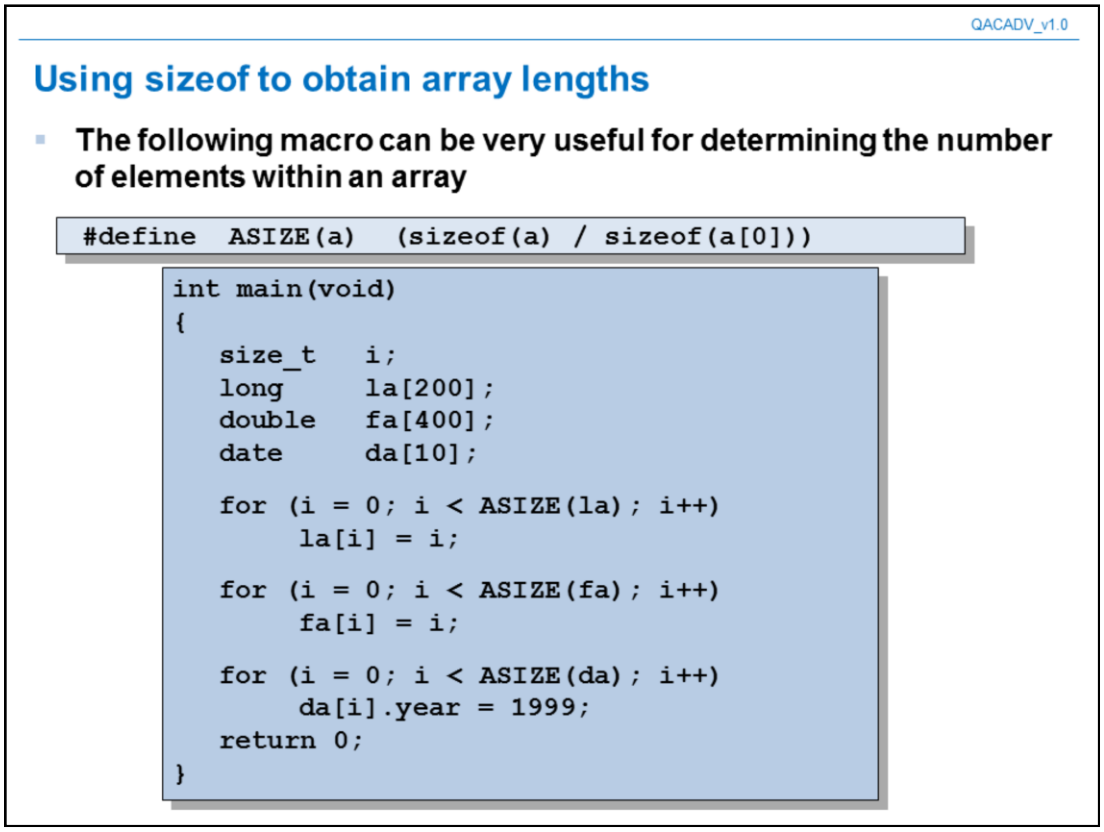
`*p`

and `*a`

and `a[0]`

will all yield the integer stored at the first location (element zero) in the array.

It is not strictly correct to say that, in this example, p and a are "the same". They refer to the same location in memory, but are distinctly different types. For example, `p++` works, pointer arithmetic is allowed, but `a++` does not because an array is not a valid l-value.


QACADV_v1.0

Using sizeof to obtain array lengths

- The following macro can be very useful for determining the number of elements within an array

```
#define  ASIZE(a)  (sizeof(a) / sizeof(a[0]))
```

```
int main(void)
{
   size_t   i;
   long     la[200];
   double   fa[400];
   date     da[10];

   for (i = 0; i < ASIZE(la); i++)
        la[i] = i;

   for (i = 0; i < ASIZE(fa); i++)
        fa[i] = i;

   for (i = 0; i < ASIZE(da); i++)
        da[i].year = 1999;
   return 0;
}
```

The macro given above will yield the size of an array of any type. The array `f` of 400 doubles declared above has a total size of 400 x 8 = 3200 bytes. The size of the zeroth element (as well that of any other element) is 8 bytes. 3200 / 8 brings back the 400 which is the number of elements in the array.

The `ASIZE` macro will be expanded in the test part of the `for` loops in the above examples. At first glance, this might seem to be quite an inefficient scheme, since a division appears to be taking place on each iteration. However, any modern C compiler worth its salt would recognise the macro expansion as an invariant expression and perform the calculation once only, before the first loop iteration.

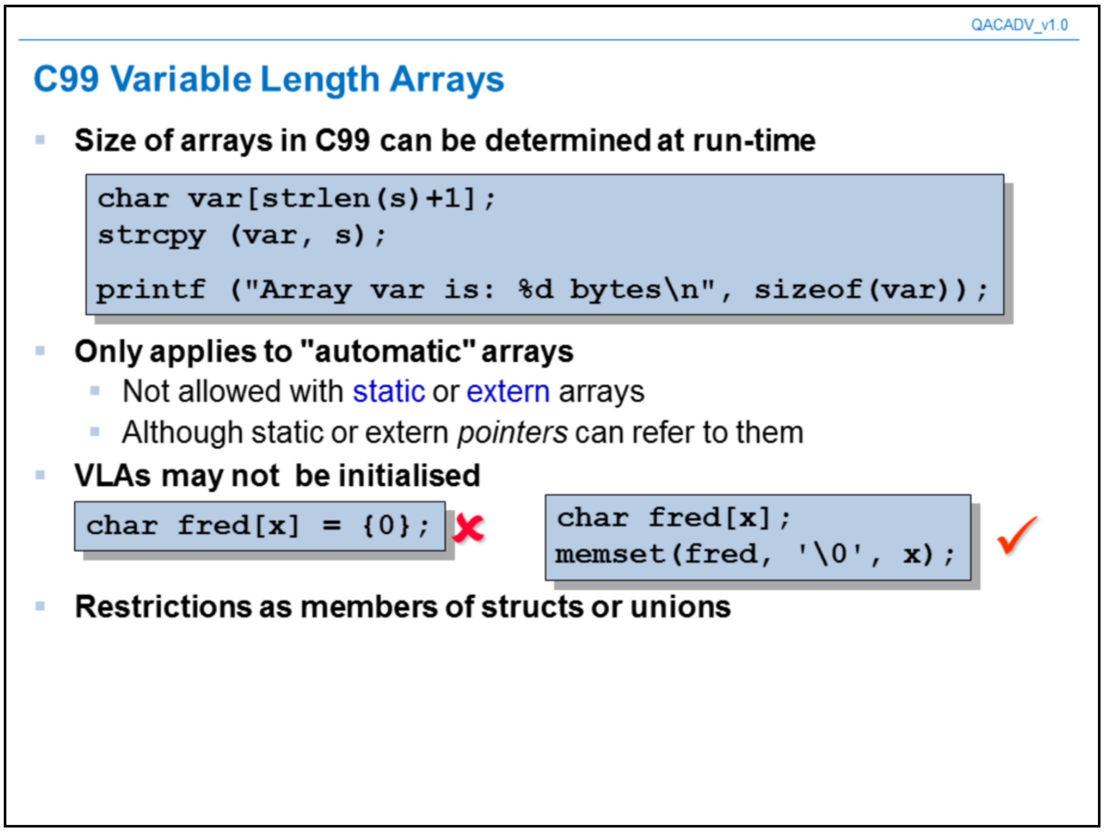
Also note that the macro could have been written as:

```
#define  ASIZE(a)      (sizeof(a) / sizeof(a[52]))
```

or even as:

```
#define  ASIZE(a)      (sizeof(a) / sizeof(*a))
```

The first of these two alternative definitions looks decidedly dubious. What if the macro is applied to an array with only 10 elements? However, the compiler does not have to find element 52 and figure out how large it is, since it is exactly the same size as all the other elements.



QACADV_v1.0

C99 Variable Length Arrays

- Size of arrays in C99 can be determined at run-time

```
char var[strlen(s)+1];
strcpy (var, s);

printf ("Array var is: %d bytes\n", sizeof(var));
```

- Only applies to "automatic" arrays
  - Not allowed with static or extern arrays
  - Although static or extern *pointers* can refer to them
- VLAs may not be initialised

```
char fred[x] = {0};
```
✗

```
char fred[x];
memset(fred, '\0', x);
```
✓

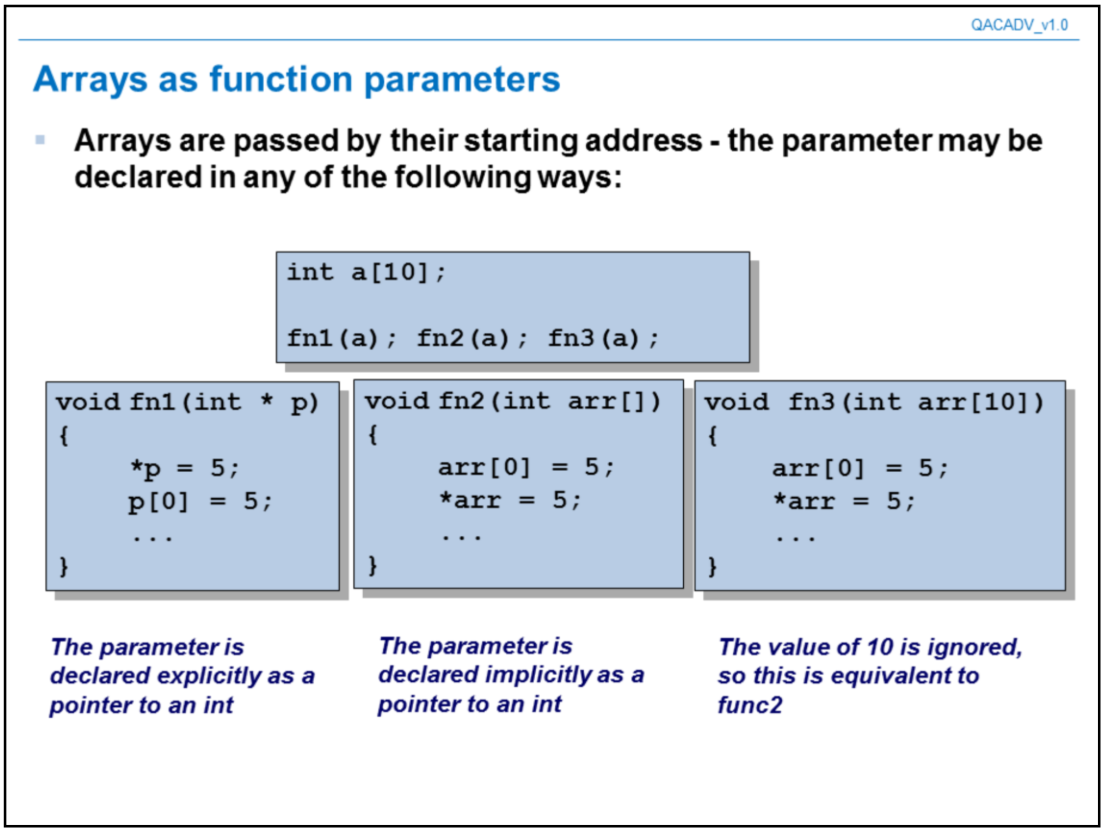
- Restrictions as members of structs or unions

A fundamental shift in the use of arrays occurred with the introduction of Variable Length Arrays (VLAs) with C99. Previously, many programmers rarely used arrays, since they were fixed length containers. They used malloc() instead to create their own VLAs, and paid dearly with the associated memory leaks! Variable Length Arrays should go some way to making C programs more reliable.

C99 VLAs are simple to use, just use a variable as the size when defining the array. The array is placed on the stack and the memory freed at the end of the block. There are a few (understandable) restrictions. Only automatic arrays (allocated on the stack) can be variable length. This is reasonable since linkers usually have to pre-allocate memory for off-stack data. VLAs are also not allowed inside structs or unions, but are allowed at the end. Again this is reasonable since the offset of other members would not be known. Some compilers (notably GNU gcc) do allow VLAs in structs. Conversely the `[*]` notation for function declarations are not implemented in all C99 compilers, including gcc.

The value returned by **sizeof** might be correct, but unfortunately the C99 standard makes the support of VLAs by **sizeof** optional.

VLAs may not be initialised in the conventional way because that is an operation performed at compile time. Using a variable for an array size means initialisation can only be performed at run-time, possibly using a library routine like memset().


QACADV_v1.0

Arrays as function parameters

- Arrays are passed by their starting address - the parameter may be declared in any of the following ways:

```
int a[10];

fn1(a); fn2(a); fn3(a);
```

```
void fn1(int * p)
{
     *p = 5;
     p[0] = 5;
     ...
}
```

The parameter is declared explicitly as a pointer to an int

```
void fn2(int arr[])
{
     arr[0] = 5;
     *arr = 5;
     ...
}
```

The parameter is declared implicitly as a pointer to an int

```
void fn3(int arr[10])
{
     arr[0] = 5;
     *arr = 5;
     ...
}
```

The value of 10 is ignored, so this is equivalent to func2

For efficiency reasons, the C compiler never passes arrays by value to functions. It would involve too many CPU cycles to copy the array to and from the stack (arrays are not good candidates for passing across in registers).

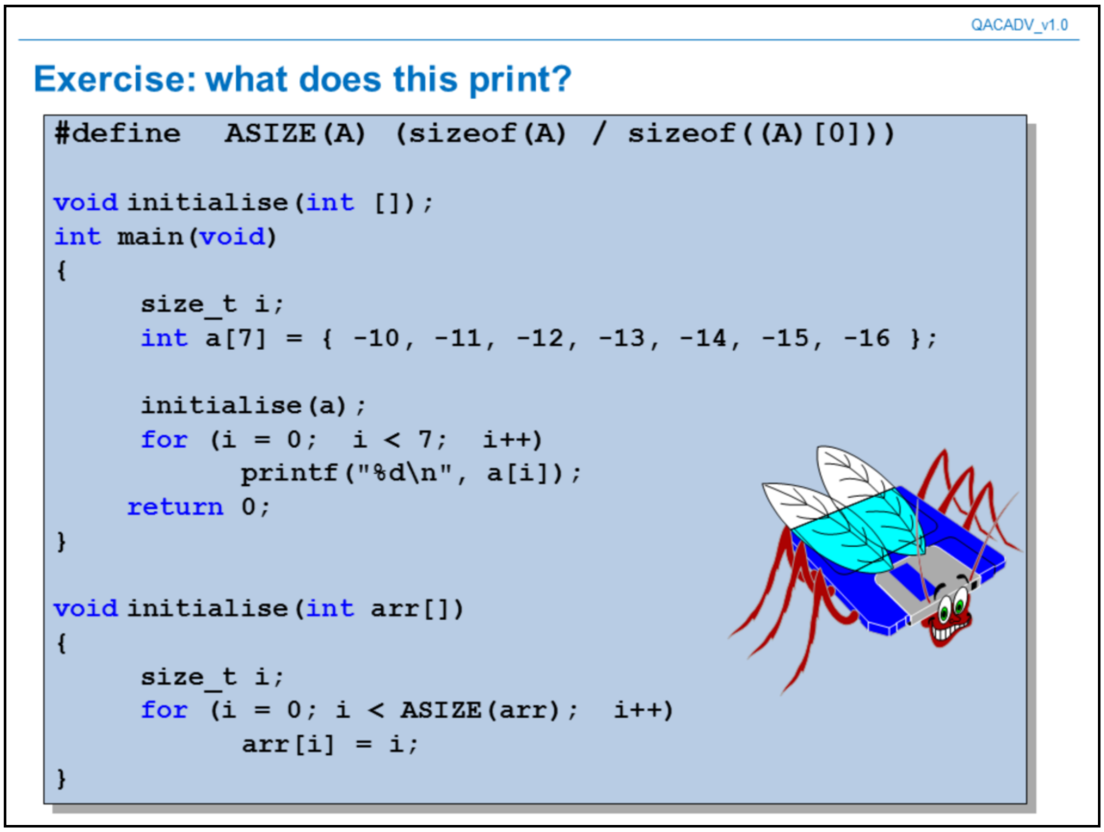
In the example shown above, the array is passed to each of the functions `func1`, `func2` and `func3`. Since `a` (without an index) is the name of an array, the compiler copies onto the stack the address at which the array starts.

The size of this address depends on the hardware architecture, the compiler, and the compiler options. Even for the small array of 10 integers, there is probably a saving. On a typical 32-bit machine, a 4-byte address is copied, rather than 40 bytes of array contents. Obviously, the larger the array, the greater the efficiency gain.

The formal parameter for the function can be declared in each of the ways shown. In the first example, `int *p` admits that what is being passed across is the address of an integer. The syntax `int arr[]`, although more obscure, says the same thing.

An integer value can be placed between the `[ ]` brackets, as in the `int arr[10]` example. The value 10 is completely ignored by the compiler. There is no additional bounds checking performed. It might as well be "100" (there would be neither warnings nor errors generated if it were 100 - the value is *entirely* ignored).

C99 allows the **`static`** keyword to be used with a function declaration, for example: **`void myfunc (int arr[static 10])`**. This (unfortunately) is only a hint to the optimiser, it does not perform any bounds check at compile or run-time.



QACADV_v1.0

Exercise: what does this print?

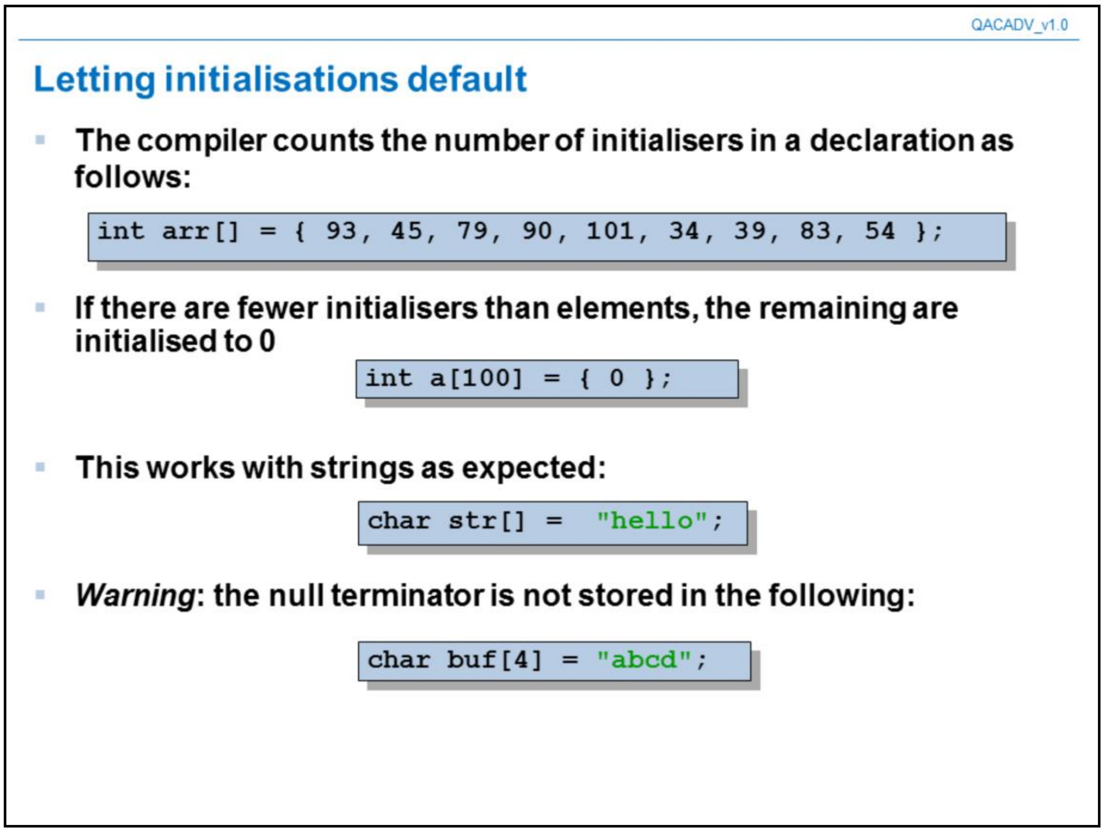
```
#define   ASIZE(A)  (sizeof(A)  /  sizeof((A)[0]))

void initialise(int []);
int main(void)
{
      size_t i;
      int a[7] = { -10, -11, -12, -13, -14, -15, -16 };

      initialise(a);
      for  (i = 0;  i < 7;  i++)
             printf("%d\n", a[i]);
     return 0;
}

void initialise(int arr[])
{
      size_t i;
      for  (i = 0; i < ASIZE(arr);  i++)
             arr[i] = i;
}
```

The program above shows that, unfortunately, the `ASIZE` macro is not infallible.

Within the function, the `arr` parameter appears to be an array but is in fact a pointer, as we have seen in our previous discussions. Therefore, the `ASIZE` macro expansion within the function effectively divides the size of a pointer by the size of an integer. This is certainly *not* the desired effect!

Thus, the `ASIZE` macro cannot be used for array parameters in a function. The only solution is to pass an extra parameter into the function which indicates the number of elements in the array:

```
void  initialise(int arr[], int num_elements);
```


QACADV_v1.0

Letting initialisations default

- The compiler counts the number of initialisers in a declaration as follows:

```
int arr[] = { 93, 45, 79, 90, 101, 34, 39, 83, 54 };
```

- If there are fewer initialisers than elements, the remaining are initialised to 0

```
int a[100] = { 0 };
```

- This works with strings as expected:

```
char str[] =  "hello";
```

- *Warning*: the null terminator is not stored in the following:

```
char buf[4] = "abcd";
```

If an array is declared with a set of initialising values, the compiler computes the length by counting the number of initialisers given. Thus, `arr` in the above example is created as an array of 9 integers. Note that the following declaration is exactly equivalent:

```
int arr[9] = { 93, 45, 79, 90, 101, 34, 39, 83, 54 };
```

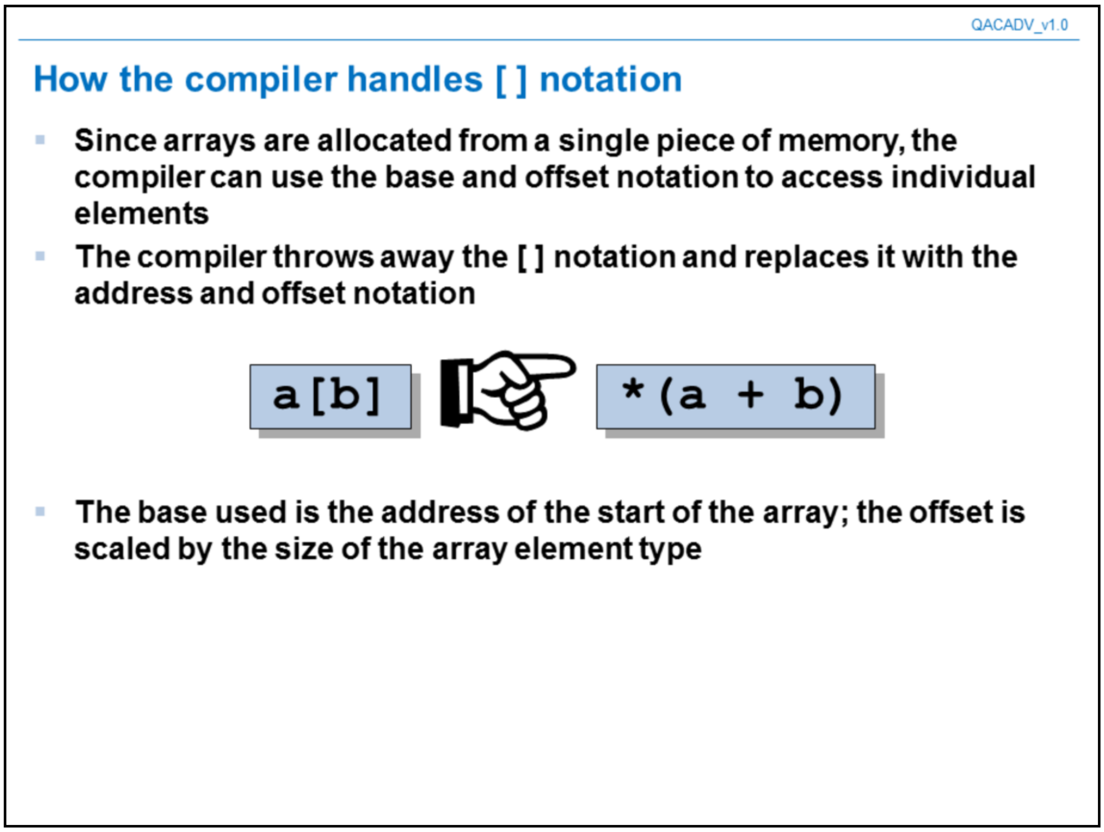
If you provide fewer initialisers than elements in the array (here you would have to specify the size), any remaining elements are initialised to zero. This is useful, since *all* 100 elements of the array `a` shown above are initialised to zero.

Character arrays are a special case. The declaration of `str` shown above is exactly equivalent to the following. Note that the null terminator is also included in the length, making the size of `str` 6 bytes:

```
char  str[ ] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

Although the declaration of `buf` shown in the slide looks as though it would be thrown out, the compiler stores just the four characters mentioned *and not the null terminator*. This does not make it a string. The declaration is equivalent to:

```
char  buf [4] = { 'a', 'b', 'c', 'd' };
```


QACADV_v1.0

How the compiler handles [ ] notation

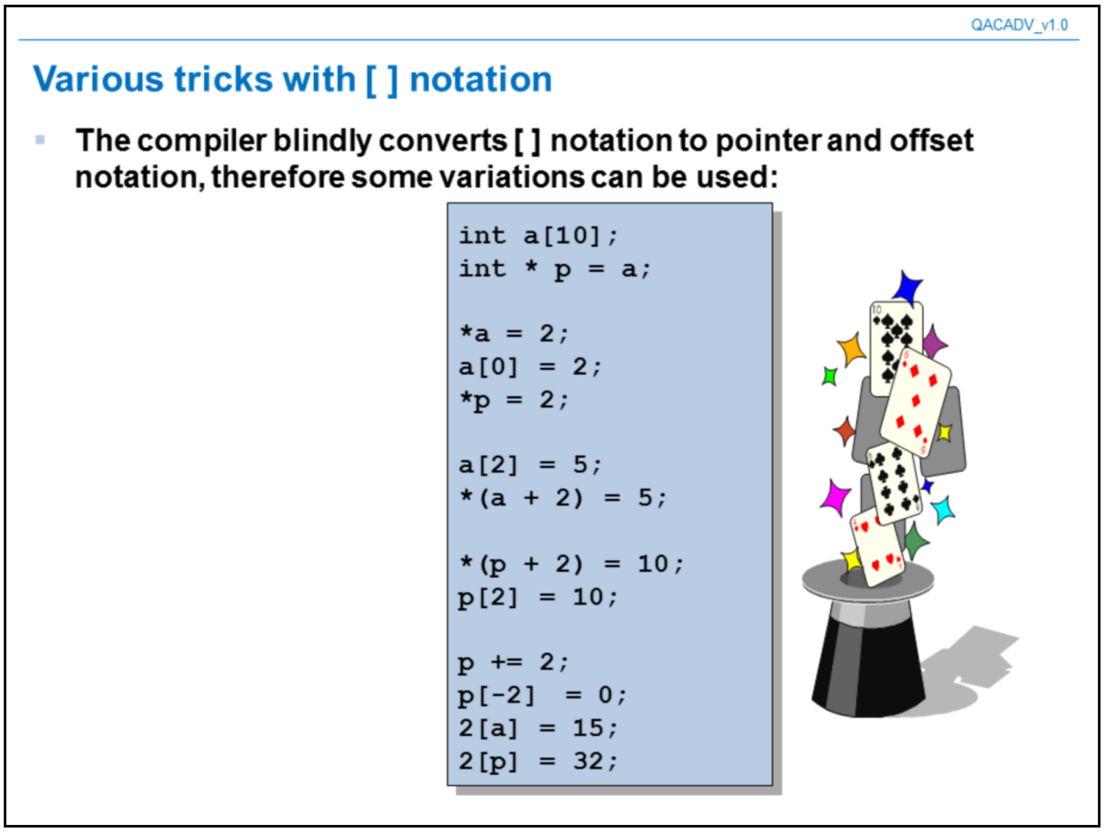
- **Since arrays are allocated from a single piece of memory, the compiler can use the base and offset notation to access individual elements**
- **The compiler throws away the [ ] notation and replaces it with the address and offset notation**

```
a[b]    *(a + b)
```

- **The base used is the address of the start of the array; the offset is scaled by the size of the array element type**

The compiler handles `[]` notation by changing it. In fact, it can be said that the compiler cannot handle arrays, but handles pointers rather well. Its solution is therefore to completely replace array notation with pointer notation.

For example, take an array of 10 `float`s and access the fifth element, `a[4]`. The compiler converts this array access into `*(a + 4)`. Now `a` appears on its own without a subscript, hence the compiler yields the address of the start of the array. Imagine this address is 1000. The `4` (our subscript) is scaled by the `sizeof` a `float`.



QACADV_v1.0

Various tricks with [ ] notation

- The compiler blindly converts [ ] notation to pointer and offset notation, therefore some variations can be used:

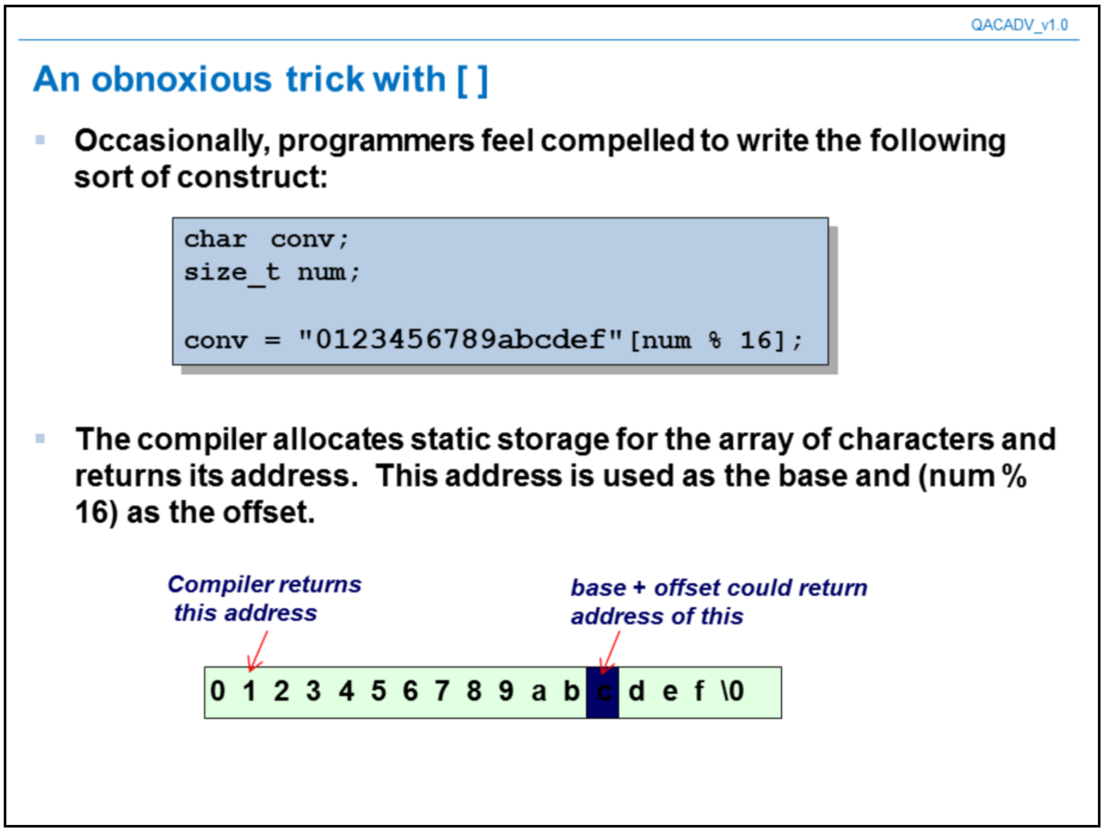
```
int a[10];
int * p = a;

*a = 2;
a[0] = 2;
*p = 2;

a[2] = 5;
*(a + 2) = 5;

*(p + 2) = 10;
p[2] = 10;

p += 2;
p[-2]  = 0;
2[a] = 15;
2[p] = 32;
```

The compiler will always convert from `[]` notation to pointer and offset notation. There is no advantage therefore in recoding array accesses of the form `a[2]` into `*(a+2)`. Clarity is lost and no efficiency is gained .

What looks slightly more surprising is a negative index used in an array access. Instead of scaling the index and moving "forward" in memory, the byte offset is used to move "backward" from the starting address.

What is even more surprising is seeing the index outside the brackets and the array name inside, as in `2[a]`. How it works is easily explained. If `a[2]` is converted to `*(a+2)` by the compiler, then `2[a]` must be converted into `*(2+a)`. Since + is a commutative operator, `a+2` is equivalent to `2+a`. Hence `a[2]` must be the same as `2[a]`.

Nevertheless, it is rather odd to be allowed to write an array access in such a way. For some assembler programmers, however, this may seem natural.



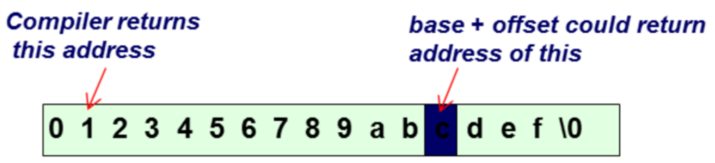
QACADV_v1.0

An obnoxious trick with [ ]

- Occasionally, programmers feel compelled to write the following sort of construct:

```
char  conv;
size_t num;

conv = "0123456789abcdef"[num % 16];
```

- The compiler allocates static storage for the array of characters and returns its address. This address is used as the base and (num % 16) as the offset.

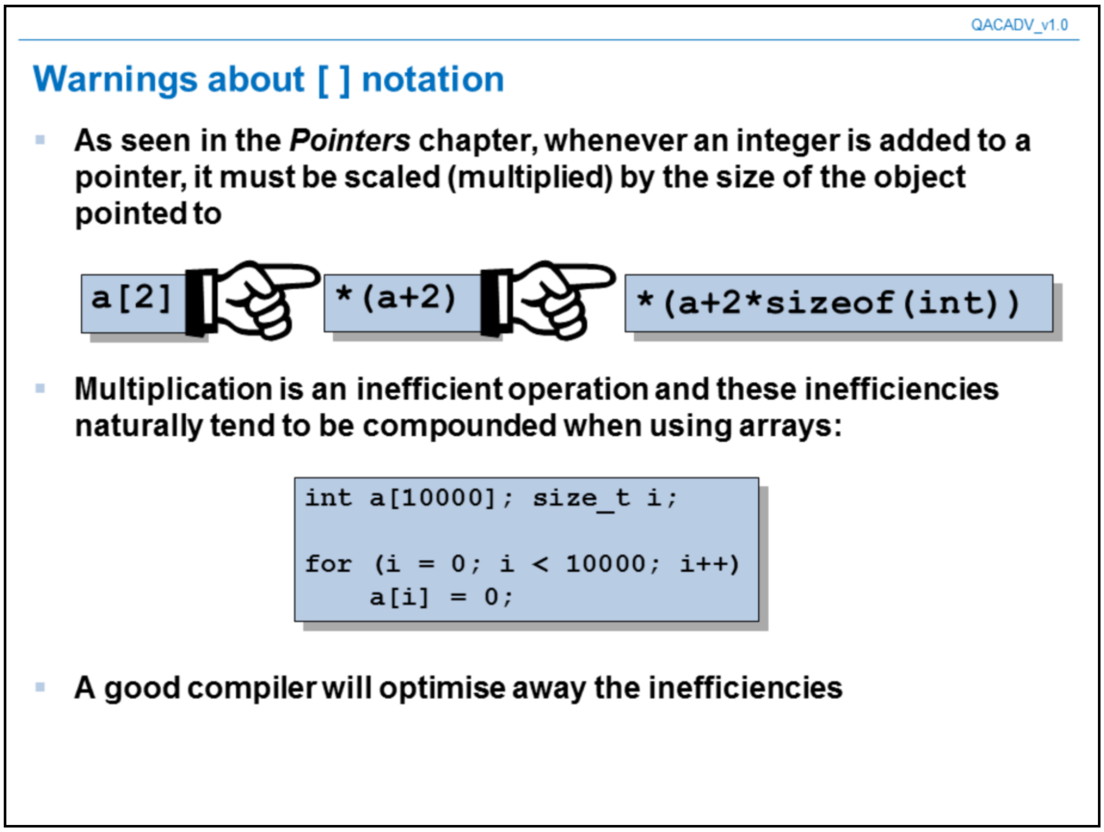

Compiler returns this address
base + offset could return address of this
0 1 2 3 4 5 6 7 8 9 a b c d e f \0

Unfortunately, experienced C programmers sometimes feel the need to write code as shown above. Invariably, a string is indexed.

This can be explained as follows. Consider the array access `a[6]`. This gets converted to `*(a+6)`, where `a` yields the address of the start of the array. Thus, we arrive at the pointer and offset notation.

In the statement `"ABCDEF"[4]` a number of things are happening. Firstly, the compiler allocates 7 bytes of static storage: 6 characters for the string and 1 for the null terminator. Secondly, the address of this storage is returned (this address is actually the address at which the character `"A"` is stored). Thirdly, the address is used as the base value and "4" (scaled by the size of a char, which is 1 byte) is used as the offset.

When 4 bytes are added to the address at which `"A"` is stored, the address of the `"E"` is found.



QACADV_v1.0

Warnings about [ ] notation

- As seen in the *Pointers* chapter, whenever an integer is added to a pointer, it must be scaled (multiplied) by the size of the object pointed to

```
a[2] ☞ *(a+2) ☞ *(a+2*sizeof(int))
```

- Multiplication is an inefficient operation and these inefficiencies naturally tend to be compounded when using arrays:

```
int a[10000]; size_t i;

for (i = 0; i < 10000; i++)
    a[i] = 0;
```

- A good compiler will optimise away the inefficiencies

Although the compiler converts array notation into pointer and offset notation for efficiency reasons, there can be an efficiency problem with the pointer and offset notation.

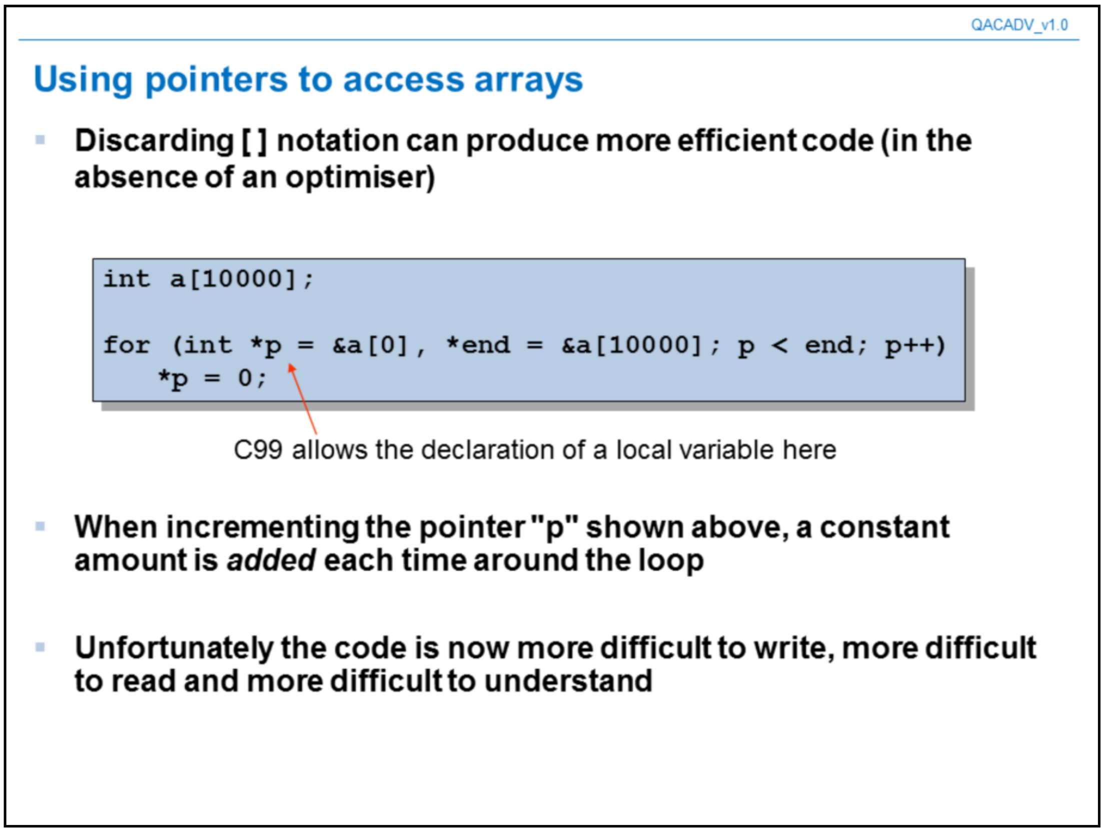
Consider `*(a+n)` which the compiler generates from `a[n]`. As previously noted, the `n` must be scaled, that is multiplied by the `sizeof` the type in the array.

For example, if `a` is an array of integers, `a[3]` is an access to the fourth element of `a`. The compiler generates the notation `*(a + 3)`. The array name `a` yields the address of the first element, say 1000. Imagine that each integer in the array is 4 bytes in size. If we add 3 directly into 1000 we will index three quarters of the way into the first integer. The 3 must be scaled by the `sizeof` an integer (4 bytes) to give 12 bytes. This 12 bytes is added to address 1000, giving a final address of 1012.

All quite straightforward, but the problem is that the 3 must be *multiplied* by the `sizeof` the type the array contains, that is 3 multiplied by 4 bytes gives 12 bytes. Multiplication is not an efficient process for the CPU to perform. Good compilers will optimise. For instance, multiplying by 4 bytes could be done by a left shift by 2 bits.

Any small, non optimised, inefficiencies tend to be naturally accentuated by the way in which we use arrays. Thus for the array of 10000 integers declared above, an inefficient multiplication operation would be executed 10000 times.

If the optimiser is not enabled by default, heavy use of arrays should cause us to reach for the manual to find out how to switch it on!



QACADV_v1.0

Using pointers to access arrays

- Discarding [ ] notation can produce more efficient code (in the absence of an optimiser)

```
int a[10000];

for (int *p = &a[0], *end = &a[10000]; p < end; p++)
    *p = 0;
```

C99 allows the declaration of a local variable here

- When incrementing the pointer "p" shown above, a constant amount is *added* each time around the loop

- Unfortunately the code is now more difficult to write, more difficult to read and more difficult to understand

If you don't have, or don't trust your optimiser, one solution to the inefficiency problems is to dispense with array notation altogether. If this is done, the compiler cannot convert it to pointer and offset notation and thus cannot introduce the inefficient multiplication operation.

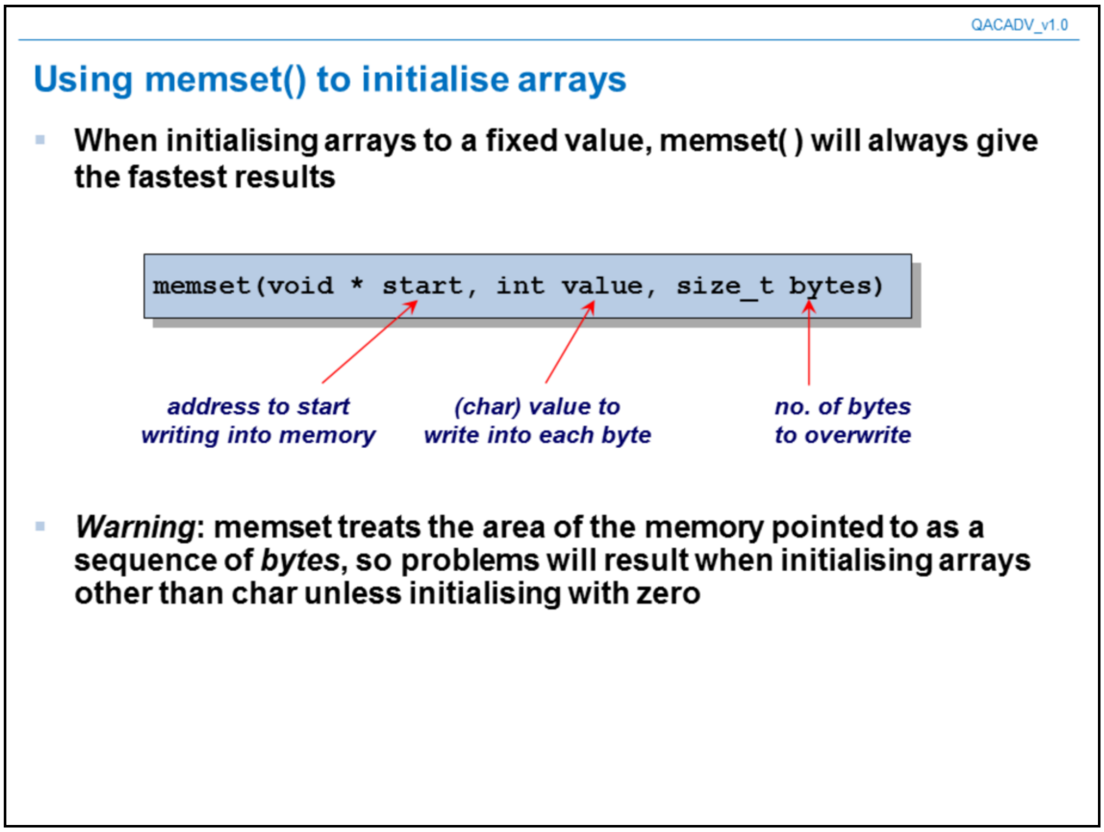
The example shown sets a pointer to the start of the array. Each time around the loop, the pointer is incremented to move to the next location in the array.

One possible argument against this code is that the pointer increment could suffer from multiplication problems. That is, when 1 is added to the pointer, the addition is scaled by the `sizeof` the datatype being pointed to, such as 4 for an integer. Therefore, each time around the loop, 1 is multiplied by 4 before adding to the pointer. The inefficiencies seem to creep back!

Fortunately, most compilers are smarter than this and realise that a constant amount (such as 4) is being added to the pointer each time around the loop. This constant amount could be added using repeated increment instructions (the increment instruction is particularly efficient).

Note the structure of the first loop. The address of the element one beyond the end of the array is stored in a variable and compared against the pointer. This can be more efficient than comparing against the address of the 10000th element directly. Depending on the compiler, the address of the 10000th element can be re-calculated each time around the loop. Doing this involves multiplying 10000 by the `sizeof` an integer. Back comes the inefficiency.

A disadvantage with the code written here is that it is difficult to read. The code on the previous page is easier to write, easier to read, easier to understand and easier to debug. We have sacrificed clarity for a dubious increase in efficiency.


QACADV_v1.0

Using memset() to initialise arrays

- **When initialising arrays to a fixed value, memset( ) will always give the fastest results**

```
memset(void * start, int value, size_t bytes)
```

address to start writing into memory
(char) value to write into each byte
no. of bytes to overwrite

- ***Warning*: memset treats the area of the memory pointed to as a sequence of *bytes*, so problems will result when initialising arrays other than char unless initialising with zero**

No matter what type of pointer is used, the fastest way to initialise an array is to set the values at definition time, e.g.

```
int  a[10]= {0};
```

The short-cut of only specifying the first item only works for zero, otherwise every element must be specified.

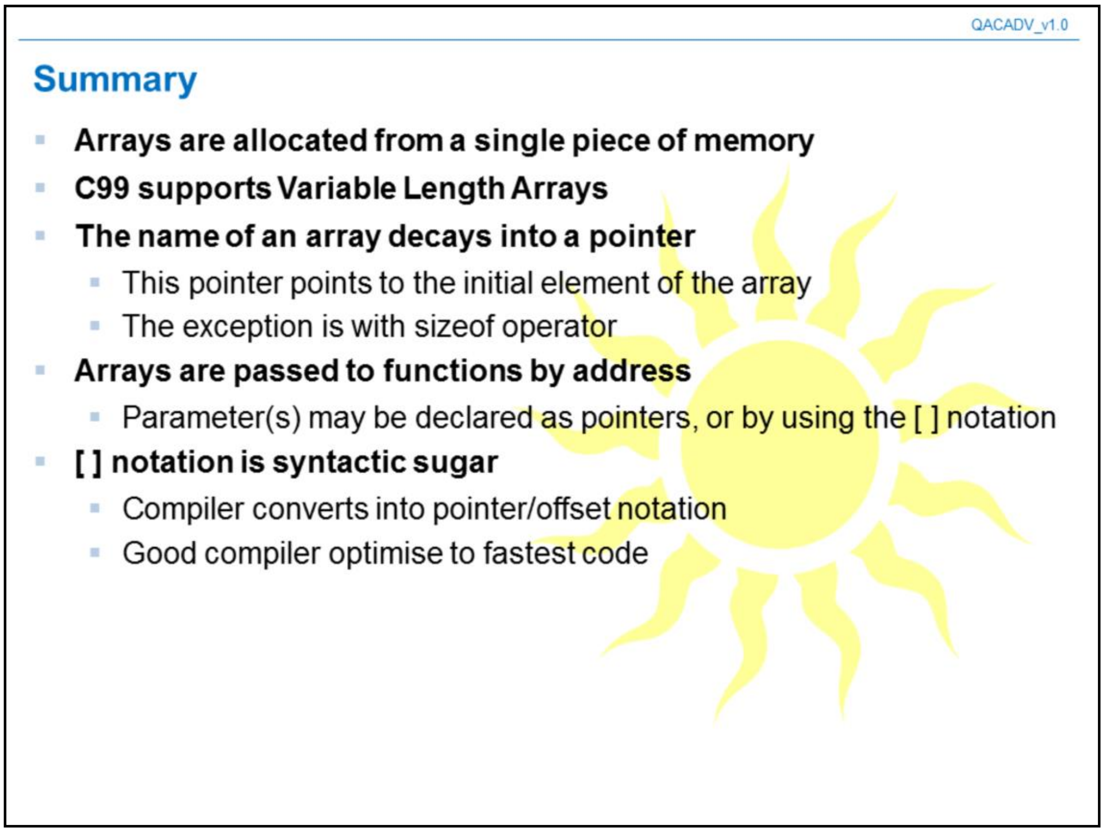
It may be impractical to hard-code each element, or we may want to re-initialise an array. In this case the fastest method is to use the `memset()` function. This usually uses machine-code instructions which can alter large or small amounts of memory within a few instruction cycles.

The main problem with `memset()` is that it treats the region of memory as a sequence of bytes. Consider the following code:

```
int  a[10];
memset(a, 1, sizeof(a));
```

This will *not* fill the elements of the array with 1s. On a typical 32-bit machine each element will contain 0x01010101, i.e. 16,843,009. Using any number > 128 (0x80) would set the sign bit on most platforms, and the effect of a number > 255 (0xff) is not defined.

Thus, unless initialising arrays of characters, or initialising an array of any type to zero, unexpected results may be obtained if `memset()` is used.



QACADV_v1.0

Summary

- **Arrays are allocated from a single piece of memory**
- **C99 supports Variable Length Arrays**
- **The name of an array decays into a pointer**
  - This pointer points to the initial element of the array
  - The exception is with sizeof operator
- **Arrays are passed to functions by address**
  - Parameter(s) may be declared as pointers, or by using the [ ] notation
- **[ ] notation is syntactic sugar**
  - Compiler converts into pointer/offset notation
  - Good compiler optimise to fastest code

In this chapter we have looked closely at the way arrays are handled by modern-day C compilers, particularly the close and often confusing relationship between arrays and pointers.

We have seen that C translates array notation into pointer and offset expressions, which in turn require a multiplication to be performed to achieve scaled pointer arithmetic. This can be particularly inefficient when processing large arrays, where any slight loss in speed is accentuated due to the repetition within a `for` loop.

The chapter also discussed a number of interesting techniques using the `sizeof` operator and macros, although there are some situations where the idea breaks down. We have also discussed in some detail the way in which C passes arrays into functions and the reasons for doing it this way.

In the next chapter we shall continue our investigation of arrays by looking at arrays of arrays, or "multi-dimensional arrays".