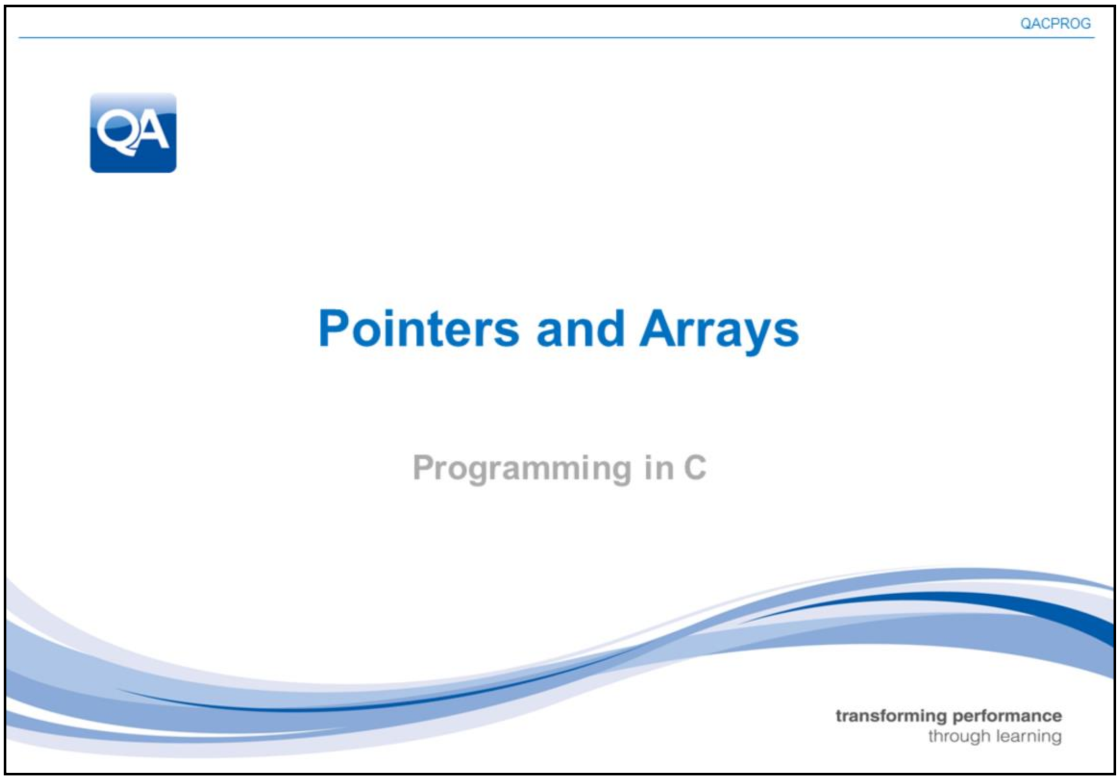
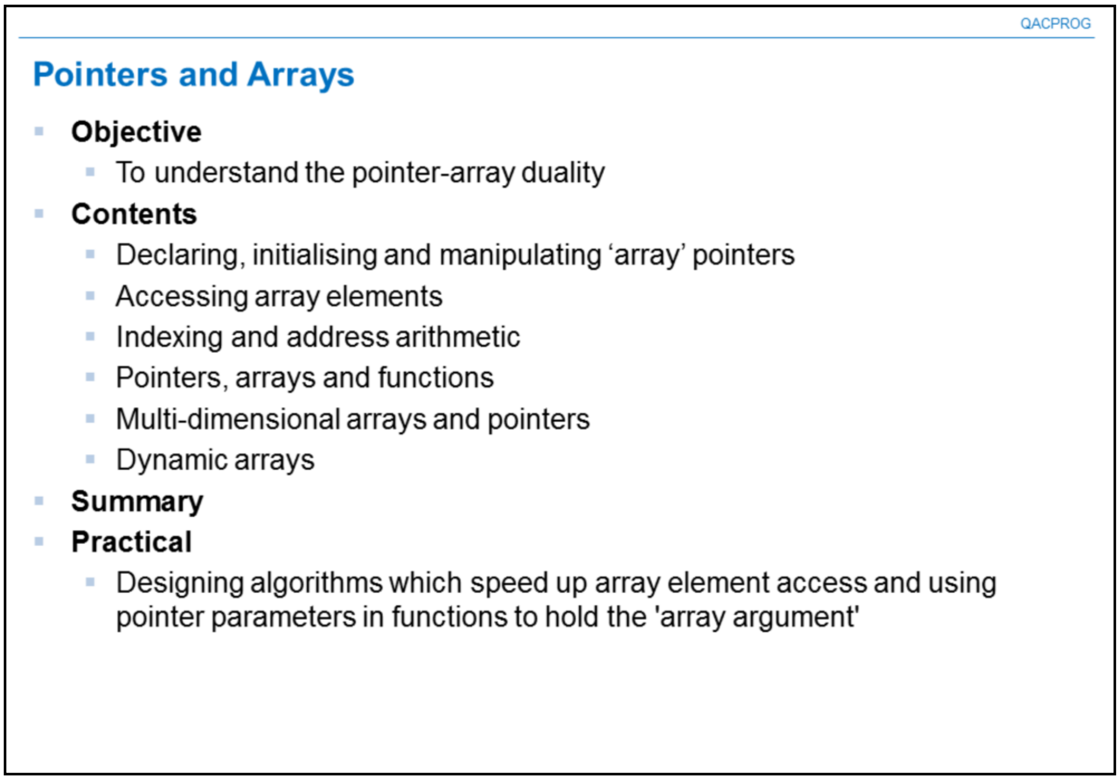


QACPROG

Pointers and Arrays

Programming in C

transforming performance
through learning



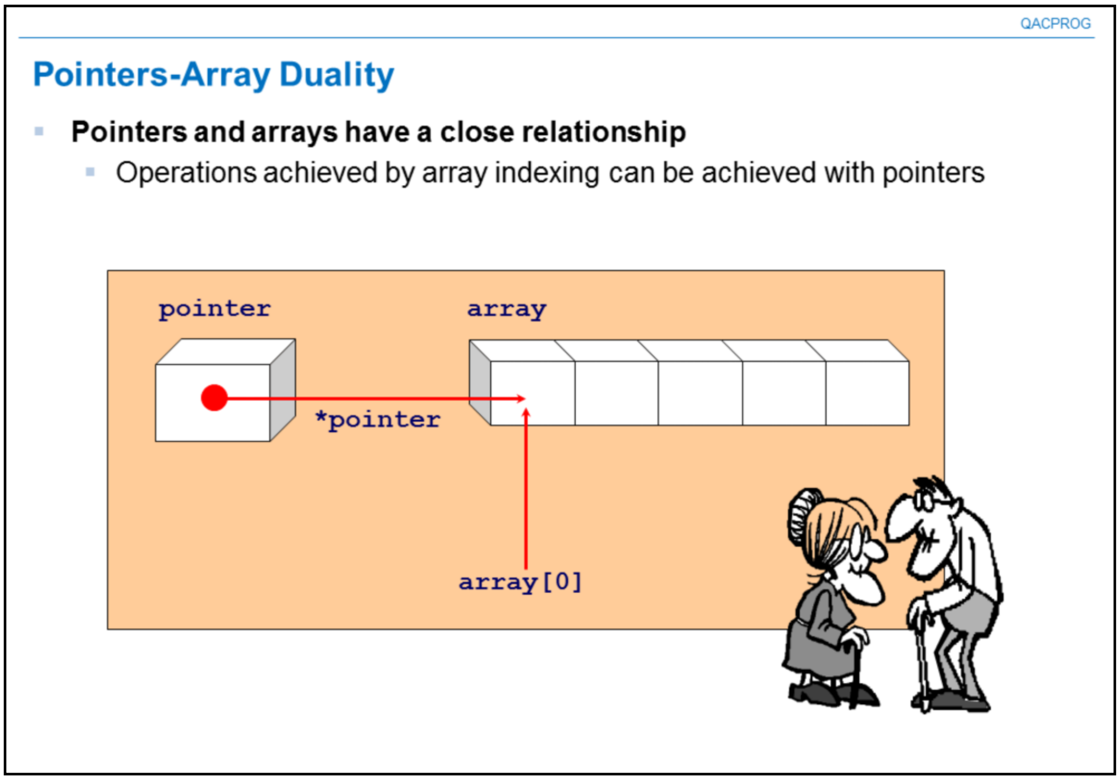
QACPROG

Pointers and Arrays

- **Objective**
  - To understand the pointer-array duality
- **Contents**
  - Declaring, initialising and manipulating 'array' pointers
  - Accessing array elements
  - Indexing and address arithmetic
  - Pointers, arrays and functions
  - Multi-dimensional arrays and pointers
  - Dynamic arrays
- **Summary**
- **Practical**
  - Designing algorithms which speed up array element access and using pointer parameters in functions to hold the 'array argument'

The objective of this chapter is to consolidate the concept of pointers, to establish their importance generally and to demonstrate their power when used with arrays, including multi-dimensional arrays.

The chapter concludes with an example of a simple 'dynamic' array.



QACPROG

Pointers-Array Duality

- **Pointers and arrays have a close relationship**
  - Operations achieved by array indexing can be achieved with pointers

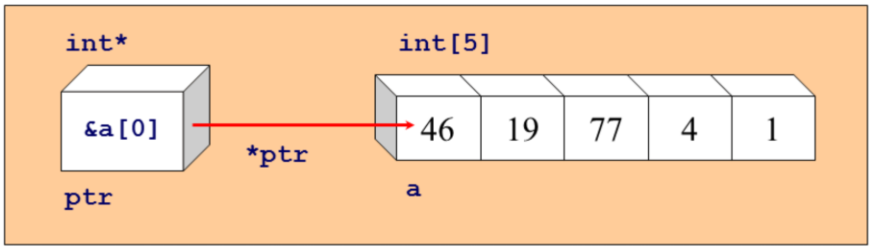
pointer
array
*pointer
array[0]

There are two major reasons why arrays are closely linked with pointers. Firstly, the array name is a constant pointer that contains the address of the initial element. Secondly, array elements are placed in contiguous memory.

The first observation will provide an insight into the way arrays are ‘passed’ into functions, which is a topic covered later in the chapter. The latter observation will explain why array access is so much faster using pointers; if you have the location of one element’s address, you can get its neighbour very easily by incrementing or decrementing the address.

QACPROG

Declaring an Array Pointer

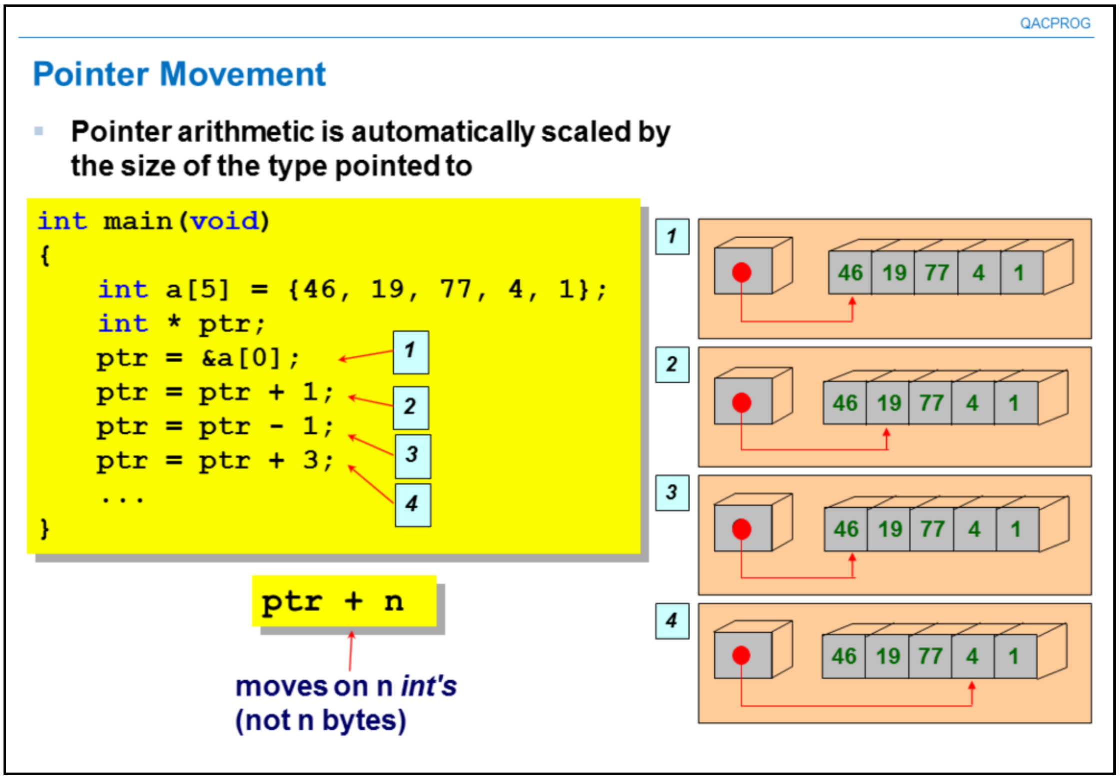


```
int main(void)
{
     int a[5] = { 46, 19, 77, 4, 1 };

     int * ptr;

     ptr = &a[0];
     ...
}
```

1. declare a plain pointer

2. point to the array!

An array still has to be declared as such; the sizing information is compulsory. If pointers are to be used to access the elements, they must be declared as pointers to the type of the element. The working pointer `ptr` goes through the same process as ordinary pointers. Declare it, initialise it, then dereference it. The first two stages are shown in the second and third statements.



QACPROG

Pointer Movement

- Pointer arithmetic is automatically scaled by the size of the type pointed to

```
int main(void)
{
    int a[5] = {46, 19, 77, 4, 1};
    int * ptr;
    ptr = &a[0];
    ptr = ptr + 1;
    ptr = ptr - 1;
    ptr = ptr + 3;
    ...
}
```

```
ptr + n
```

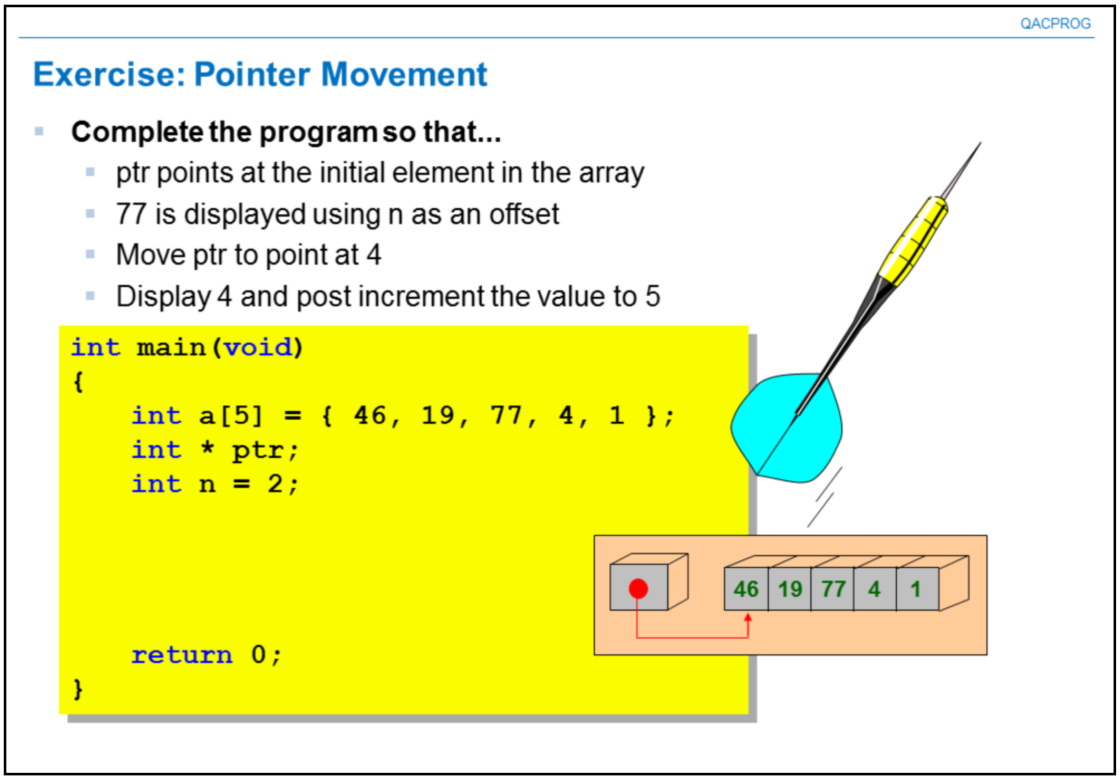
moves on n *int's* (not n bytes)

Once a pointer has a value, it can be manipulated like any other variable. In the previous chapters on pointers, the value, once initialised/assigned, is simply dereferenced using the `*` operator. If that value is pointing at an element in an array, it is possible to perform arithmetic on the pointer value itself. This means that the address is changed by adding or subtracting integer values. This is legal as long as the resulting addresses are legal, i.e. they are still within the bounds of the array.

In the example shown above, the pointer `ptr` is updated by adding 1, subtracting 1 and adding `3` to the current value. These are all legal and have the desired effect; the pointer is 'moved' forward and backward along the array.

The movement through memory is precisely the same as that achieved by indexing. The size of the element type is used as a unit, i.e. a number of memory locations for a single element. This unit is then multiplied by the integer used to add and subtract.

We still need to access the data, i.e. the integer elements themselves.



QACPROG

Exercise: Pointer Movement

- **Complete the program so that...**
  - ptr points at the initial element in the array
  - 77 is displayed using n as an offset
  - Move ptr to point at 4
  - Display 4 and post increment the value to 5

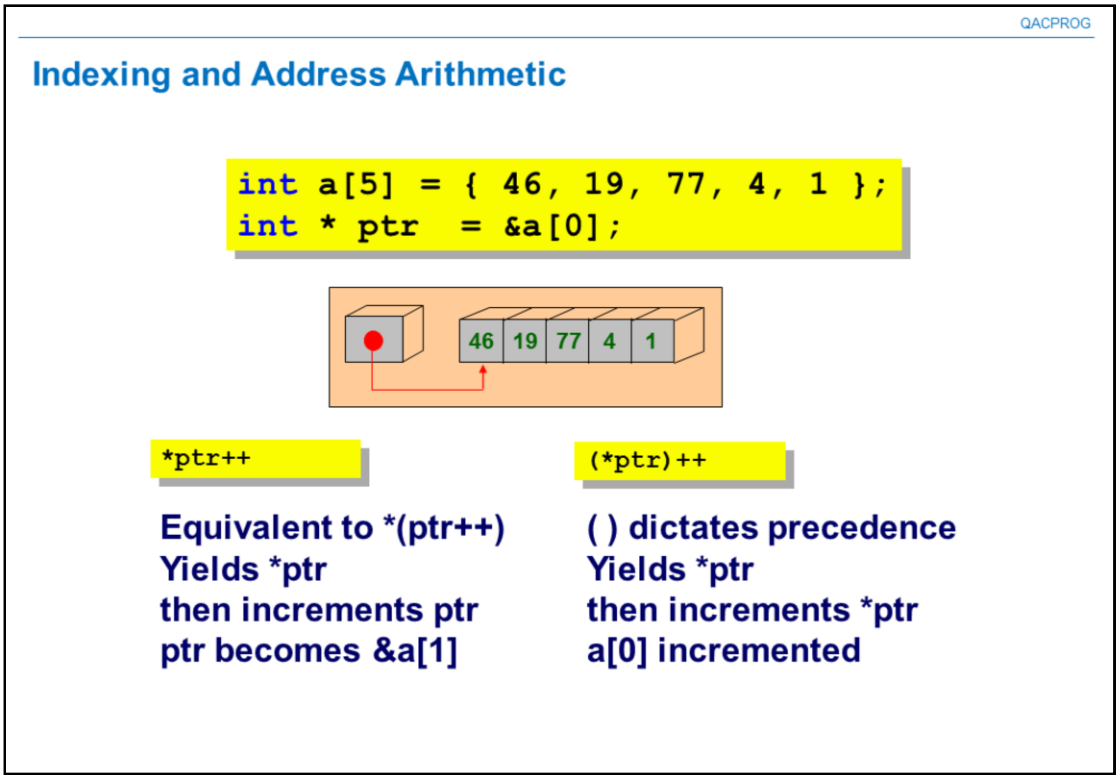
```
int main(void)
{
    int a[5] = { 46, 19, 77, 4, 1 };
    int * ptr;
    int n = 2;



    return 0;
}
```

Access to the data is achieved by using the traditional `*` indirection. Once accessed, the element behaves as if it has been indexed, i.e. it can be read from or written to.

The examples supplied so far will not generate code that is any more efficient than that produced using indexing with the `[]` operator. It does, however, reaffirm the syntax of pointers and the use of the two operators, `&` and `*` .

Solution:

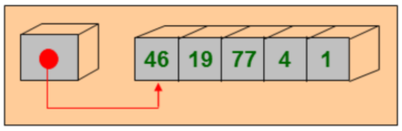
```
#include <stdio.h>

int main(void)
{
    int a[5] = { 46, 19, 77, 4, 1 };
    int * ptr;
    int n = 2;

    ptr = &a[0];
    printf("%d\n", *(ptr + n));
    ptr += 3;
    printf("%d\n", (*ptr)++);

    return 0;
}
```



QACPROG

Indexing and Address Arithmetic

```
int a[5] = { 46, 19, 77, 4, 1 };
int * ptr  = &a[0];
```



```
*ptr++
```

Equivalent to *(ptr++)
Yields *ptr
then increments ptr
ptr becomes &a[1]

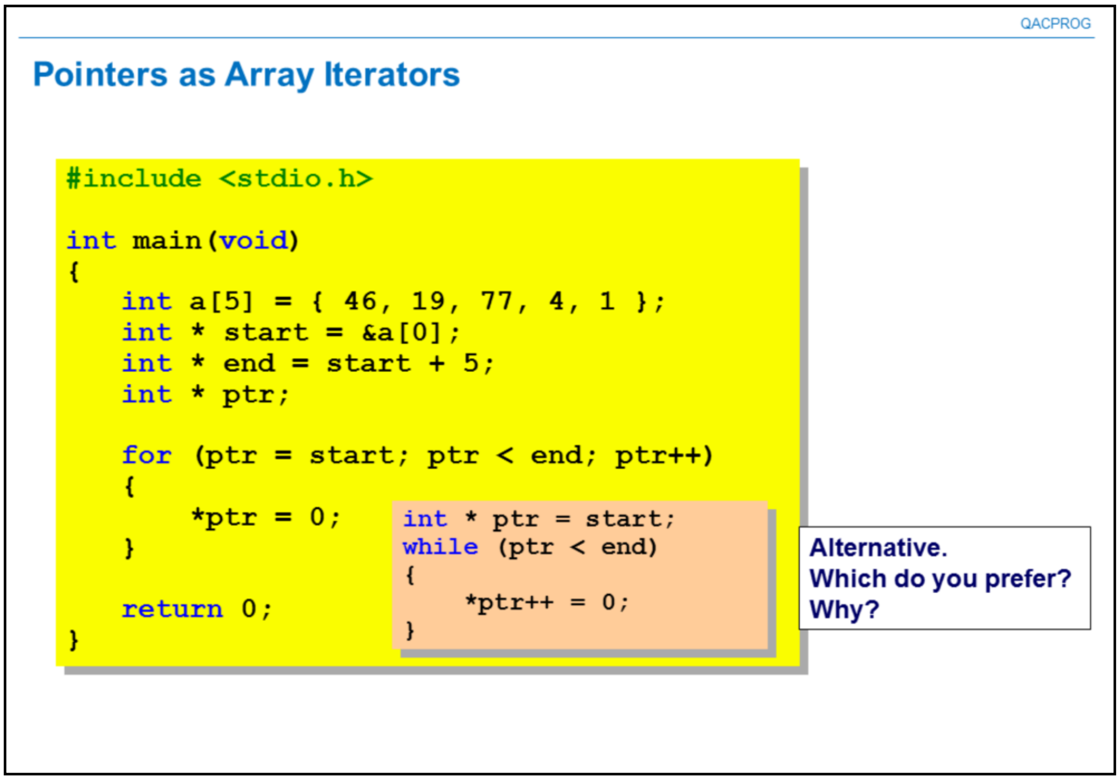
```
(*ptr)++
```

( ) dictates precedence
Yields *ptr
then increments *ptr
a[0] incremented

Leading on from the previous example, a pointer can be incremented and decremented using the `++` and `--` operators, e.g. `p++` is equivalent to `p = p + 1`.

This is where pointers with arrays are most effective. The pointer is being constrained to move forwards (or backwards) one step at a time. Once the pointer has been set in place, these operators will do the movement; no algorithm using base addresses and offsets is required.

The precedence rules need to be studied. `*` and the `++/--` operators have the same precedence. This means that the compiler will read them from right to left to determine the bonding. Brackets are used both for clarity and for forcing the bonding. Some C coding standards will suggest the use of bracketing whenever there is a potential reading problem. Others recommend avoid the issue by separating the two operators out into two statements. This avoids having to learn the precedence table and can also make the refactored sequential code easier to understand.



QACPROG

Pointers as Array Iterators

```
#include <stdio.h>

int main(void)
{
    int a[5] = { 46, 19, 77, 4, 1 };
    int * start = &a[0];
    int * end = start + 5;
    int * ptr;

    for (ptr = start; ptr < end; ptr++)
    {
        *ptr = 0;
    }

    return 0;
}
```

```
int * ptr = start;
while (ptr < end)
{
    *ptr++ = 0;
}
```

Alternative.
Which do you prefer?
Why?

This is the first of many examples that show the strength of pointers. The single statement in the `while` statement body is performing three things: /1/ `ptr` is post incremented; `ptr` will 'move' as a final thought at the end of the statement. /2/ The integer pointed at by `ptr` is accessed by `*`. /3/ Zero is assigned to the integer pointed to by `ptr`. /4/ `ptr` will now get updated.

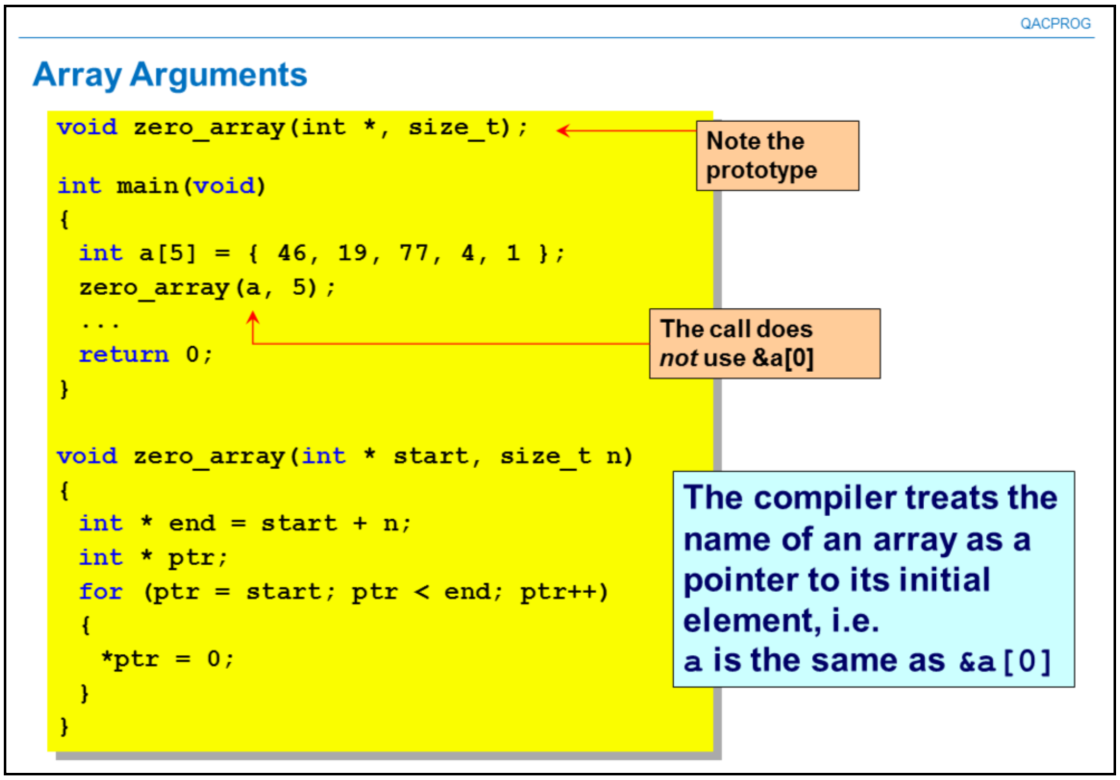
This is pretty tricky! Many programmers prefer the `for` statement - the 'scaffolding' performs the iteration and the 'body' performs the access - a clear separation of responsibilities.

The `end` pointer points just *past* the end of the array. This is allowed. However, any attempt to dereference such a pointer is undefined behaviour. In other words, the sole use of a one-beyond-the-end address is to compare against it.

The iteration can of course be written as:

```
for (i = 0; i < 5; i++)
    a[i] = 0;
```

Note: the comparison operators == and != can be used on *any* pointers, but the other four relational operators are only defined when both pointer arguments point *within* the same array.



Array Arguments

```
void zero_array(int *, size_t);

int main(void)
{
  int a[5] = { 46, 19, 77, 4, 1 };
  zero_array(a, 5);
  ...
  return 0;
}

void zero_array(int * start, size_t n)
{
  int * end = start + n;
  int * ptr;
  for (ptr = start; ptr < end; ptr++)
  {
    *ptr = 0;
  }
}
```

The compiler treats the name of an array as a pointer to its initial element, i.e.
a is the same as `&a[0]`

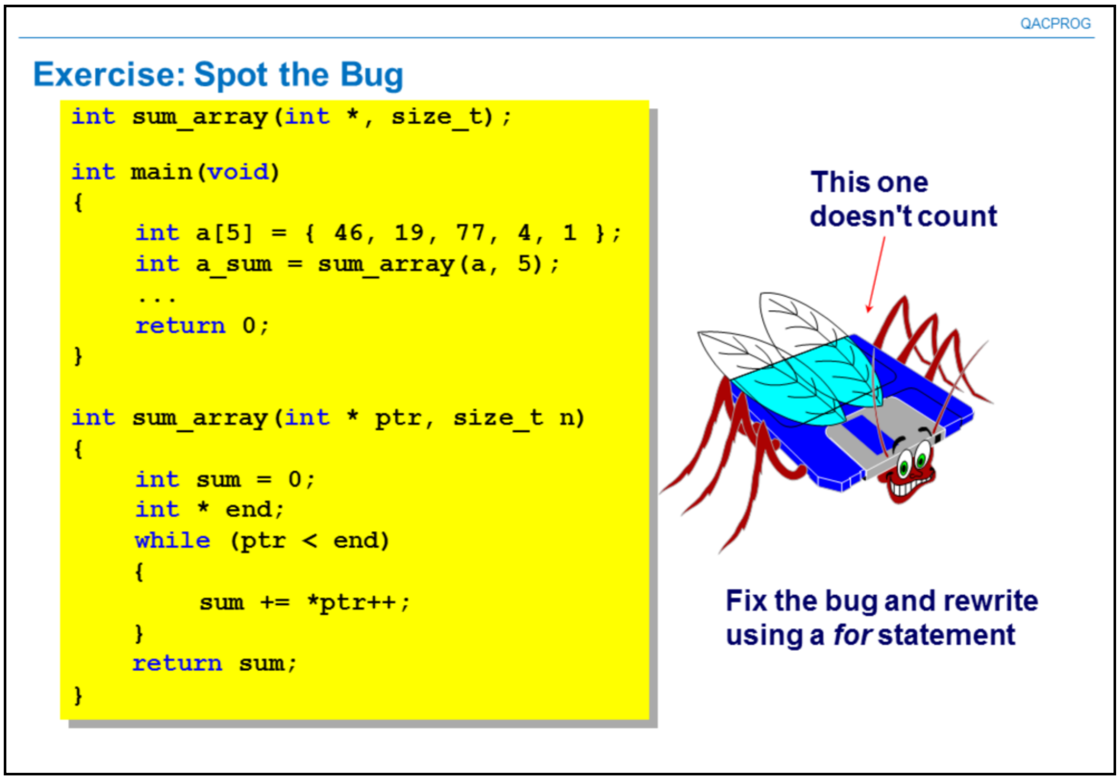
When used in an expression, the name of an array automatically 'decays' into the address of the initial element of the array. This again highlights the close nature of arrays and pointers. It also reveals the mechanism by which C automatically achieves pass by reference semantics for array arguments.

The name of an array is actually a const pointer. Hence we cannot write a++. This makes sense. The name of an array is the address where the compiler locates the array. So what does a++ mean? It does not mean increment every element inside a (which would be disallowed anyway as it is a whole array operation). Remember, a++ is an expression using the name of an array. The name of an array *automatically* decays into the address of the initial element of the array. Hence a++ is conceptually an attempt to change the address of the array, to relocate the array. This is not allowed (and can also be seen as a whole array operation).

This gives use two way to use the name of an array to access an array element: We can use simple array subscripting, or we can use use pointer arithmetic combined with a dereference:

```
a[n];    *(a + n);
```

In fact, the subscript operator is pure syntactic sugar. Under the hood the compiler will translate a[n] into *(a + n).



QACPROG

Exercise: Spot the Bug

```
int sum_array(int *, size_t);

int main(void)
{
    int a[5] = { 46, 19, 77, 4, 1 };
    int a_sum = sum_array(a, 5);
    ...
    return 0;
}

int sum_array(int * ptr, size_t n)
{
    int sum = 0;
    int * end;
    while (ptr < end)
    {
        sum += *ptr++;
    }
    return sum;
}
```

This one doesn't count

Fix the bug and rewrite using a *for* statement

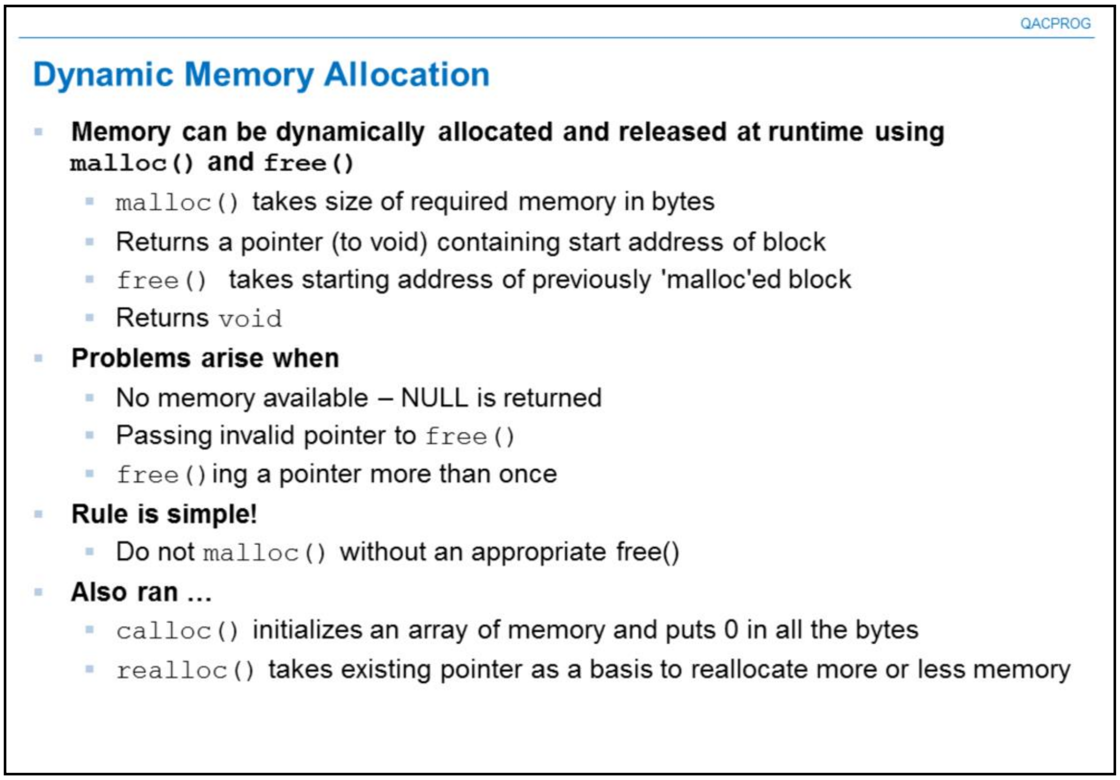
The code fragment compiles but you are warned not to try it. The `end` pointer is not initialised or assigned. Its value is undefined. It could point *anywhere*! All bets are off.

Here are two solutions reworked using `for` statements:

```
int sum_array(int * start, size_t n)
{
    int sum = 0;
    int * end = start + n;
    int * ptr;
    for (ptr = start; ptr < end; ptr++)
        sum += *ptr;
    return sum;
}

int sum_array(int * array, size_t n)
{
    int sum = 0;
    int i;
    for (i = 0; i < n; i++)
        sum += array[i];
    return sum;
}
```

If we wish we can change the function prototype to the semantically identical:

```
int sum_array(int array[], size_t n) ...
```


QACPROG

Dynamic Memory Allocation

- **Memory can be dynamically allocated and released at runtime using `malloc()` and `free()`**
  - `malloc()` takes size of required memory in bytes
  - Returns a pointer (to void) containing start address of block
  - `free()` takes starting address of previously 'malloc'ed block
  - Returns `void`
- **Problems arise when**
  - No memory available – NULL is returned
  - Passing invalid pointer to `free()`
  - `free()`ing a pointer more than once
- **Rule is simple!**
  - Do not `malloc()` without an appropriate free()
- **Also ran …**
  - `calloc()` initializes an array of memory and puts 0 in all the bytes
  - `realloc()` takes existing pointer as a basis to reallocate more or less memory

The system header file `stdlib.h` contains these prototypes:

```
    void * malloc(size_t bytes);
    void free(void * ptr);
```

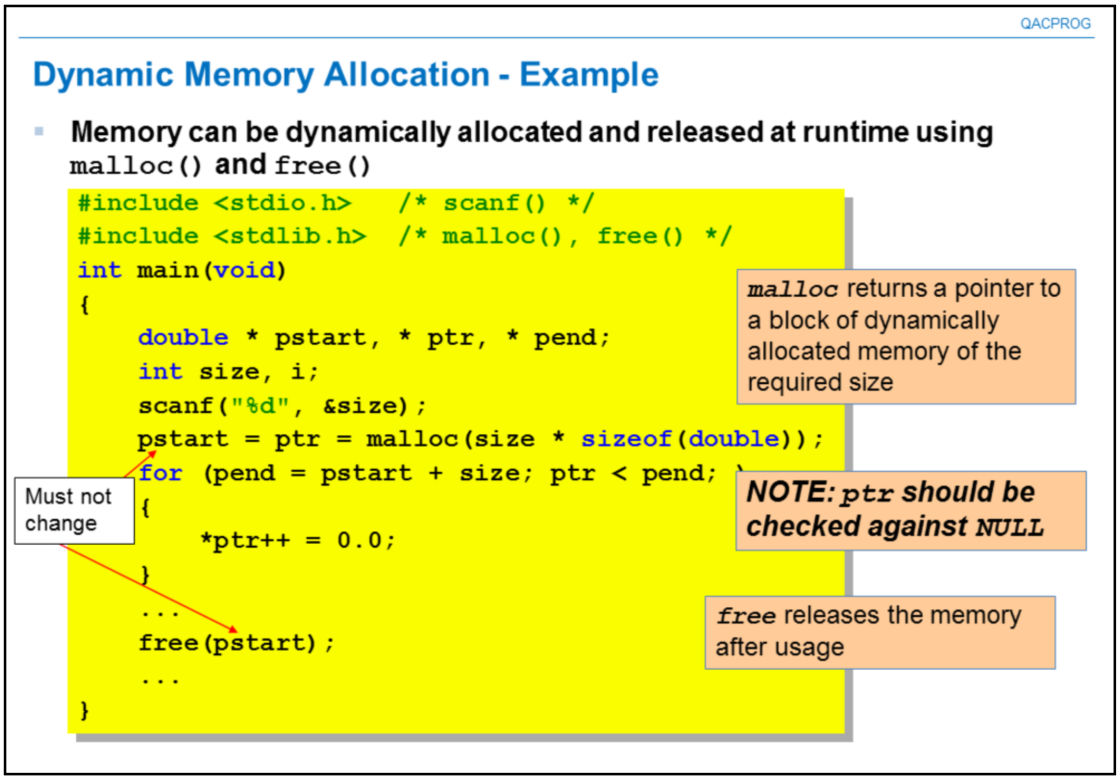
`malloc()` takes an unsigned integer argument and passes back a pointer that contains the address of a contiguous block of memory. The amount of memory required (in bytes) is specified as the single argument (also used as the second argument in the other allocation functions below). If there is not enough memory, the zero or `NULL` pointer is returned (this is ignored in the slide). The pointer type returned is a 'pointer to `void`' (the closest thing to a generic pointer).

`free()` has to take a pointer argument that has 'cleanly' been returned from a previous `malloc()`. This means that the pointer has not moved further into the starting address of the block. Also, we must not call `free()` more than once on the same pointer.

For prevention of memory leakage we MUST free what we allocate – exactly, precisely and timely!

The `calloc()` function takes an extra first argument which is the size of the array to be allocated. It also initializes the memory with (byte) `0`'s. Like `malloc()`, it will return `NULL` if unsuccessful

The `realloc()` function takes an extra first argument which is a previously allocated address. The returned pointer (if successful) returns a pointer to allocated memory with all the data in the first argument preserved. It can cope with both shrinkage and expansion. Like both `malloc()` and `calloc()`, it will return `NULL` if unsuccessful



QACPROG

Dynamic Memory Allocation - Example

- Memory can be dynamically allocated and released at runtime using `malloc()` and `free()`

```
#include <stdio.h>   /* scanf() */
#include <stdlib.h>  /* malloc(), free() */
int main(void)
{
    double * pstart, * ptr, * pend;
    int size, i;
    scanf("%d", &size);
    pstart = ptr = malloc(size * sizeof(double));
    for (pend = pstart + size; ptr < pend;
    {
        *ptr++ = 0.0;
    }
    ...
    free(pstart);
    ...
}
```

Must not change

malloc returns a pointer to a block of dynamically allocated memory of the required size

NOTE: ***ptr should be checked against NULL***

free releases the memory after usage

It is essential to read all about `malloc()` and `free()`.
In our example, the pointer is implicitly cast to an `double *`. An explicit cast is *required* only to maintain C++ compatibility (increasingly an issue, so worth doing). The `double *` is then used to access the memory as an integer array. Note the `[]` operator used with the pointer:

```
    ptr[i] = 0.0;
```

Remember this is just syntactic sugar for:

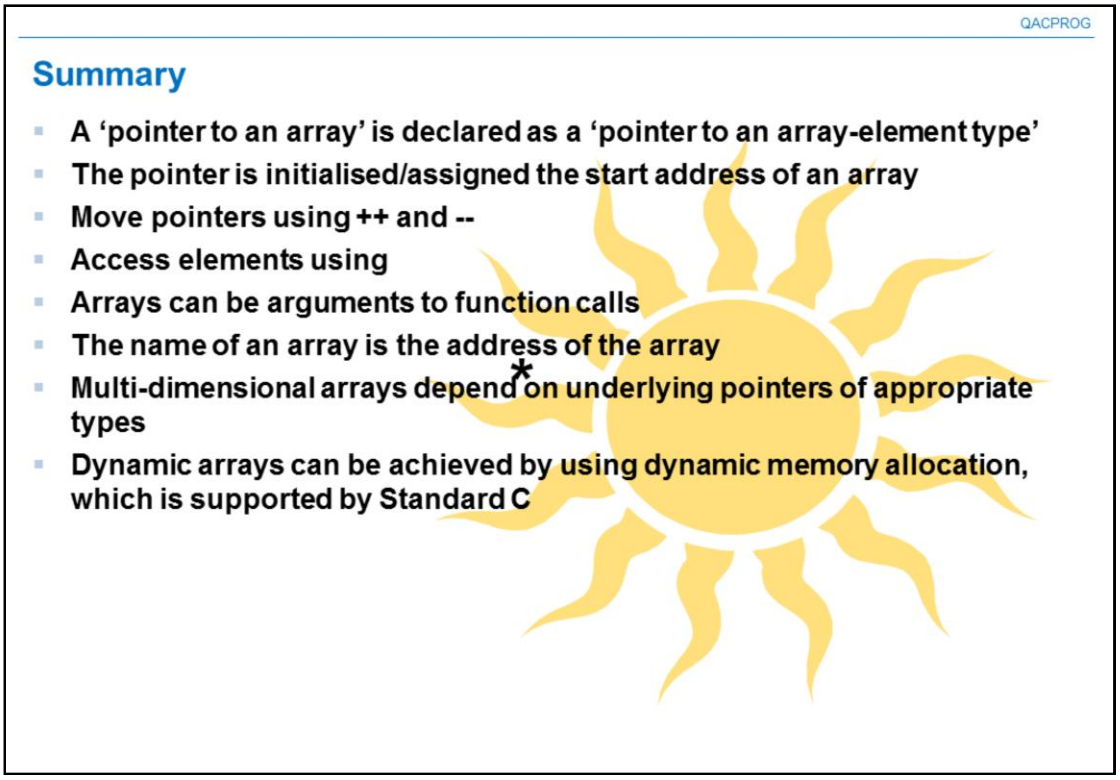
```
    *(ptr + i) = 0.0;
```

In order to preserve the allocated pointer, it is sometimes stored prior to access, i.e.

```
     double * ptr, * start;
     …
     ptr = malloc(size * sizeof(double));
     start = ptr;
     if (ptr != NULL)    /* Essential!!!! */
     {
          while (ptr < start + size)
          {
               *(ptr++) = 0.0;
          }
          . . .
     free(start);
     }
```

Note that we could have used `calloc` instead of `malloc`:

```
     ptr = calloc(size, sizeof(double));
```



QACPROG

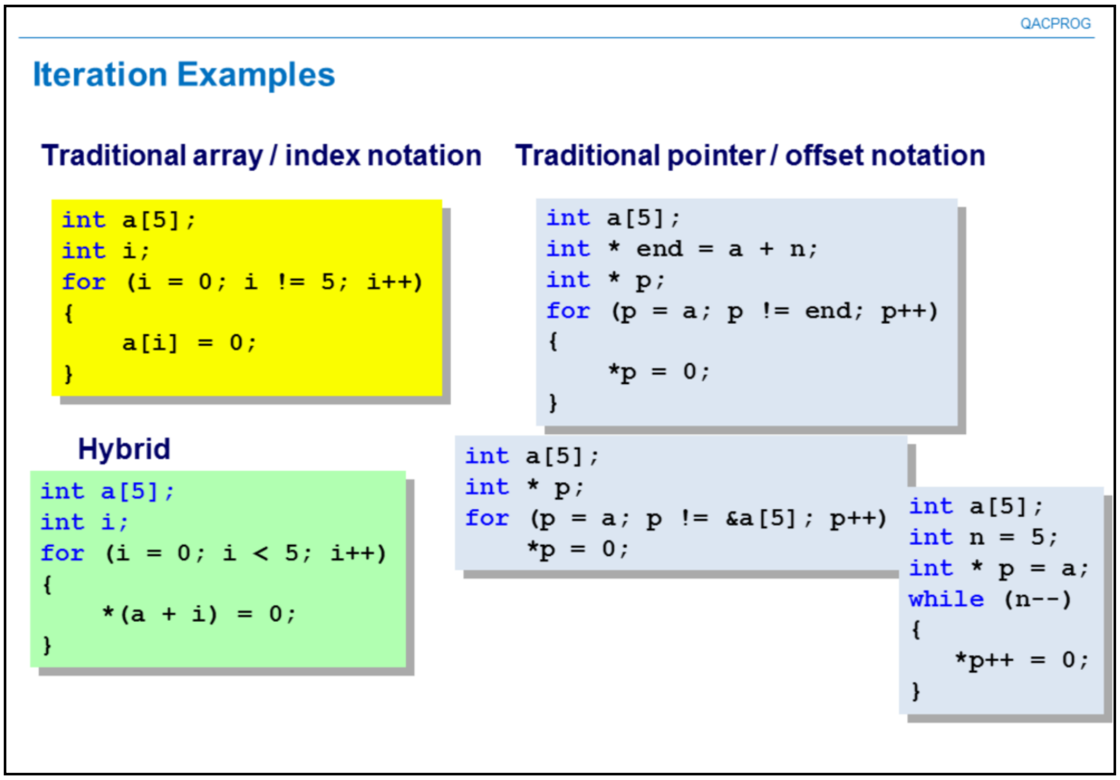
Summary

- A ‘pointer to an array’ is declared as a ‘pointer to an array-element type’
- The pointer is initialised/assigned the start address of an array
- Move pointers using ++ and --
- Access elements using *
- Arrays can be arguments to function calls
- The name of an array is the address of the array
- Multi-dimensional arrays depend on underlying pointers of appropriate types
- Dynamic arrays can be achieved by using dynamic memory allocation, which is supported by Standard C

Pointers can be used to perform any array manipulation that can be carried out using indices. Their use can result in more-efficient code. This is guaranteed if access is made sequentially through an array. Pointers are usually declared as pointers to an element type. This includes the case when the elements are structures and unions. If the element is an array type, i.e. the enveloping array is multidimensional, the use of pointers to pointers is required.

Multidimensional arrays are held in contiguous memory with for example, (in the 2 D case), the first row, followed by the 2nd row, etc … Efficiency is achieved through the use of pointers to blocks of elements, i.e. pointer to an array of 13 ints.

Dynamic arrays are implemented using functions that perform dynamic memory allocation. The size is then a runtime expression that is passed into one of these functions. From then on, conventional array access is available. Care must still be taken to check the bounds.



QACPROG

Iteration Examples

Traditional array / index notation

```
int a[5];
int i;
for (i = 0; i != 5; i++)
{
    a[i] = 0;
}
```

Traditional pointer / offset notation

```
int a[5];
int * end = a + n;
int * p;
for (p = a; p != end; p++)
{
    *p = 0;
}
```

```
int a[5];
int * p;
for (p = a; p != &a[5]; p++)
    *p = 0;
```

```
int a[5];
int n = 5;
int * p = a;
while (n--)
{
    *p++ = 0;
}
```

Hybrid

```
int a[5];
int i;
for (i = 0; i < 5; i++)
{
    *(a + i) = 0;
}
```