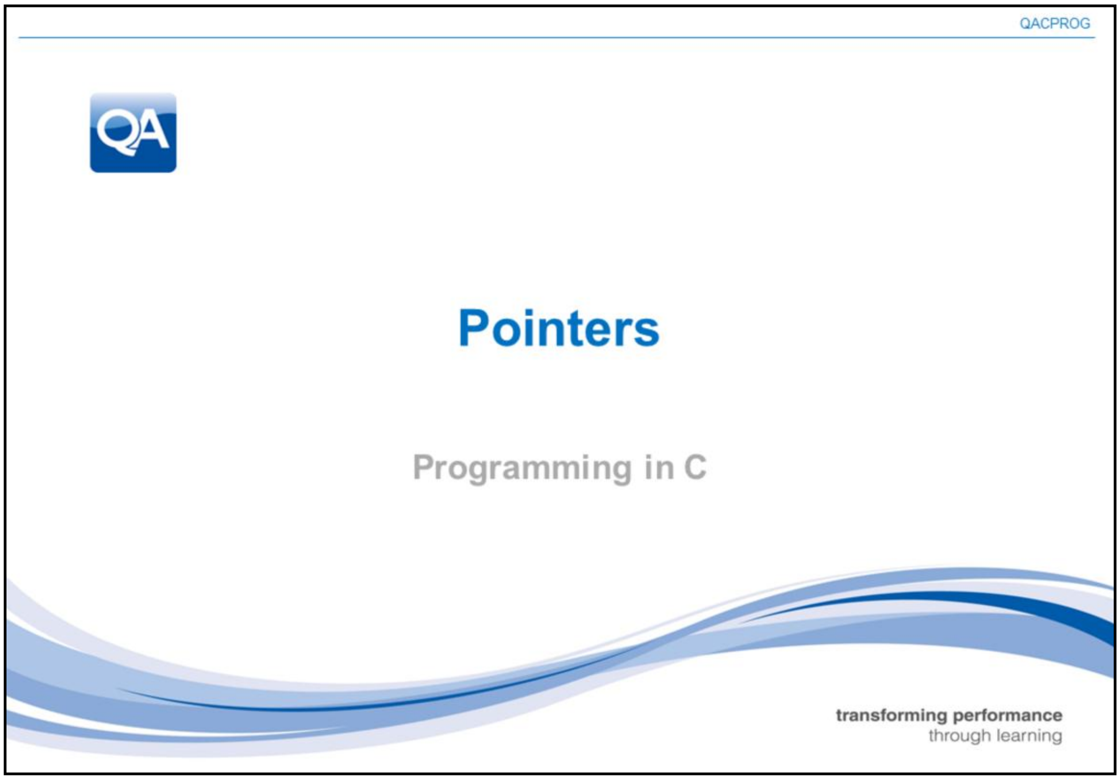
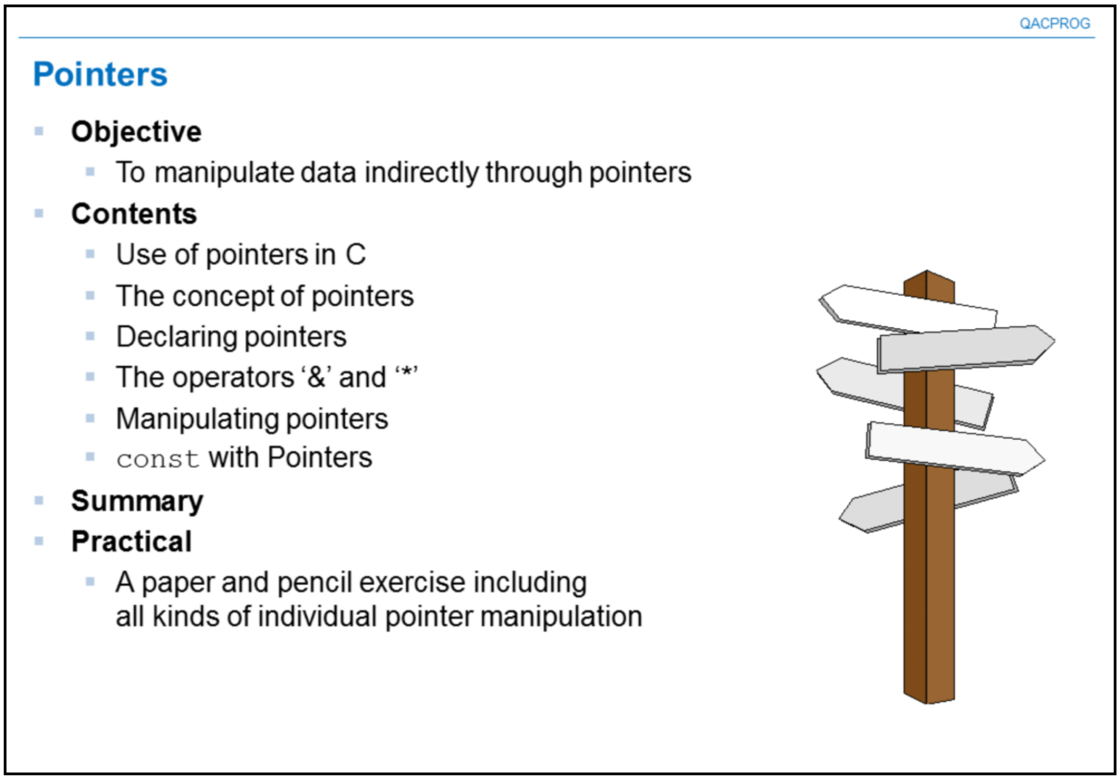


QACPROG

Pointers

Programming in C

transforming performance
through learning

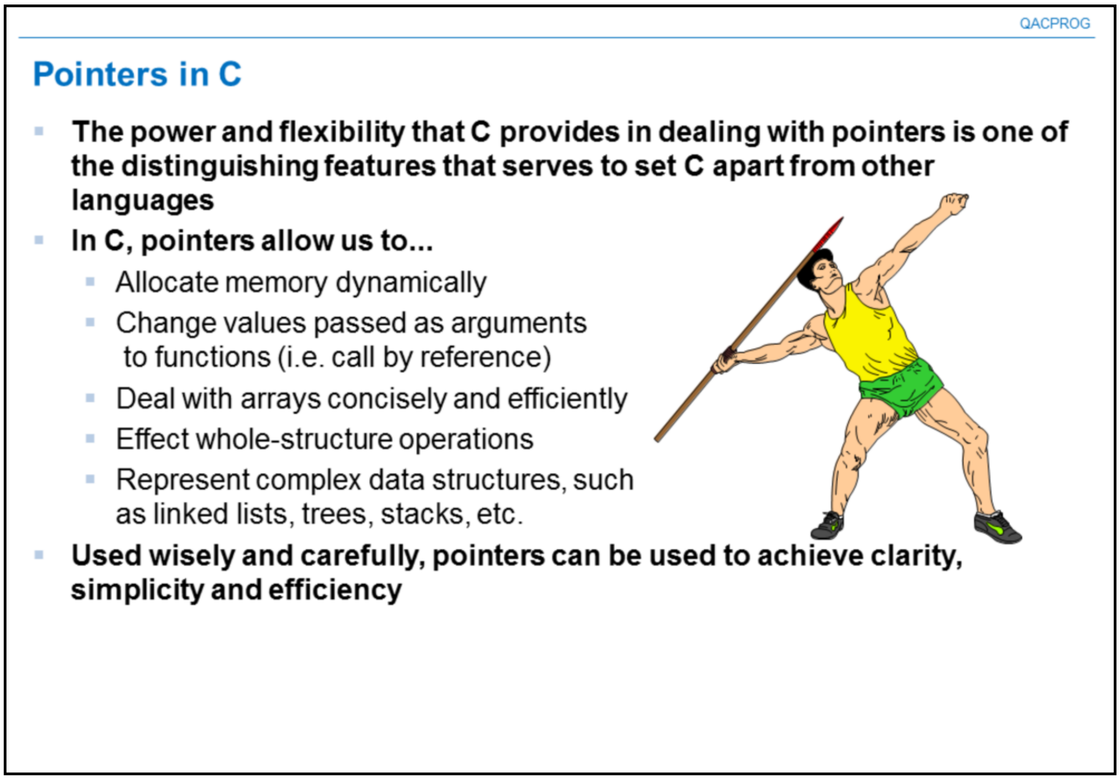


QACPROG

Pointers

- **Objective**
  - To manipulate data indirectly through pointers
- **Contents**
  - Use of pointers in C
  - The concept of pointers
  - Declaring pointers
  - The operators '&' and '*'
  - Manipulating pointers
  - `const` with Pointers
- **Summary**
- **Practical**
  - A paper and pencil exercise including all kinds of individual pointer manipulation

This chapter is the first of many dealing with pointers. The objective of this chapter is to introduce the concept of pointers, the syntax used to manipulate them and some of their tricks.



QACPROG

Pointers in C

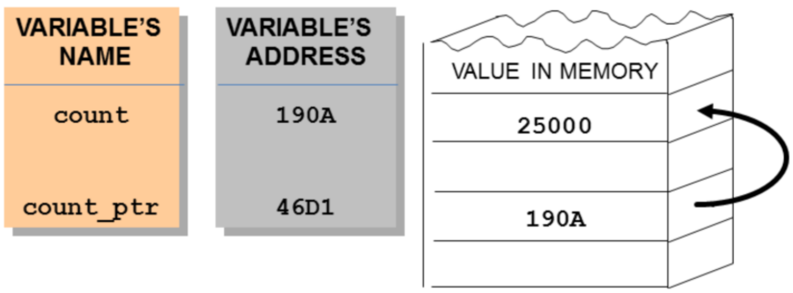
- **The power and flexibility that C provides in dealing with pointers is one of the distinguishing features that serves to set C apart from other languages**
- **In C, pointers allow us to...**
  - Allocate memory dynamically
  - Change values passed as arguments to functions (i.e. call by reference)
  - Deal with arrays concisely and efficiently
  - Effect whole-structure operations
  - Represent complex data structures, such as linked lists, trees, stacks, etc.
- **Used wisely and carefully, pointers can be used to achieve clarity, simplicity and efficiency**

It is essential to get to grips with the concept of pointers. It is the only way that some techniques can be achieved. They also provide more-efficient alternative techniques without decreasing readability.

QACPROG

Concept of Pointers - Indirection

- A pointer is a variable that contains the address of another variable
- Since a pointer contains the address of a variable, it is possible to access the value of the variable indirectly using the pointer

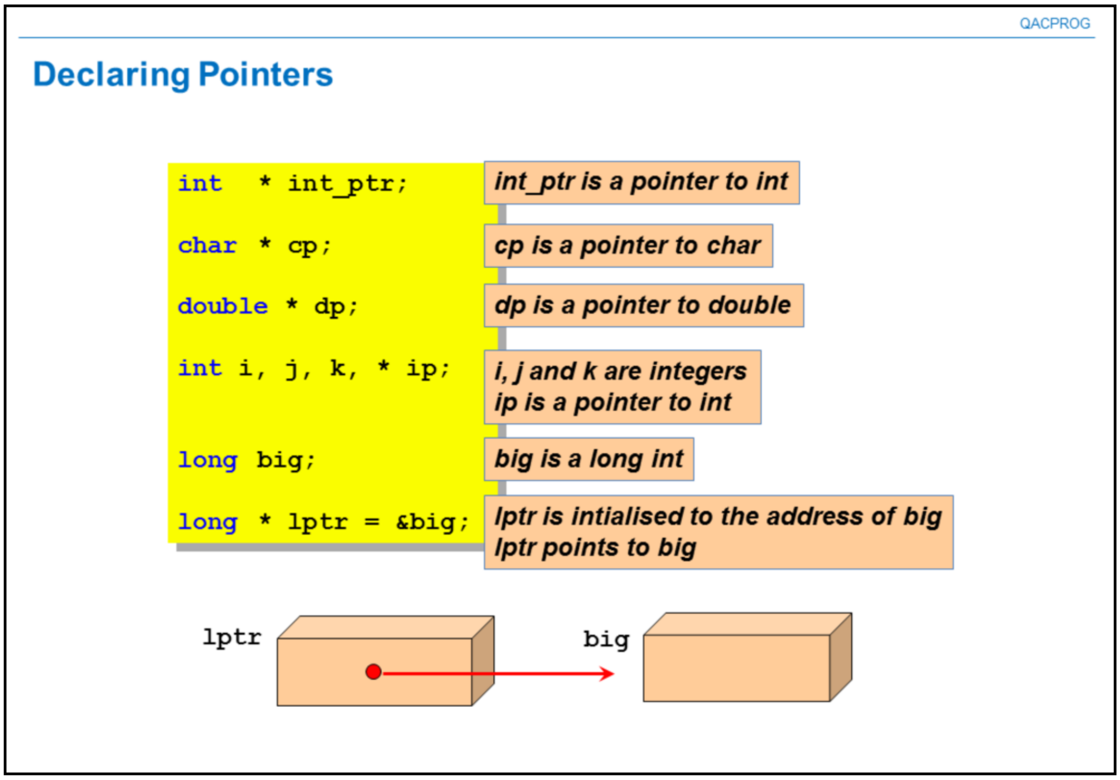


A pointer is a scalar data item (variable or constant). The value contained is an address. The address will be one known to the runtime environment and will contain data. It is possible for the value of the pointer to be another address that will contain data, etc.

Here is an analogy:

Example number	1	2	3
TYPE	`int`	`'pointer'`	`struct Card`
VARIABLE NAME	`x`	`ptr`	`mycard`
VALUE	`10`	`0x70A3`	`2, 'C'`

The pointer value (i.e. the address) is used to access the data in that address. This is an indirection, as `x`'s value is `10`, but `ptr`'s value is the address `0x70A3`. Presumably, `0x70A3` contains the required data.



QACPROG

Declaring Pointers

`int    * int_ptr;`	*int_ptr is a pointer to int*
`char   * cp;`	*cp is a pointer to char*
`double * dp;`	*dp is a pointer to double*
`int i, j, k, * ip;`	*i, j and k are integers* *ip is a pointer to int*
`long  big;`	*big is a long int*
`long  * lptr = &big;`	*lptr is intialised to the address of big* *lptr points to big*

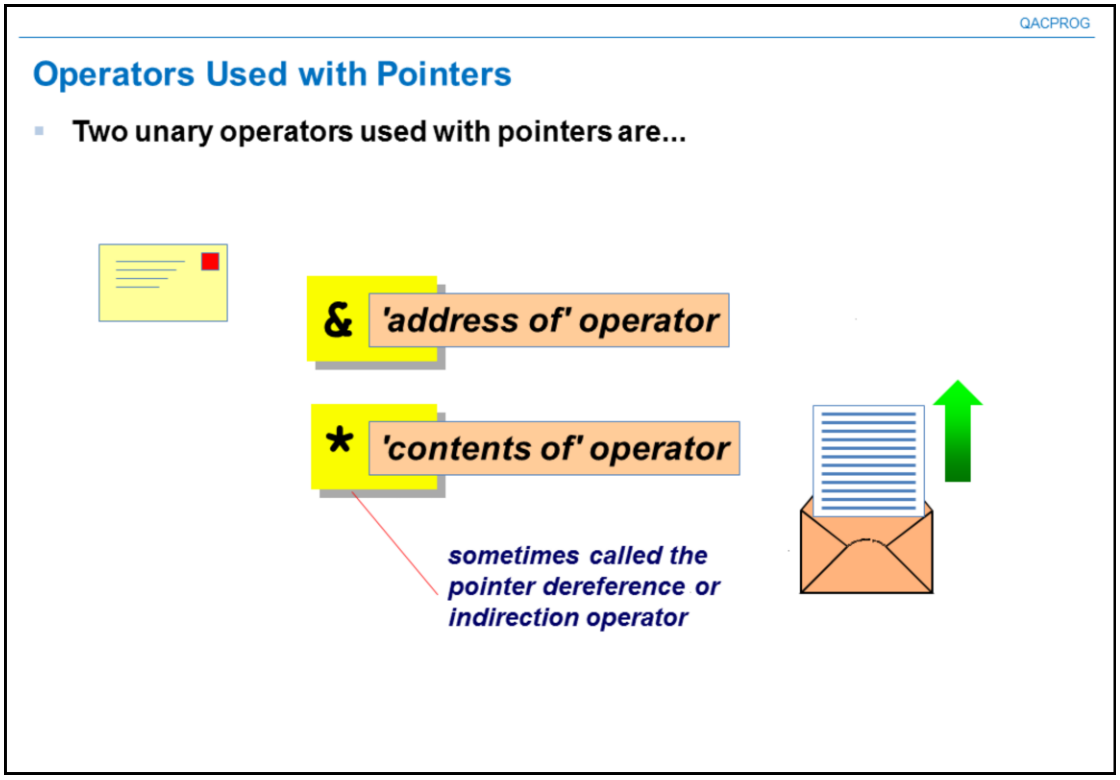
An individual pointer is constrained to contain addresses of only one type. The type is specified in the declaration. Therefore, a pointer is declared as a 'pointer to type'. The position of the `*` in the declaration is important, but any amount of white space can be used. It is advisable that the `*` should be close to the pointer's name and that pointers and 'ordinary' data items should not be declared in the same statement, as illustrated in the fourth declaration in the example.

Note that the last example includes an initialisation. The `&` operator is used to generate the address of its operand, the variable big.

Words and phrases are quite important when describing these items. Here are some examples:

> `lptr` is a pointer to `long`
> `lptr` has the value `&big`
> `lptr` points at `big`

Note: Throughout this chapter, the pointer will be declared as a variable, i.e. it can be updated during runtime. It is possible to declare `const` pointers.


QACPROG

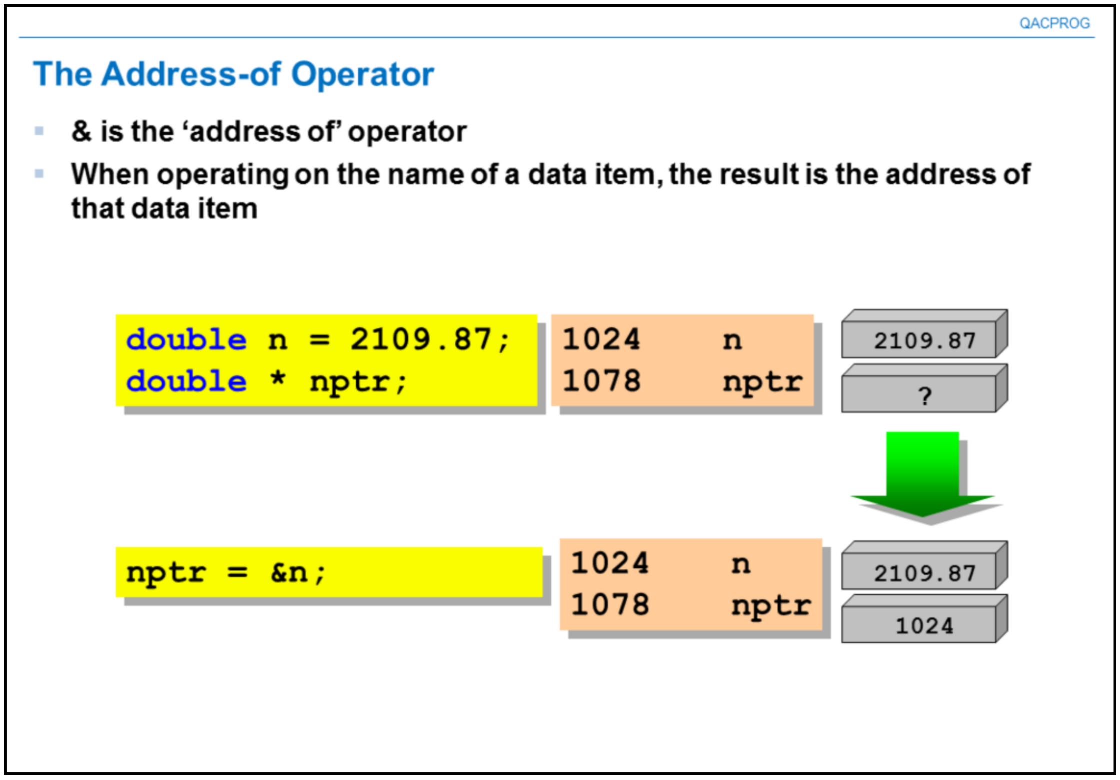
Operators Used with Pointers

- Two unary operators used with pointers are...

&
'address of' operator
*
'contents of' operator
sometimes called the pointer dereference or indirection operator

The `&` operator is old news. It generates the address of a previously-declared data item. This operator can only be applied to variable (or `const`) names; the compiler will pick up any misuse. Some examples are given later in the chapter.

The `*` operator is the one that does the work. It accesses the value contained in the address value of the pointer. This operator can be applied only to an expression whose value is an address; the compiler will check this too.



QACPROG

The Address-of Operator

- & is the 'address of' operator
- When operating on the name of a data item, the result is the address of that data item

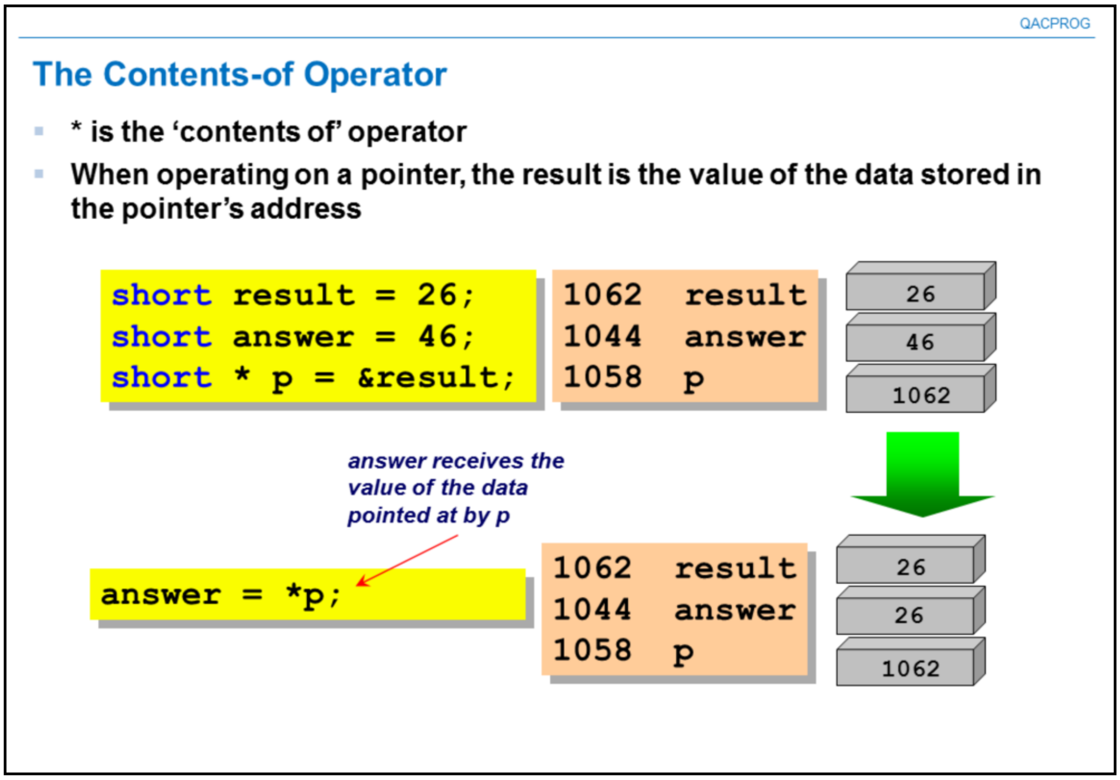
```
double n = 2109.87;
double * nptr;
```

```
nptr = &n;
```

The pointer `nptr` could be initialised when declared, in which case the second and third statements could be incorporated into the single data statement:

```
        double * nptr = &n ;
```

This is good programming practice. It is wise to give a value to a pointer as soon as possible. In a pointer, a garbage value would be interpreted as an address.



QACPROG

The Contents-of Operator

- * is the ‘contents of’ operator
- When operating on a pointer, the result is the value of the data stored in the pointer’s address

```
short result = 26;
short answer = 46;
short * p = &result;
```

```
1062   result
1044   answer
1058   p
```

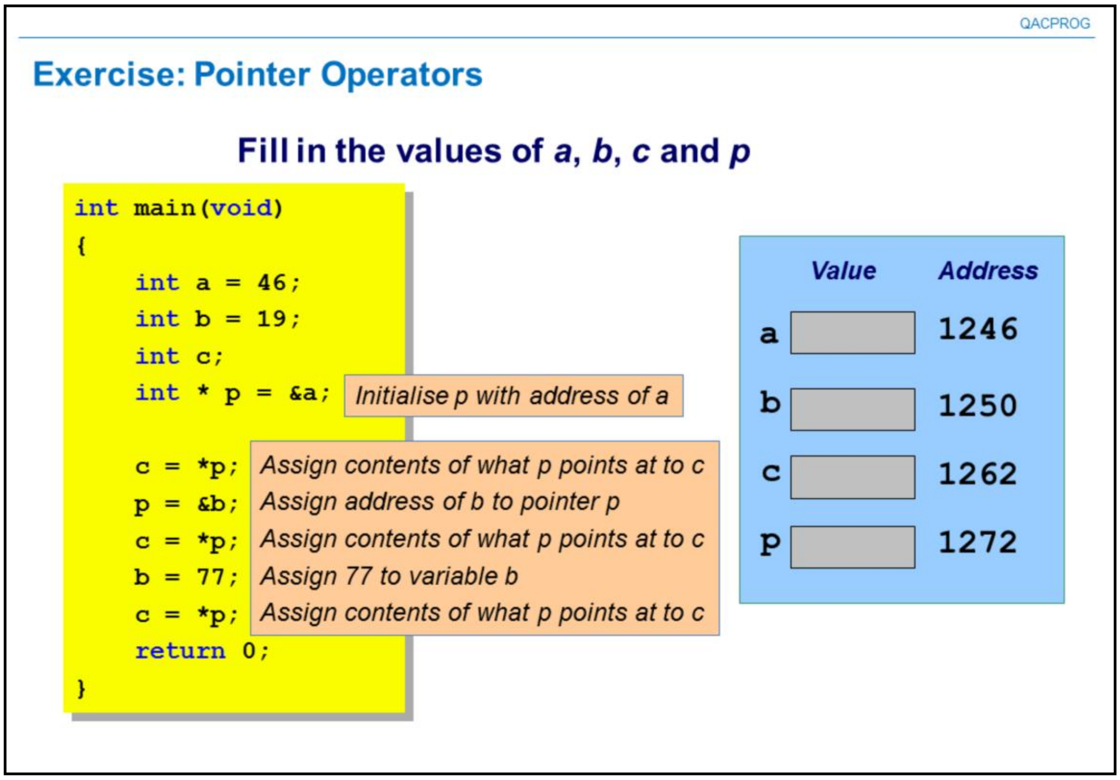
answer receives the value of the data pointed at by p

```
answer = *p;
```

```
1062   result
1044   answer
1058   p
```

The contents operator is usually applied to a pointer, as illustrated above. It has properties similar to that of the array and structure operators `[]` and `.`.

The expression `*p` can be used as a variable of that type.



QACPROG

Exercise: Pointer Operators

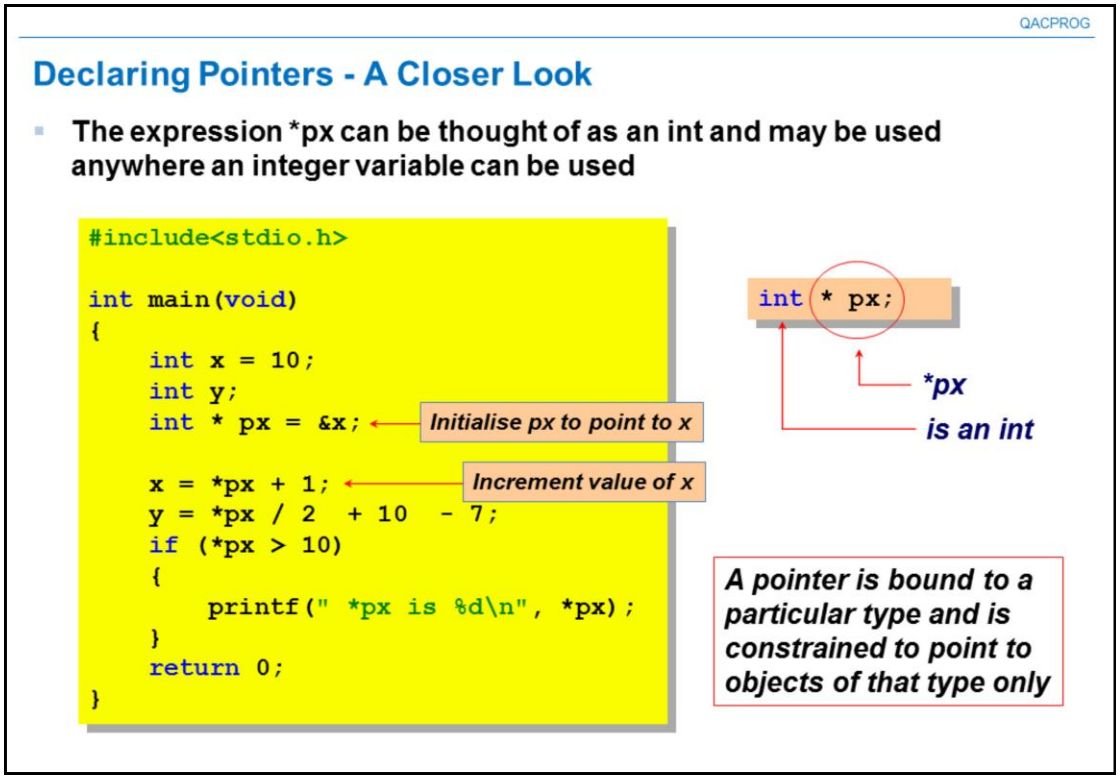
Fill in the values of *a*, *b*, *c* and *p*

```
int main(void)
{
    int a = 46;
    int b = 19;
    int c;
    int * p = &a;   Initialise p with address of a

    c = *p;   Assign contents of what p points at to c
    p = &b;   Assign address of b to pointer p
    c = *p;   Assign contents of what p points at to c
    b = 77;   Assign 77 to variable b
    c = *p;   Assign contents of what p points at to c
    return 0;
}
```

	Value	Address
a		1246
b		1250
c		1262
p		1272

The actual addresses are not important and the example values serve to indicate the relevant indirections.

The example is slightly contrived, but provides a useful exercise on the syntax of handling pointers. It is unusual for data to be accessed both by name and via a pointer, and is bad style. In the vast majority of cases the data and pointer are defined in different functions so that the choice of access directly or indirectly is not available.



QACPROG

Declaring Pointers - A Closer Look

- The expression *px can be thought of as an int and may be used anywhere an integer variable can be used

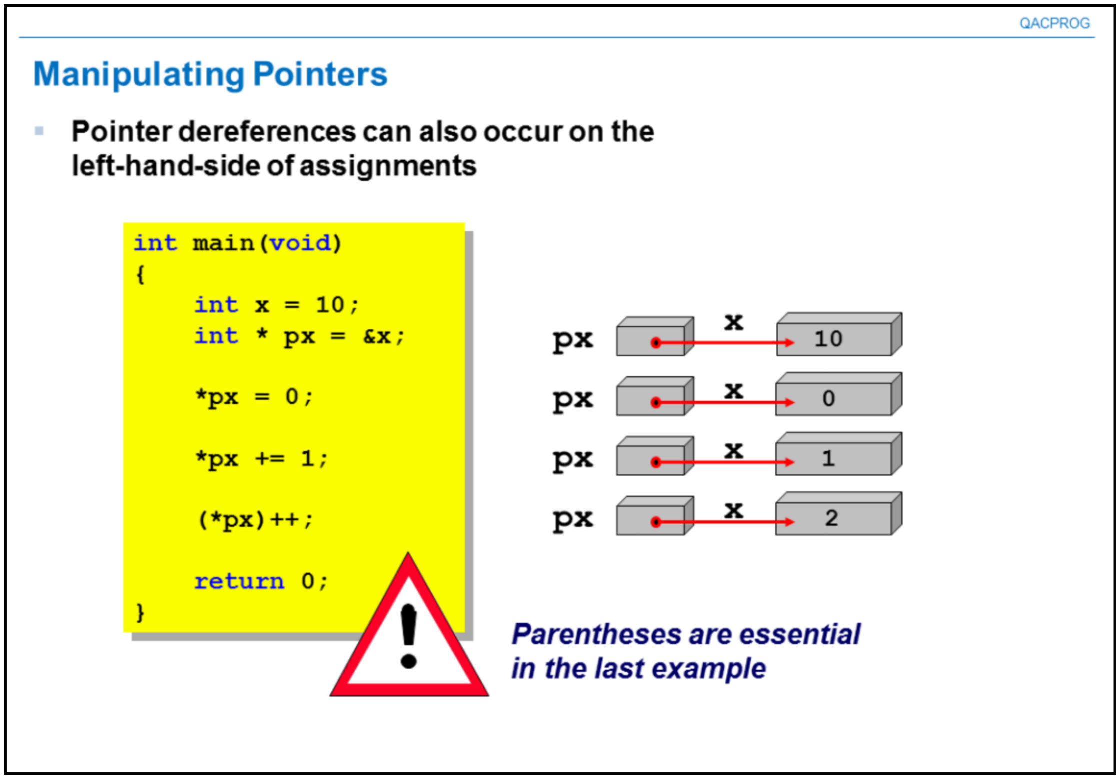
```
#include<stdio.h>

int main(void)
{
    int x = 10;
    int y;
    int * px = &x;

    x = *px + 1;
    y = *px / 2  + 10  - 7;
    if (*px > 10)
    {
        printf(" *px is %d\n", *px);
    }
    return 0;
}
```

Initialise px to point to x
Increment value of x
int * px;
*px
is an int

A pointer is bound to a particular type and is constrained to point to objects of that type only

This slide illustrates the fact that the dereferenced item `*px` can be used as an integer in its own right (cf. the integer items `a[i]` and `today.day`). Note that the `*` operator, surprisingly, does not share the highest precedence with the other access operators. This may prove to be a problem.

Like the expressions `a[i]` and `today.day`, `*px` can also be assigned to.



QACPROG

Manipulating Pointers

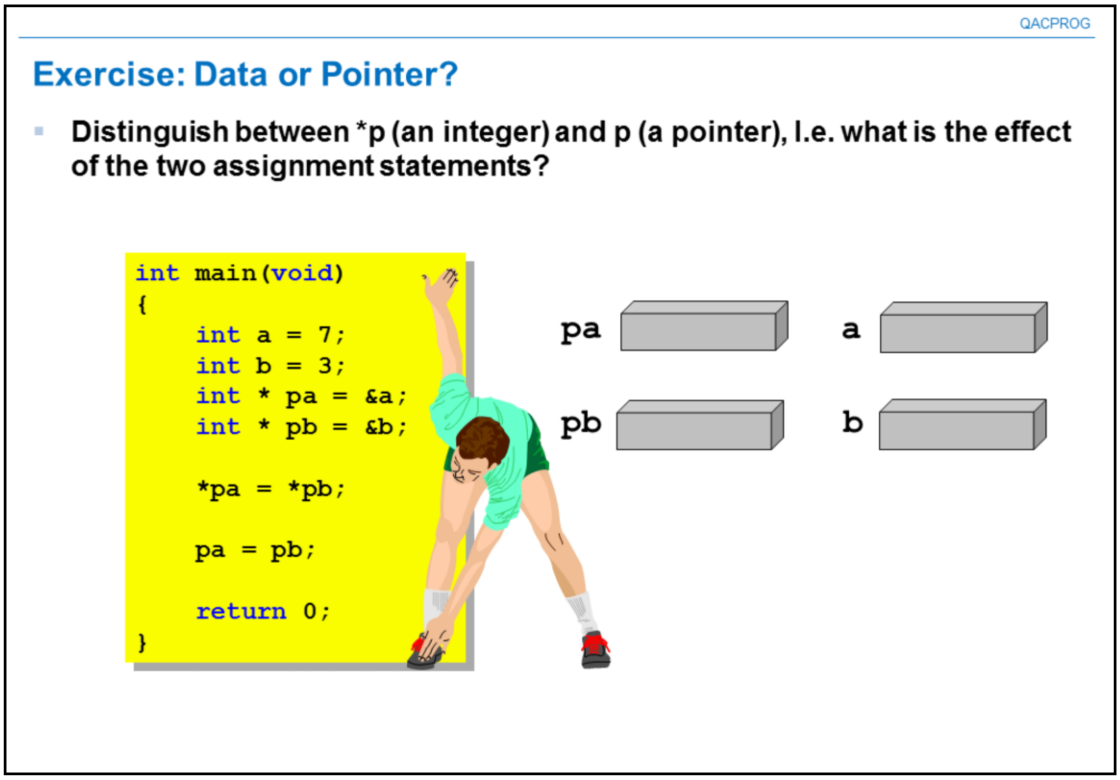
- Pointer dereferences can also occur on the left-hand-side of assignments

```
int main(void)
{
    int x = 10;
    int * px = &x;

    *px = 0;

    *px += 1;

    (*px)++;

    return 0;
}
```

Parentheses are essential in the last example

`*px` can be thought of as an integer variable. One can assign to it!

The most common and most evasive logical errors in C programs are caused by programmers who misinterpret precedence and associativity of pointer operations.



QACPROG

Exercise: Data or Pointer?

- Distinguish between *p (an integer) and p (a pointer), I.e. what is the effect of the two assignment statements?

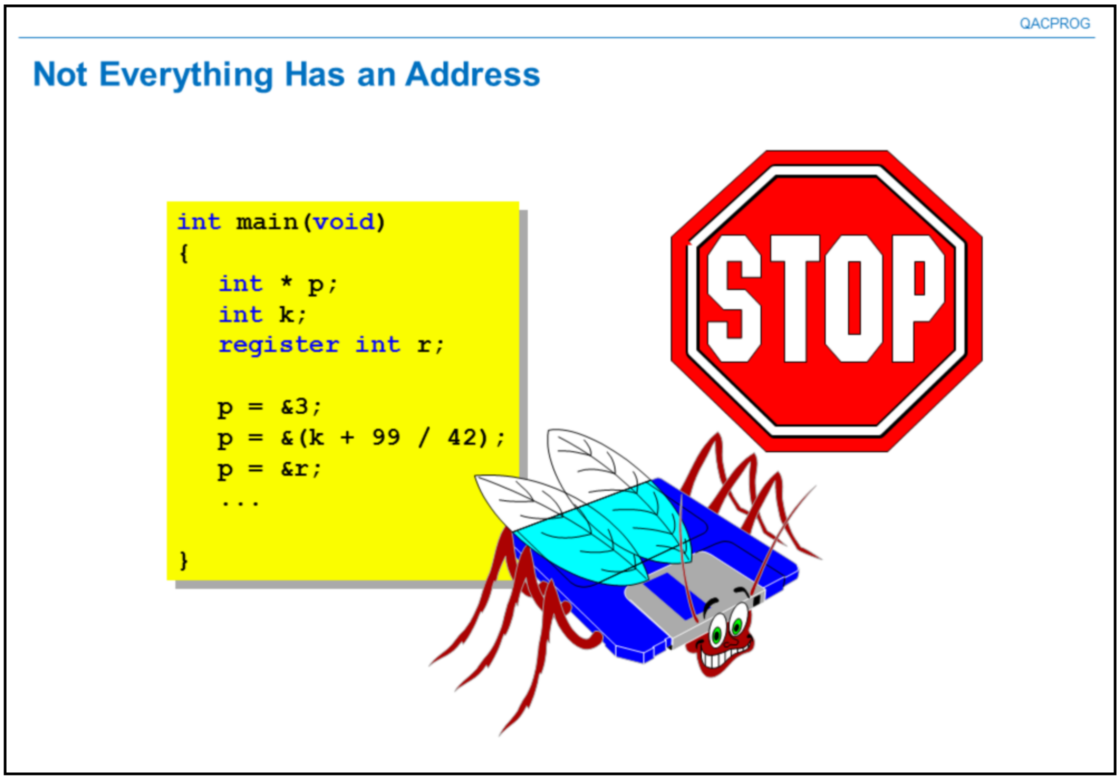
```
int main(void)
{
    int a = 7;
    int b = 3;
    int * pa = &a;
    int * pb = &b;

    *pa = *pb;

    pa = pb;

    return 0;
}
```

Pointers are data items whose values are addresses. If a pointer is accessed for manipulation, it will be an address that is being processed. Once a pointer has been dereferenced using the * operator, it is the data pointed at by the pointer that is being processed.

The compiler will warn you about mismatches such as `*pa = pb`.

Solution: The statement `*pa = *pb` is one of integer assignment, as both `*pa` and `*pb` are `int`s. The effect is to assign `3` to `a`.

However, the statement `pa = pb` is one of pointer to integer assignment, as both `pa` and `pb` are pointers to `int`s. The effect is to assign `b`'s address to `pa`, i.e. both pointers now point as `b`.



QACPROG

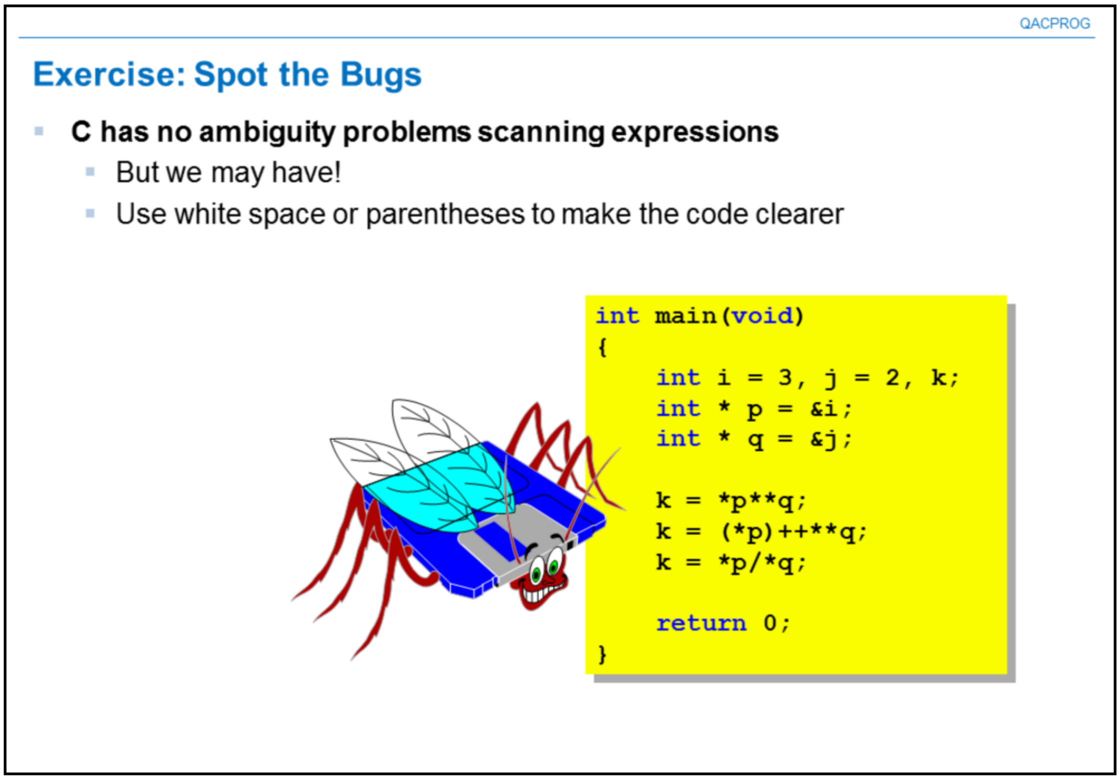
Not Everything Has an Address

```
int main(void)
{
   int * p;
   int k;
   register int r;

   p = &3;
   p = &(k + 99 / 42);
   p = &r;
   ...

}
```

The rule is that `&` can have only a declared data item (with a guaranteed memory location) as its operand, i.e. a static or auto variable or a constant. The examples shown above are illegal for the following reasons:

- `r` could be in a register, not a memory location.
- `3` is constant; it has no memory location accessible to the programmer.
- `(k + 99/42)` is an expression; the value of k is uninitialised. Assume k happens to contain a bit pattern representing the integer 2 - in this the value of the whole expression would be the integer constant 2 (remember integer division produces an integer result). Its runtime value is *not* a variable which contains the number 2, it is actually the number 2. This will be true whatever the value of k.

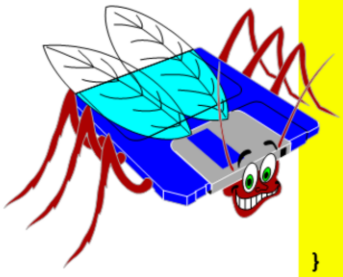
A bit field is a structure (or `union`) member that does not use a complete memory location; only a few bits of one. It would probably share a location with other bitfields. Its address is not accessible to the programmer.



QACPROG

Exercise: Spot the Bugs

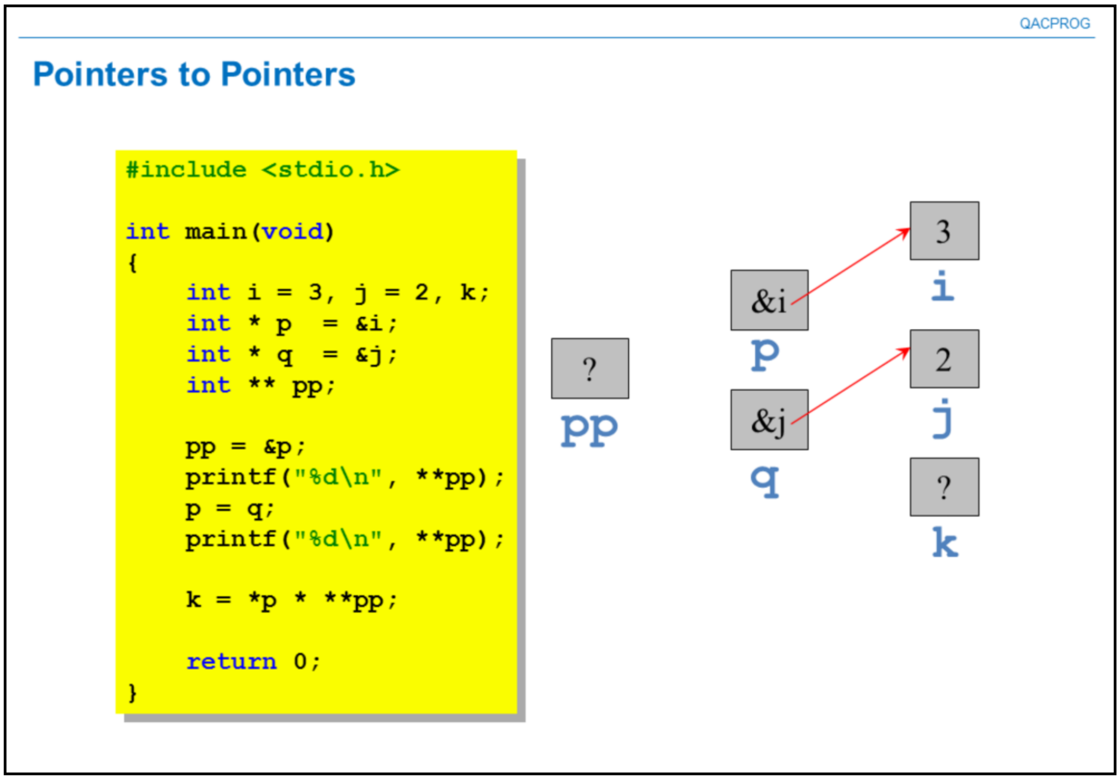
- **C has no ambiguity problems scanning expressions**
  - But we may have!
  - Use white space or parentheses to make the code clearer



```
int main(void)
{
    int i = 3, j = 2, k;
    int * p = &i;
    int * q = &j;

    k = *p**q;
    k = (*p)++**q;
    k = *p/*q;

    return 0;
}
```

The code statements should be written as follows:

```
k = *p * *q ;

k = (*p)++ * *q ;

k = *p / *q ;
```

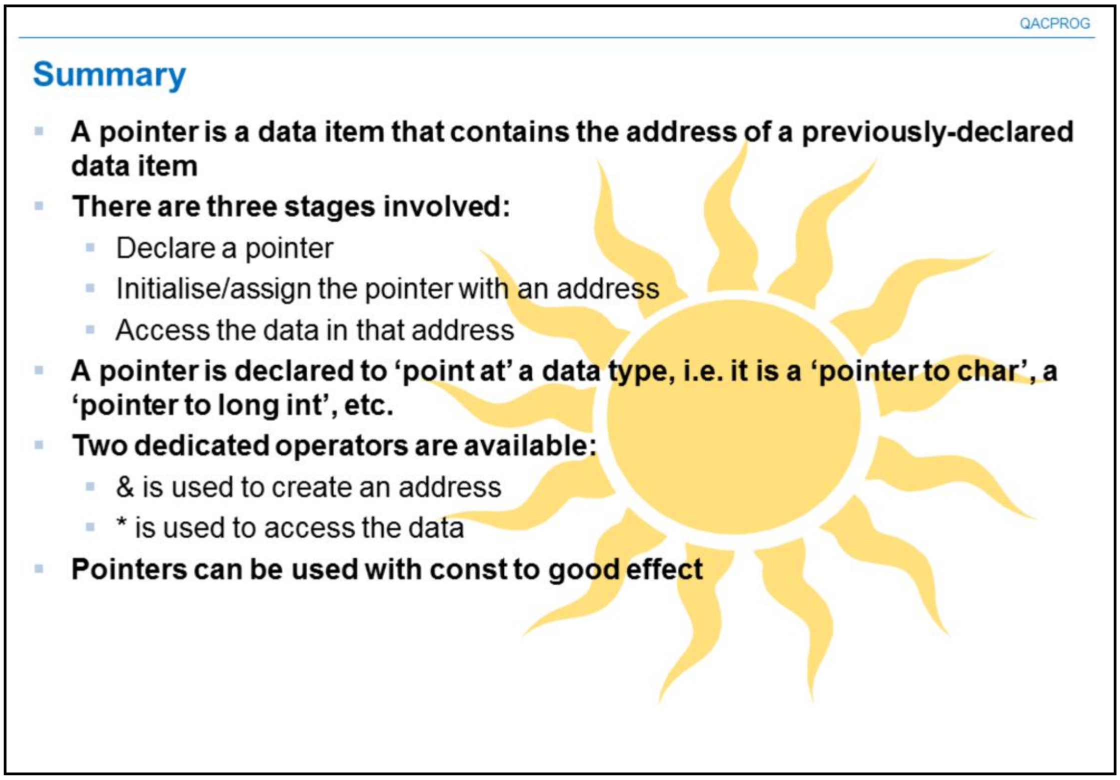


QACPROG

Pointers to Pointers

```
#include <stdio.h>

int main(void)
{
    int i = 3, j = 2, k;
    int * p  = &i;
    int * q  = &j;
    int ** pp;

    pp = &p;
    printf("%d\n", **pp);
    p = q;
    printf("%d\n", **pp);

    k = *p * **pp;

    return 0;
}
```

The `int ** pp` declaration should be read as `int *(*pp).` This indicates that `pp` is a pointer, because it has a single `*` right up close to it. The remaining `int *` in the declaration conjures up the idea of a pointer to int. So `pp` is a pointer to ... a pointer to `int`. The brackets can be included. The runtime expression `**pp` in the `printf` call can be interpreted in a similar manner.



QACPROG

Summary

- A pointer is a data item that contains the address of a previously-declared data item
- There are three stages involved:
  - Declare a pointer
  - Initialise/assign the pointer with an address
  - Access the data in that address
- A pointer is declared to 'point at' a data type, i.e. it is a 'pointer to char', a 'pointer to long int', etc.
- Two dedicated operators are available:
  - & is used to create an address
  - * is used to access the data
- Pointers can be used with const to good effect

A pointer, like all data items, has a name, a type and a value. The name, like all names, must obey the rules of all legal identifiers. The type is best described as a 'pointer to a specified type', where the specified type is any C scalar, including pointers, aggregates or user-defined types. The value is always a machine address; usually the address of a data item.

This chapter has introduced the concepts, without covering the major uses. The chapters that follow on from this cover the use of pointers with functions (call by reference), pointers with arrays (efficiency) and pointers with structures (call by reference).