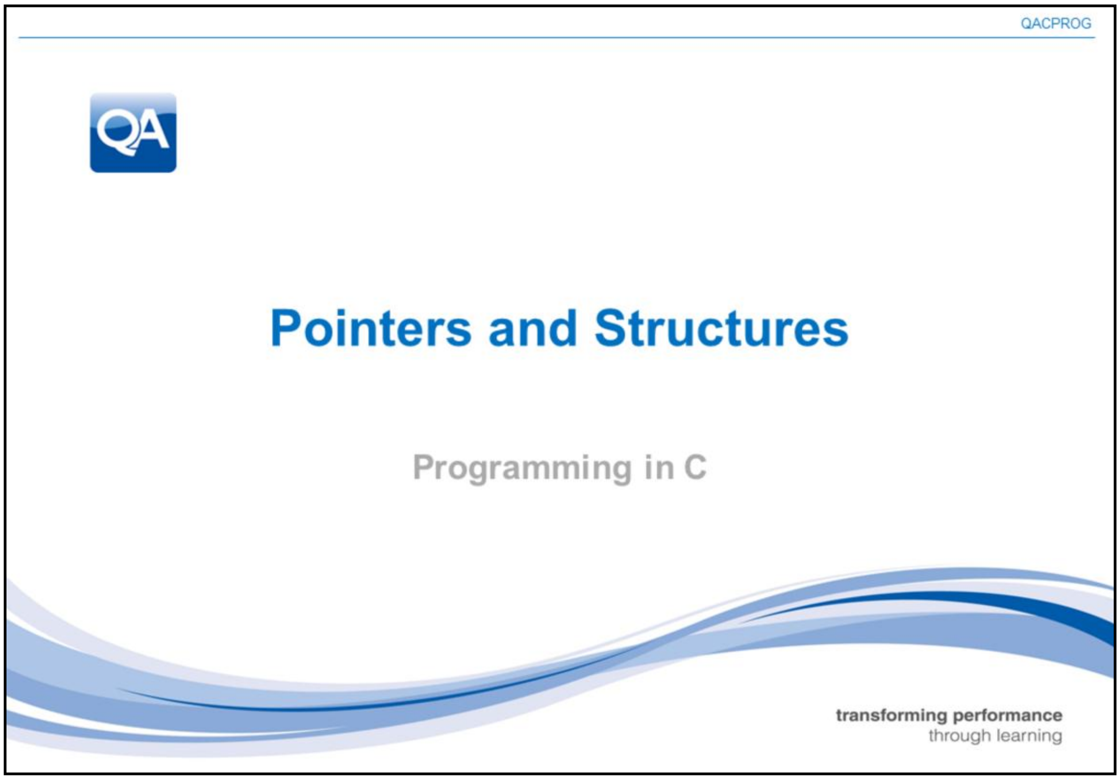
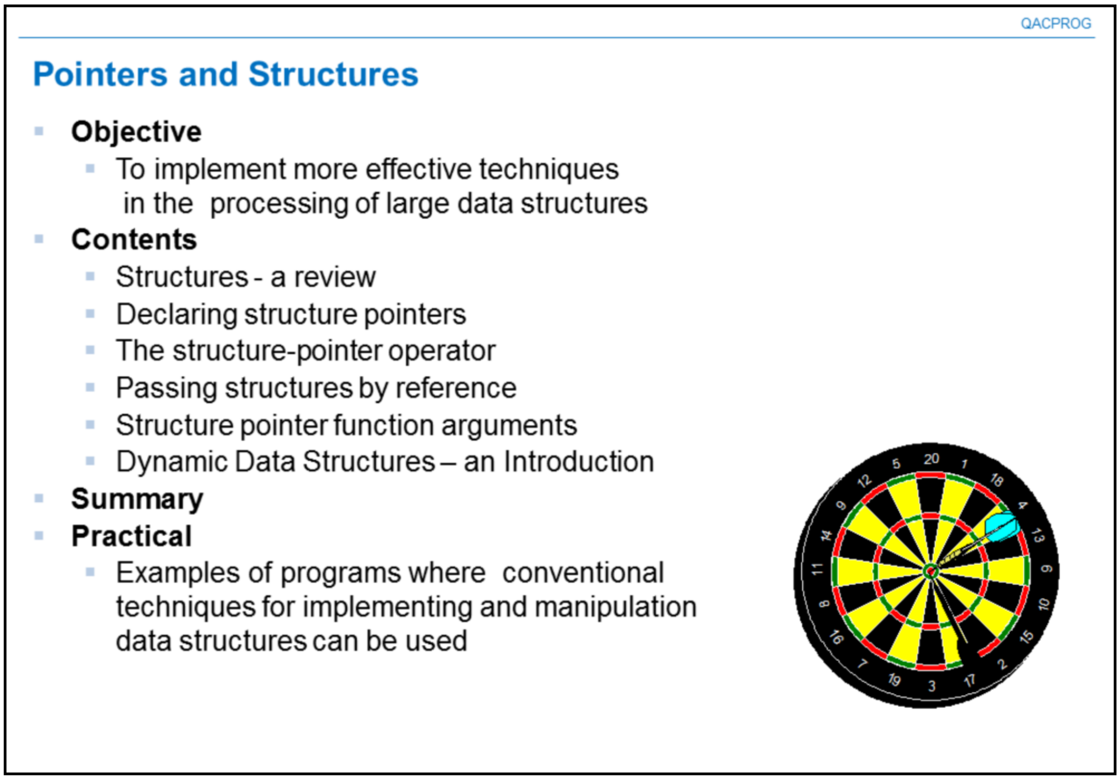


QACPROG

Pointers and Structures

Programming in C

transforming performance
through learning



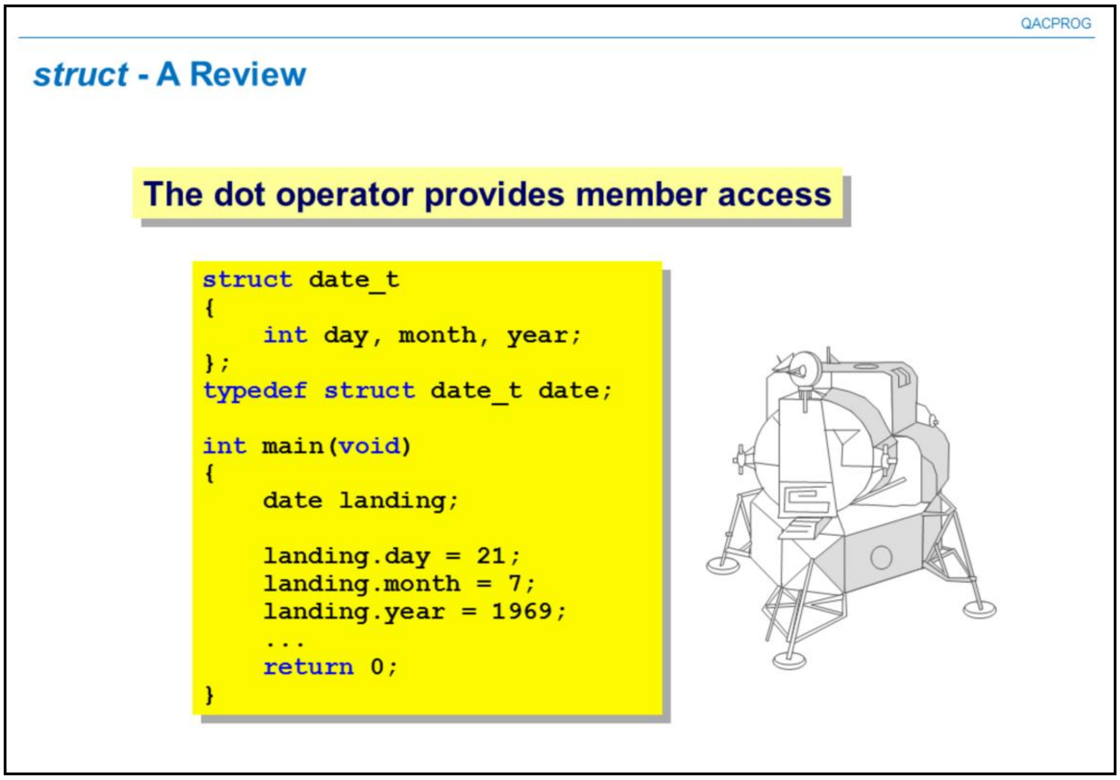
QACPROG

Pointers and Structures

- **Objective**
  - To implement more effective techniques in the processing of large data structures
- **Contents**
  - Structures - a review
  - Declaring structure pointers
  - The structure-pointer operator
  - Passing structures by reference
  - Structure pointer function arguments
  - Dynamic Data Structures – an Introduction
- **Summary**
- **Practical**
  - Examples of programs where conventional techniques for implementing and manipulation data structures can be used

The objective of this chapter is to introduce the call-by-reference mechanism for structures. This is achieved by considering pointers to structure data items.

The address of a structure can also be used as a member of a structure. It is this that opens up the idea of data structures that consist of collections of items, often called nodes, which are able to reference their neighbours. Such data structures are often called dynamic data structures, because it is relatively straightforward to maintain the insertion and removal of items at runtime.

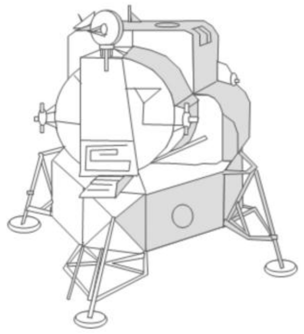


QACPROG

struct - A Review

The dot operator provides member access

```
struct date_t
{
    int day, month, year;
};
typedef struct date_t date;

int main(void)
{
    date landing;

    landing.day = 21;
    landing.month = 7;
    landing.year = 1969;
    ...
    return 0;
}
```



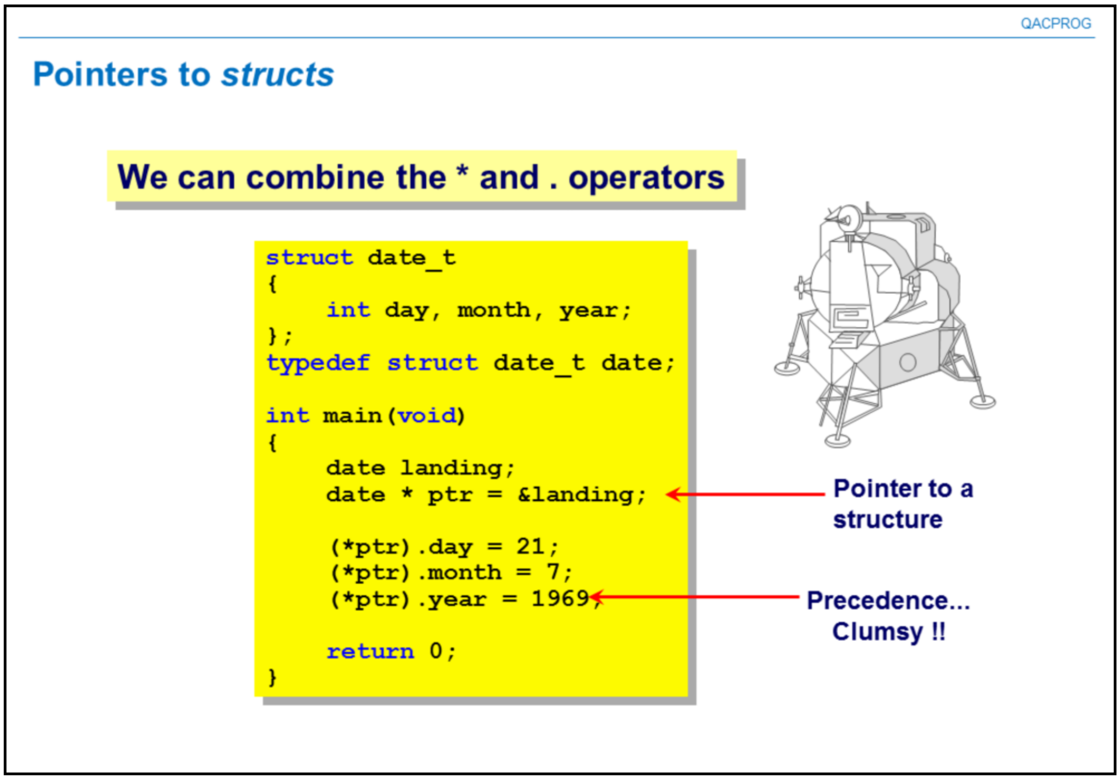
The program shown above reminds us of the three stages in the implementation and use of a structure:

Define the template - a blueprint from which other structures are created.
Declare a variable - a data item to house an individual set of data (this could be initialised).
Access the data members.

The dot operator is powerful. It takes a variable name and the name of the element. The underlying code needs to perform the following:

Get the address from the data name (equivalent to applying the `&` operator).
From the template, obtain information on how the members are organised in memory.
Grab hold of the data member from the above information using the member's name.

If we have the address, this process is easier, since the dereferencing has already taken place.



QACPROG

Pointers to *structs*

We can combine the * and . operators

```
struct date_t
{
    int day, month, year;
};
typedef struct date_t date;

int main(void)
{
    date landing;
    date * ptr = &landing;

    (*ptr).day = 21;
    (*ptr).month = 7;
    (*ptr).year = 1969;

    return 0;
}
```

Pointer to a structure

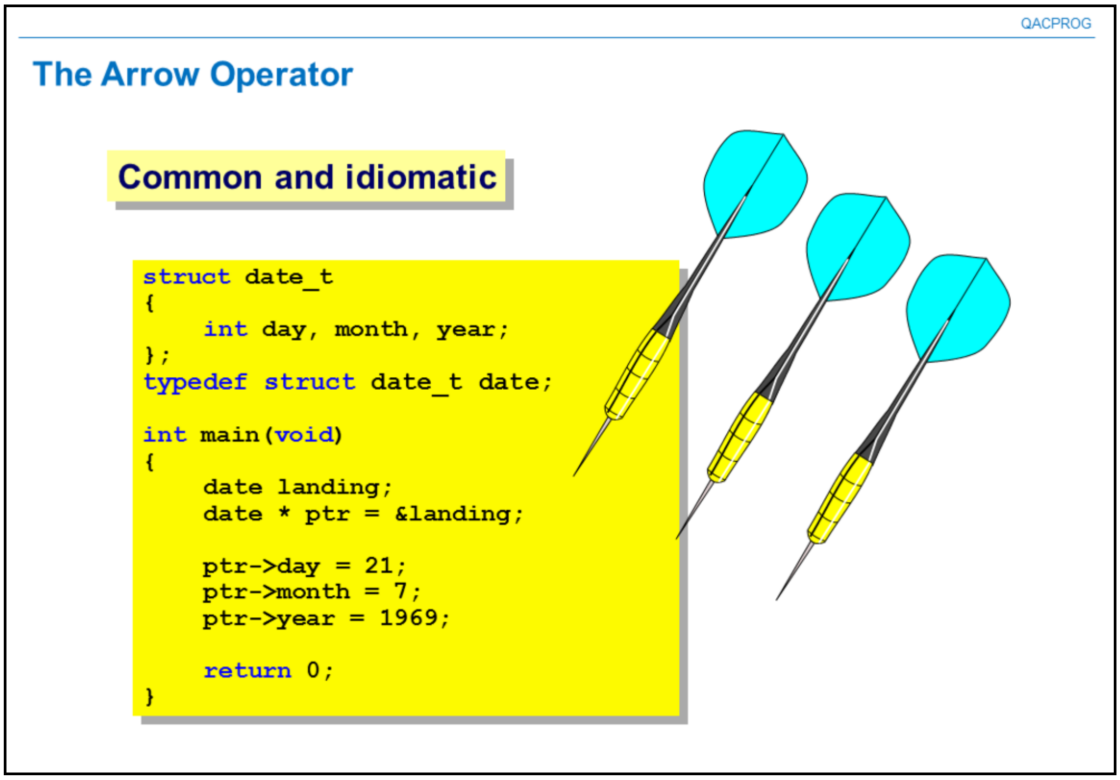
Precedence... Clumsy !!

The address is provided by applying the `&` operator to the data name. Note that the resulting address is then placed in the appropriate pointer, namely a pointer to the `struct date`. We have used &landing to initialise the pointer.

The notation `(*ptr).membername` is clumsy, because we appear to be taking a step backwards to go forwards. We have the address of the structure, so why not use that to access the data? To achieve the short cut, another operator is required. It will need to do the last two items in the list described on the last page, i.e.:

Get the address from the data name (equivalent to applying the `&` operator).
From the template, obtain information on how the members are organised in memory.
Grab hold of the data member from the above information using the member's name.

This is a description of the arrow operator, `->` .



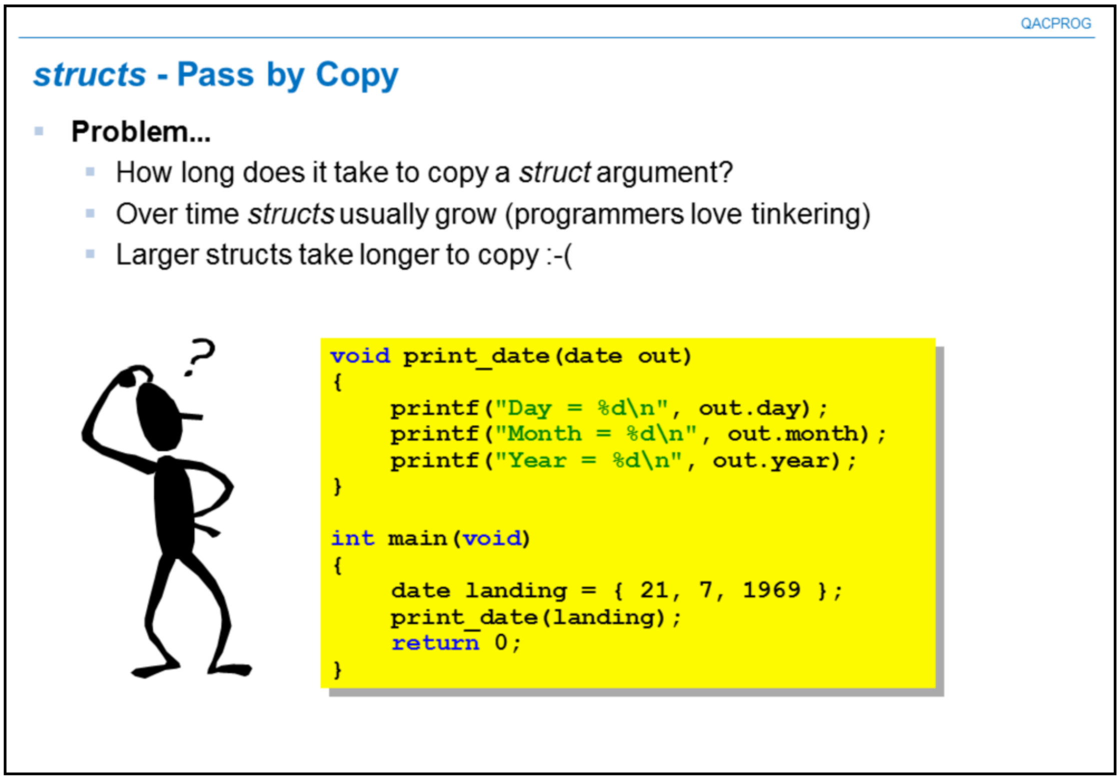
QACPROG

The Arrow Operator

Common and idiomatic

```
struct date_t
{
    int day, month, year;
};
typedef struct date_t date;

int main(void)
{
    date landing;
    date * ptr = &landing;

    ptr->day = 21;
    ptr->month = 7;
    ptr->year = 1969;

    return 0;
}
```

The example shown above illustrates the syntax of the -> operator only. It is not a realistic example, because the data item today is in the same scope as the code using the address, but we have not covered functions with structures yet!



QACPROG

structs - Pass by Copy

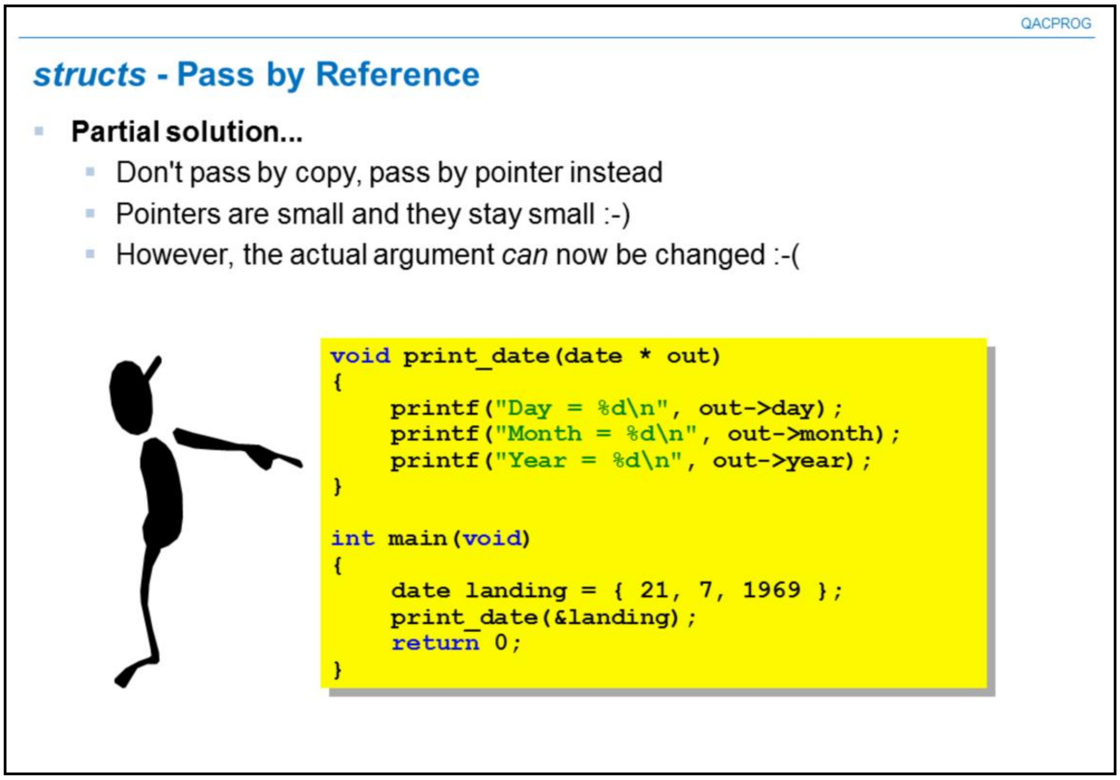
- **Problem...**
  - How long does it take to copy a *struct* argument?
  - Over time *structs* usually grow (programmers love tinkering)
  - Larger structs take longer to copy :-(

```
void print_date(date out)
{
    printf("Day = %d\n", out.day);
    printf("Month = %d\n", out.month);
    printf("Year = %d\n", out.year);
}

int main(void)
{
    date landing = { 21, 7, 1969 };
    print_date(landing);
    return 0;
}
```

The time taken to copy a struct depends on how big the struct is. In other words how many members the struct definition contains. Smaller structs can be passed as arguments faster than large structs. Note that we are not talking about how long the called function takes to actually run - we are just talking about *making* the call.

If a struct definition acquires more members, or if the existing members (which could themselves be structs or arrays) get bigger the time taken to pass struct variables as function arguments will increase. The program will get slower and slower.

Unfortunately, structs are more likely to grow than to shrink. It's called maintenance.



QACPROG

structs - Pass by Reference

- **Partial solution...**
  - Don't pass by copy, pass by pointer instead
  - Pointers are small and they stay small :-)
  - However, the actual argument *can* now be changed :-(

```
void print_date(date * out)
{
    printf("Day = %d\n", out->day);
    printf("Month = %d\n", out->month);
    printf("Year = %d\n", out->year);
}

int main(void)
{
    date landing = { 21, 7, 1969 };
    print_date(&landing);
    return 0;
}
```

If it hurts to pass by copy then don't do it! Instead, pass by reference using pointers. A pointer contains only an memory address. The struct located at that memory address might be huge, the pointer itself is guaranteed to be small. Hence the time taken to pass a pointer will be constant and will not depend on the size of the struct itself. The struct can increase in size as more members are added but the time taken to pass it by reference will not increase at all.

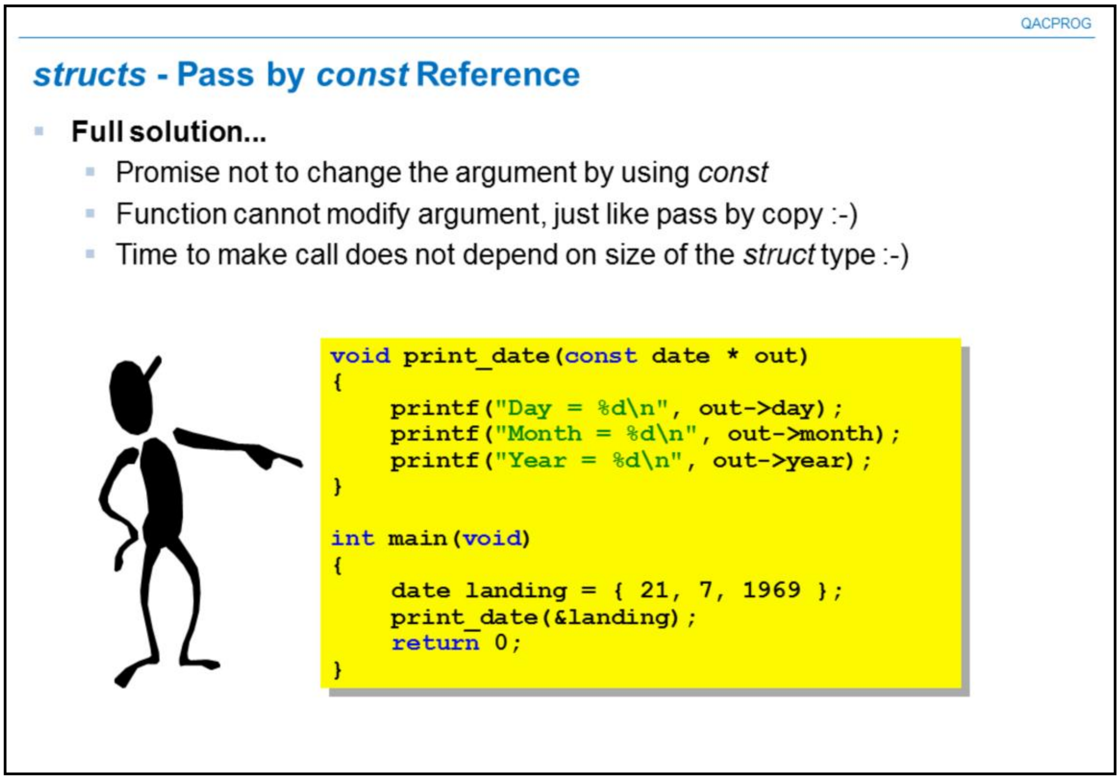
The function has complete freedom to access the `landing` structure via its address. Logically `print_date` does not *need* to modify `landing`. Yet a statement inside the function definition *could* modify `landing` - it would compile. Remember too that a function can be called with only the prototype visible (or perhaps not even that). This means the compiler has to *assume* that `landing` *will* be modified. This can cause problems. For example, suppose `landing` had been declared as a `const date` (which would be entirely reasonable)

```
const date landing = { 21, 7, 1969 };
```

The effect would be that the following line:

```
print_date(&landing);
```

would no longer compile. This would be very unreasonable and needlessly restrictive.



QACPROG

structs - Pass by *const* Reference

- **Full solution...**
  - Promise not to change the argument by using *const*
  - Function cannot modify argument, just like pass by copy :-)
  - Time to make call does not depend on size of the *struct* type :-)

```
void print_date(const date * out)
{
    printf("Day = %d\n", out->day);
    printf("Month = %d\n", out->month);
    printf("Year = %d\n", out->year);
}

int main(void)
{
    date landing = { 21, 7, 1969 };
    print_date(&landing);
    return 0;
}
```

Notice that the definition of print_date has changed. The single parameter is typed as a pointer to a date struct that is "const". This is an unfortunate overloading of the keyword const. A const used like this - on the "other-side" of a pointer does not mean const! It means readonly. The actual date variable pointed to for a particular call may or may not be const and in the slide landing is not const. In other words landing can change. That is not the issue. The issue is whether print_date itself is allowed to indirectly (through out) modify landing. It is not. The print_date function has readonly access to landing (for that call) through the pointer out.

Note that there is nothing wrong with having both read and write access through a pointer. Indeed sometimes it is required. For example:

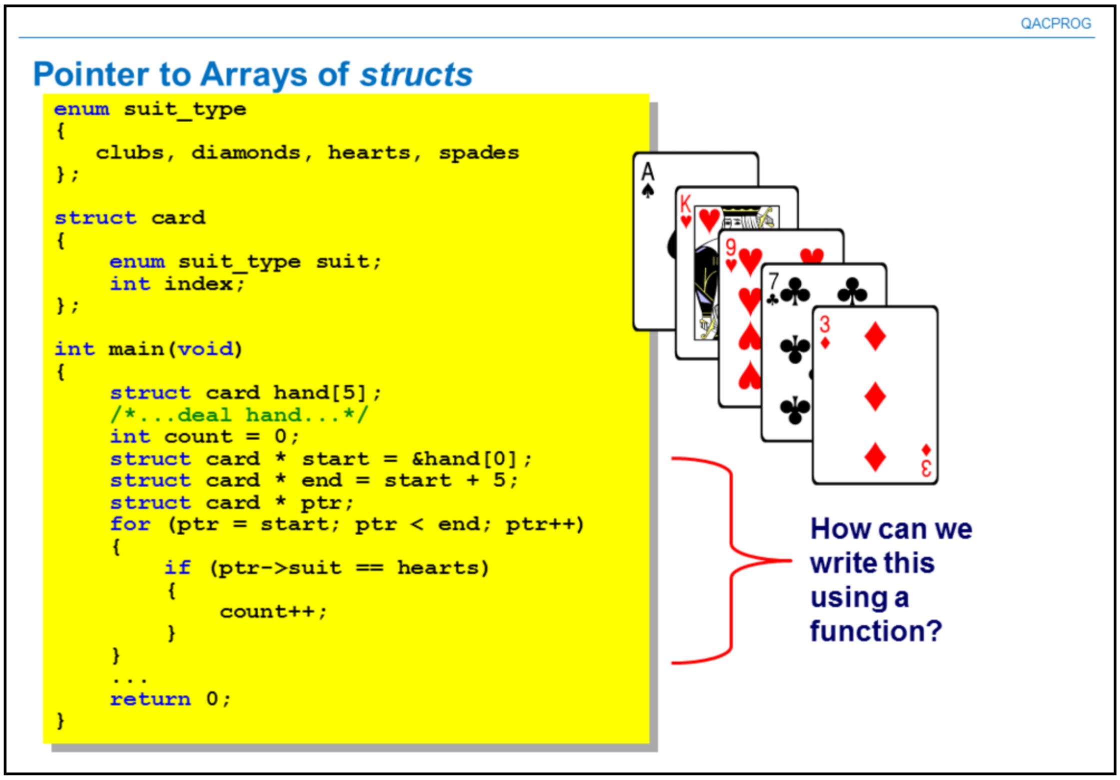
```
void set_date(date *, int, int, int);
```

This allows the following (which does not depend on the types or names of the date members):

```
date hangover;
set_date(&hangover, 1, 1, 2000);
```

The definition would be:

```
void set_date(date * ptr, int d, int m, int y)
{
   ptr->day   = d;
   ptr->month = m;
   ptr->year  = y;
}
```



QACPROG

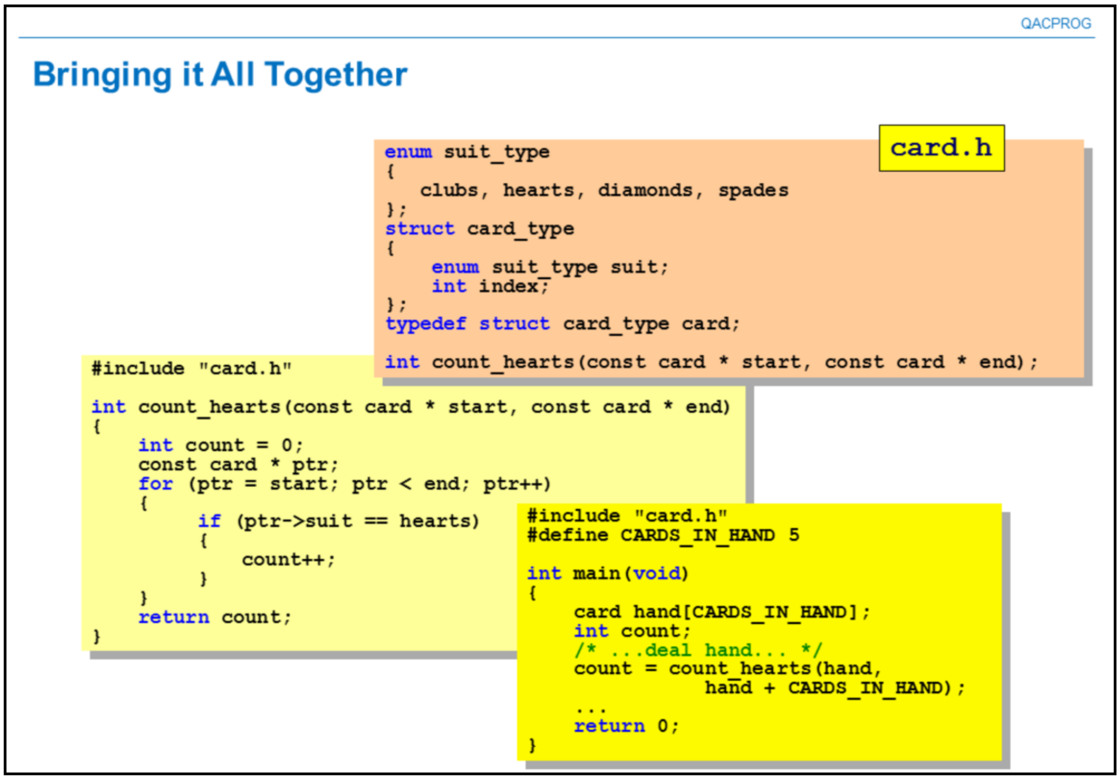
Pointer to Arrays of *structs*

```
enum suit_type
{
   clubs, diamonds, hearts, spades
};

struct card
{
     enum suit_type suit;
     int index;
};

int main(void)
{
     struct card hand[5];
     /*...deal hand...*/
     int count = 0;
     struct card * start = &hand[0];
     struct card * end = start + 5;
     struct card * ptr;
     for (ptr = start; ptr < end; ptr++)
     {
          if (ptr->suit == hearts)
          {
               count++;
          }
     }
     ...
     return 0;
}
```

Note the use of the end pointer; something we saw in the *Pointers and Arrays* chapter.

Solution:

```
int main(void)
{
     struct card hand[5];
     /*...deal hand...*/
     int count = 0;
     size_t card;

     for (card = 0; card < 5; card++)
     {
          if (hand[card].suit == hearts)
          {
               count++;
          }
     }
     ...
}
```


QACPROG

Bringing it All Together

card.h

```
enum suit_type
{
    clubs, hearts, diamonds, spades
};
struct card_type
{
    enum suit_type suit;
    int index;
};
typedef struct card_type card;

int count_hearts(const card * start, const card * end);
```

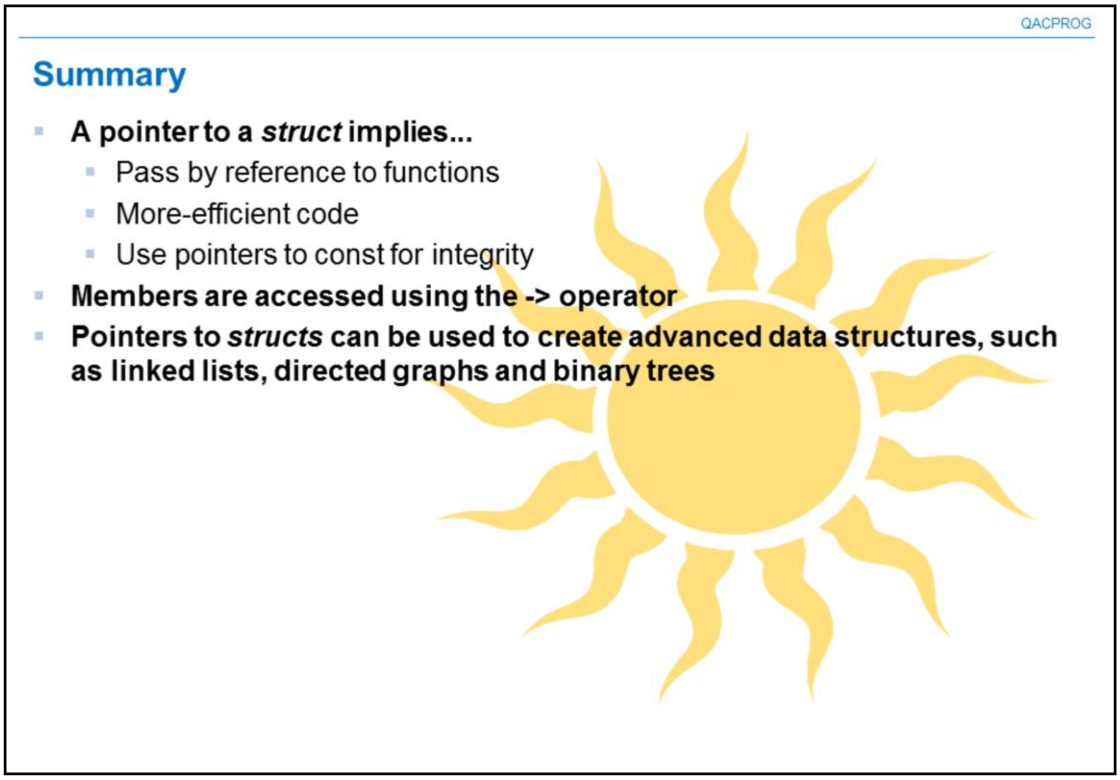
```
#include "card.h"

int count_hearts(const card * start, const card * end)
{
    int count = 0;
    const card * ptr;
    for (ptr = start; ptr < end; ptr++)
    {
        if (ptr->suit == hearts)
        {
            count++;
        }
    }
    return count;
}
```

```
#include "card.h"
#define CARDS_IN_HAND 5

int main(void)
{
    card hand[CARDS_IN_HAND];
    int count;
    /* ...deal hand... */
    count = count_hearts(hand,
                hand + CARDS_IN_HAND);
    ...
    return 0;
}
```

This example is an excellent one for illustrating program organisation. The header file holds a structure template and a prototype.

There are a few ways we might consider improving this code. Firstly, the sequence of cards specified by [start,end) is not modified inside the `count_hearts` function. Secondly `count_hearts` must by definition return a non negative integer. Finally, the function is needlessly hardwired to count `hearts` - we can make the suit a function parameter:

```
size_t
count_suit(const card * start, const card * end,
           enum suit_type suit)
{
    size_t count = 0;
    const card * current;
    for (current = start; current < end; current++)
    {
        if (current->suit == suit)
        {
            count++;
        }
    }
    return count;
}
```



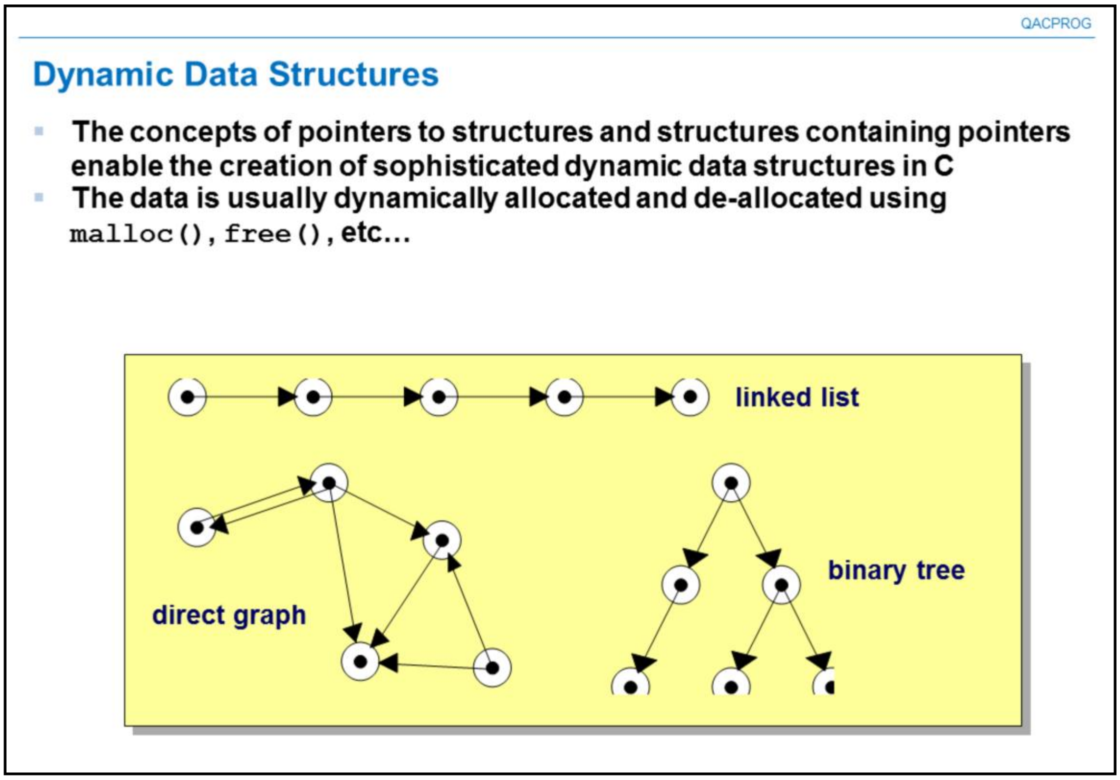
QACPROG

Summary

- **A pointer to a *struct* implies...**
  - Pass by reference to functions
  - More-efficient code
  - Use pointers to const for integrity
- **Members are accessed using the -> operator**
- **Pointers to *structs* can be used to create advanced data structures, such as linked lists, directed graphs and binary trees**

For any non trivial system implemented in C it is essential to master the art of pointers to structures.

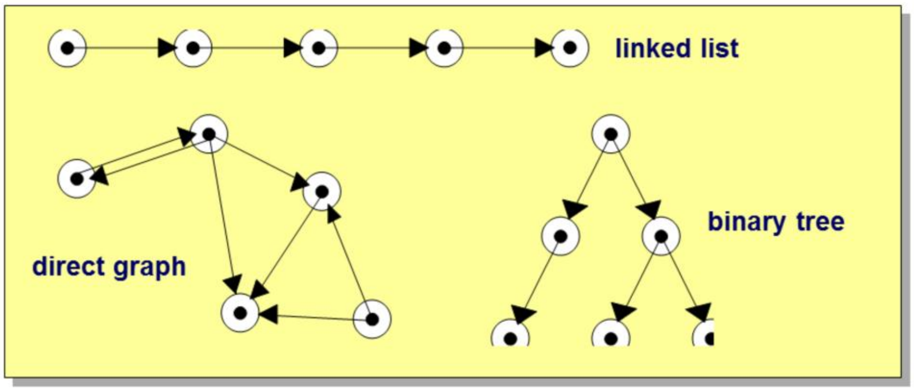
The whole concept of dynamic data structures depends on a knowledge of this subject.



QACPROG

Dynamic Data Structures

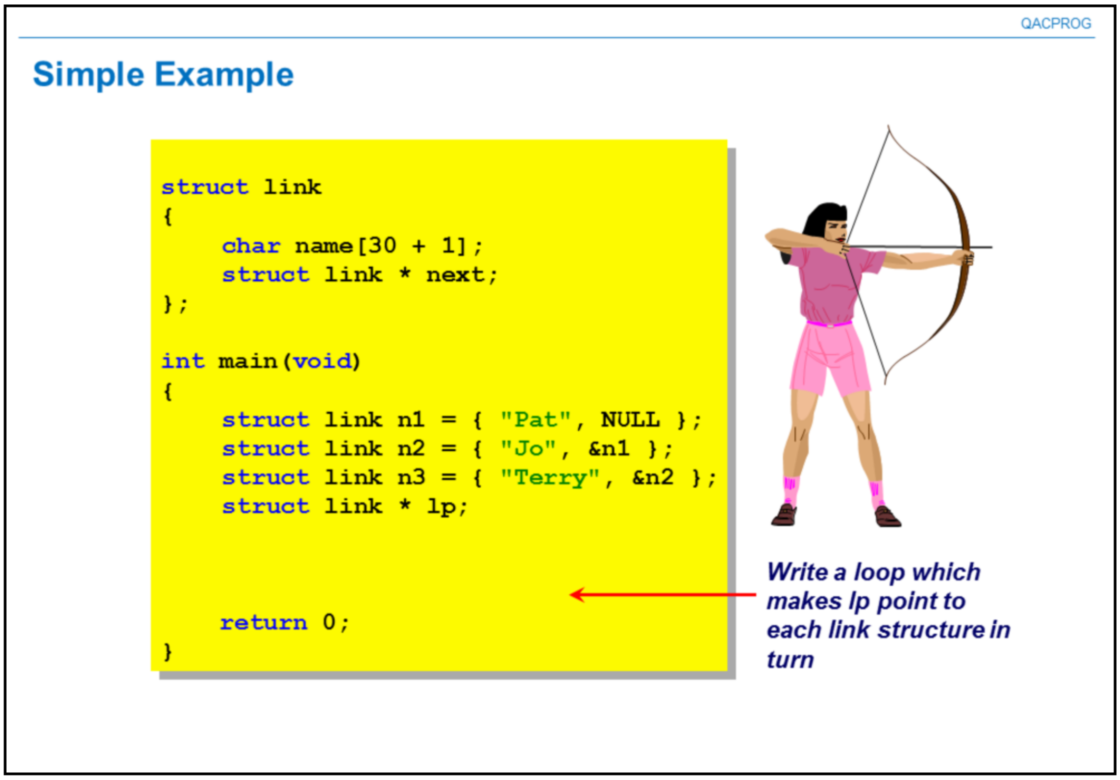
- The concepts of pointers to structures and structures containing pointers enable the creation of sophisticated dynamic data structures in C
- The data is usually dynamically allocated and de-allocated using `malloc()`, `free()`, etc...



These data structures require background knowledge. They are examples of dynamic structures that are widely used to represent program data. Often, the amount of program data is large. There are books that describe the basic concepts and common algorithms that create and maintain such structures.

For those with a knowledge of dynamic data structures, the following page is vital. There is a practical that handles a very elementary linked list.

For those new to this type of construct, these two pages are useful, but not essential reading at this stage. The practical should be tackled with care; perhaps viewing the solution would be more beneficial. Further coverage of this topic, together with the use of the library memory allocation and de-allocation functions are covered in a later chapter.



QACPROG

Simple Example

```
struct link
{
    char name[30 + 1];
    struct link * next;
};

int main(void)
{
    struct link n1 = { "Pat", NULL };
    struct link n2 = { "Jo", &n1 };
    struct link n3 = { "Terry", &n2 };
    struct link * lp;



    return 0;
}
```

Write a loop which makes lp point to each link structure in turn

The `link` structure is the smallest example that could illustrate the concept of a linked list. Normally, there are either more data members or the single data member is a pointer to another structure that contains the data. The code is not the usual way in which list items are declared. It is usual to make use of the library's dynamic memory allocation functions.

Solution:

```
for (lp = &n3; lp != NULL; lp = lp->next)
{
    printf("%s\n", lp->name);
}
```

For your interest only, a possible template for a directed graph (eight arcs maximum) containing data on personnel (using our `struct person` from the *Structures* chapter) is:

```
#include "person.h"
/* contains the struct person template amongst others */

struct person_graph
{
    struct person * data;
    struct person_graph *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7;
};
```

Intentionally left blank