Structures

Grouping

So far, we have been concentrating on primitive types and arrays

But there is a need to group data values together into 'meaningful' structures

For example, a bird in an angry birds game might have variables for its position, its velocity, and maybe other values such as a size or a colour, yet you want to treat it in a program as a single 'object'

Structs

The mechanism in C to group several values together into a single entity is the `struct`

This represents the first step in the process of developing 'object oriented programming', which will be explored in the Java unit

Declaring a struct

Suppose you have a 2-D graphics program, and in it a bird has a pair of x-y coordinates giving its position

Then you can declare a bird structure like this:

```
struct bird {
    int x, y;
};
```

Warning: don't forget the semicolon after the close curly bracket (it is like an initializer, not like an `if` or `while` block)

Using a struct

bird.c

```c
/* Struct demo */
#include <stdio.h>

// A bird 'object' looks like this
struct bird {
     int x, y;
};

// Move a bird a bit then print
int main() {
    struct bird b = {41, 37};
    b.x++;
    b.y = b.y + 5;
    printf("%d %d\n", b.x, b.y);
}
```

New type

The `struct` declaration creates a new type

```
struct bird {
    int x, y;
};
```

This is the type `struct bird` of bird variables

Each bird variable has two `int` fields (sub-variables) called `x` and `y`

Struct variables

New variables can then be created

```
struct bird b = {41, 37};
```

In this case, the variable b is initialized by specifying its fields in an initializer (don't forget the semicolon)

Accessing fields

Fields are accessed using the dot (`.`) notation

```
b.x++;
b.y = b.y + 5;
```

The field `b.x` is incremented by one, and `b.y` has `5` added to it.

Passing a struct


```c
/* Struct demo: doesn't work */
#include <stdio.h>

struct bird { int x, y; };

// Move a bird by a given amount
void move(struct bird b, int dx, int dy) {
    b.x = b.x + dx;
    b.y = b.y + dy;
}

// Move and print
int main() {
    struct bird jay = {41, 37};
    move(jay, 1, 5);
    printf("%d %d\n", jay.x, jay.y);
}
```

Pass by value

The failed experiment shows that structs are passed by value

In the example, the variable `jay` is copied into the argument variable `b`, so changes to `b` do not affect `jay`

This is ***different*** from arrays, presumably because structs are typically small

An abbreviation

The struct declaration has been abbreviated (to fit the example onto one slide)

```
struct bird { int x, y; };
```

Both semicolons are still needed

Returning a struct

struct.c

```c
/* Struct demo */
#include <stdio.h>

struct bird { int x, y; };

// Move a bird by a given amount
struct bird move(struct bird b, int dx, int dy) {
    b.x = b.x + dx;
    b.y = b.y + dy;
    return b;
}

int main() {
    struct bird jay = {41, 37};
    jay = move(jay, 1, 5);
    printf("%d %d\n", jay.x, jay.y);
}
```

Returning two things

In the example we looked at, returning a structure was an annoyance - it was just to allow us to write functions which update structures temporarily, before we get to pointers

But it is a good solution to a common problem: writing a function which finds two things

So if `position(...)` returns the position of something, a structure with `x`, `y` fields can be returned

Pass by reference

In practice, most programmers want to pass structs by reference, not by value, to avoid the cost of repeatedly copying the fields to and fro, and to allow functions to update the fields directly

That's done by passing pointers to structs instead of the structs themselves (see pointer chapter)

That will be a second step towards object oriented programming

Typedefs

Typedefs let you avoid using the `struct` keyword so often:

```
struct bird { int x, y; };
typedef struct bird Bird;
...
Bird move(Bird b) { ... }
...
int main() {
    Bird jay = {41, 37};
    ...
}
```

How typedefs work

A `typedef` doesn't define a type:

```
typedef struct bird Bird;
```

It defines a type synonym - another name for an existing type

The new name `Bird` comes at the end (as if you were declaring a variable), defined as a synonym for `struct bird`

Don't leave out the final semicolon

Using a name twice

Often the same name is used:

```
typedef struct bird bird;
```

Technically, there are now two identical names `bird`

But one only ever comes straight after `struct`, and the other never does, so the compiler can tell them apart

Using a name 3 times

It is possible to have a variable which has the same name as a typedef:

```
bird bird;
bird.x = ...;
```

The compiler can tell the two names apart because one only ever appears in 'type' positions, and the other only appears in 'variable' positions

But you can't do that for built-in type names:

```
int int = 42;
```

Arrays inside structures

To analyse a document, counting the number of times each word appears, you could use a structure like this:

```
struct word {
    char s[10];
    int count;
};
```

The array field `s` holds the word as a string

It has to be fixed length (a compile-time constant) so that the C compiler can do type and size analysis

We will see how to make it variable length in the memory chapter

Constant structures

Sometimes, you want to provide mostly readonly-access to a structure

A neat way to do it is this:

```
struct word { char s[10]; int count; };

typedef struct word const word;
```

From now on, the type `word` provides readonly access, whereas the type `struct word` provides full access