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 ball in a game might have several position and velocity coordinates, 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 we will go into in some depth in Panda 2

Declaring a struct

Suppose you have a 2-D graphics program, and you want to be able to pass around pairs of x-y coordinates as the positions of a ball

Then you can declare a ball structure like this:

```
struct ball {
    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


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

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

// Move a ball by one pixel horizontally and print
int main() {
     struct ball b = {41, 37};
     b.x++;
     printf("%d %d\n", b.x, b.y);
}
```

New type

The `struct` declaration creates a new type

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

This is the type `struct ball` of ball variables

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

Struct variables

New variables can then be created

```
struct ball 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 dot (`.`) notation

```
b.x++;
```

In this case, the field `b.x` is incremented by one

Passing a struct


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

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

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

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

Pass by value

The failed experiment shows that C treats a struct argument as an abbreviation for passing the fields individually

In other words, structs are passed by value

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

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 ball { int x, y; };
```

Both semicolons are still needed

Returning a struct

struct.c

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

// A ball has x, y coodinates
struct ball { int x, y; };

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

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

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

Creating a structure pointer

Here's how to create a pointer to a structure:

```
struct ball bdata = {41, 37};
struct ball *b = &bdata;
```

You can pass the pointer b to functions, but you can't pass it back as a result of the current function (because the structure will no longer exist)

The & operator means "create pointer to"

Pointer functions

Here's how to write a function to act on a structure pointer:

move.c

```
// Move a ball by a given amount
void move(struct ball *b, int dx, int dy) {
    b->x = b->x + dx;
    b->y = b->y + dy;
}
```

The notation `b->x` means "find the structure that `b` points to and access field `x`"

Typedefs

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

```
struct ball { int x, y; };
typedef struct ball Ball;
...
Ball move(Ball b) { ... }
...
int main() {
    Ball ball = {41, 37};
    ...
}
```

How typedefs work

A `typedef` doesn't define a type:

```
typedef struct ball Ball;
```

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

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

Don't leave out the final semicolon

Names

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

```
typedef struct ball ball;
...
ball ball;
ball.x = ...;
```

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

```
int int = 42;
```