Objects

Using Pointers

It is time to look at applications of pointers

This chapter is about using pointers with structs

In object-oriented C programming, an *object* is (usually) a ***pointer to a struct***

Object-oriented means lots and lots of different things, and this is just one of them, but it is a good start

Structs without pointers

So far, we have been passing structs directly to functions, and returning them as results:

```
struct state doSomething(struct state s, ...) {
    ... s.x = ...
}
```

There are three problems with this

The struct keyword

You can end up with `struct` scattered throughout a program, reducing compact readability, so do this:

```
struct state { ... };
typedef struct state state;

state doSomething(state s, ...) {
    ... s.x = ...
}
```

One-word type names seem more readable

(The only slight downside is that syntax colouring editors and tools may not colour the type name nicely)

The return problem

The first problem is that if the function `doSomething` updates the struct, then it is only updating its local copy (in its local argument variable `s`) so the updated state has to be returned

```
state original;
...
original = doSomething(original, ...);
```

It is incredibly easy to forget the `original =` bit which copies the updated state back into the original variable

The copying problem

The other problem is that the fields in the struct are all copied across into the function's argument variable, and then all copied back into the original (whether they have been updated or not)

This is inefficient

The inefficiency may matter for programs where structs are passed around a lot, or where some structs are very big (e.g. containing an array)

The solution

The answer is to pass structs around using pointers

Passing structs without pointers is ***very rare*** in real C programming, so stop it at once!

There are some changes to the functions that get called, and there are changes to the calling functions

Changing called functions

A called function is passed a pointer to a struct:

```
void doSomething(state *s, ...) {
    ... s->x = ...
}
```

The function no longer needs to return the struct, and it can return something else, if you want

It uses `s->x` to access fields, which is a shorthand, which all C programmers use, for `(*s).x`

Local allocation

In calling functions, one strategy is to continue to create structs as local variables

```
void someFunction(...) {
    state original_data = { ... };
    state *original = &original_data;
    ...
    doSomething(original);
}
```

Personally, I like to give the struct variable an obscure name, e.g. `..._data` (to make sure that I don't use it by accident) and to create a pointer variable with a nice name for general use

Naming

Don't do this, it is inferior:

```
state original = { ... };
...
doSomething(&original);
```

It is far too easy to forget the &, and it doesn't match the way that pointer variables are used in called functions

It is the *pointer* which is the object and which deserves the nice name

Returning

Never do this:

```
state *someFunction(...) {
    state original_data = { ... };
    state *original = &original_data;
    ...
    return original;
}
```

The memory for `original_data` disappears (to be reused by other function calls) when the function returns, so you are returning a dangling pointer

Choosing a scope

So, if you allocate a struct as an ordinary local variable in a function, then you have to choose a high-level function in which to allocate the memory, so that the lifetime of the object matches a single call to the function

If necessary, to create an object that lasts for the whole of a program run, you could allocate in `main` and pass it wherever it is needed

Dynamic allocation

There are very common circumstances where local allocation isn't flexible enough

One is where re-allocation is needed, e.g. for re-sizing an array, as with the `readline` that we saw before

Another is where a module needs to allocate an object and return it to a caller, without the caller needing to know about the details of the fields of the struct, or even its size

These are so common that some programmers use ***only*** dynamic allocation

Malloc

The code we had before becomes:

```
#include <stdlib.h>
...
void someFunction(...) {
    state *original = malloc(sizeof(state));
    *original = (state) { ... };
    ...
    doSomething(original);
}
```

To fill in a struct with a quick initialization, but not while the struct is being declared, you need to add a cast as shown (C99 feature)

Sketch

Let's look at some code from the sketch assignment:

```
struct state { int x, y; bool pen; };
typedef struct state state;

void doDX(state *s, int dx) { s->x = s->x + dx; }

void doPEN(state *s) { s->pen = ! s->pen; }

void interpret() {
    state *s = malloc(sizeof(state));
    *s = (state) { 0, 0, false };
    ...
    doDX(s, dx);
    ...
}
```

Shorthand

This line:

```
*s = (state) { 0, 0, false };
```

is just an abbreviation for this:

```
*s = (state) { .x=0, .y=0, .pen=false };
```

or this:

```
s->x = 0;
s->y = 0;
s->pen = false;
```

Same name

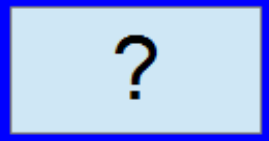
In the program, the state object is called `s` in every function

Thats OK, as long as you remember that the `s` in each function is a ***different*** local variable, which can't be seen from any other function

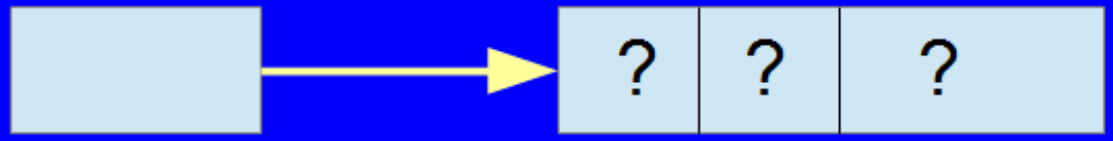
Visualisation

Visualising the memory during `interpret`:

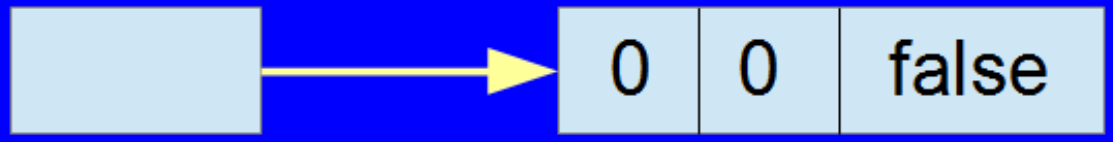
```
state *s;
```




```
s = malloc(...);
```




```
*s = (state) { 0, 0, false };
```



More visualisation

Visualising the memory during a call: