Objects

What are objects?

C doesn't properly support object oriented programming

But it is reasonable to use the word *object* to mean a structure or array, accessed using a pointer

This represents another step towards object oriented programming

This chapter looks at what changes when structures and arrays are handled using pointers

Strings

Compare these program fragments:

```
char s[] = "bat";
```

```
char *s = "bat";
```

These are nearly identical: extracting a character `s[i]` is the same, printing is the same, comparing with other strings is the same, ...

Differences

```
char s[] = "bat";
```

```
char *s = "bat";
```

- the first version is less efficient for a long string (it allocates more memory, and involves copying)
- in the second version, `s[0] = 'c'` is illegal (because `s` points to a constant string, probably in read-only memory as part of the program)

So use the first version when the string might get updated, otherwise it is a minor efficiency issue

Puzzle

What does this program do?

```
#include <stdio.h>
int main() {
    char *s1 = "cat";
    char *s2 = "cat";
    if (s1 == s2) printf("same\n");
    else printf("different\n");
}
```

Puzzle

What does this program do?

```
#include <stdio.h>
int main() {
    char *s1 = "cat";
    char *s2 = "cat";
    if (s1 == s2) printf("same\n");
    else printf("different\n");
}
```

Answer: *it depends* whether the compiler optimises by noticing that it can reuse the same constant string

The important thing is that `s1 == s2` is pointer comparison

Passing arrays

Compare these functions:

```
void print(char s[]) { ... }
```

```
void print(char *s) { ... }
```

These are identical, because in the first version, the array is passed by reference, i.e. by pointer

In other words, the compiler converts the first version into the second

Returning arrays

Pointer notation allows us to return an array from a function, which we couldn't do before:

```
char *show(int n) {  // BAD
    char s[12];
    itoa(n, s, 10);
    return s;
}
```

This is *illegal*, with undefined behaviour, because the array `s` disappears when the `show` function returns

You are returning a dangling pointer

Returning arrays 2

This is OK, though:

```
char *min(char *s1, char *s2) {
    if (strcmp(s1, s2) < 0) return s1;
    else return s2;
}
```

The fact that newly allocated memory can't be returned from a function is a serious restriction

A complex program might have to allocate lots of memory in advance, without knowing how much is going to be needed

We will sort this out when we get to `malloc`

Pointing into arrays

Pointers allow us to do this:

```
int *p = &a[i];
```

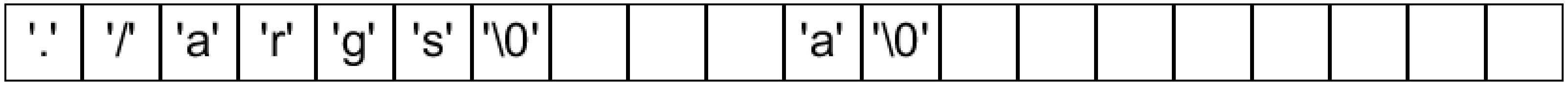
This allows us to handle subarrays, loop through arrays using pointers (not necessarily recommended) and so on

`main`

When `main` has arguments, you might expect:

```
... main(int n, char args[n][]) ...
```

But then the strings must have the same length (say 10) so if the program is called `args`, and you type `./args a`, you get:

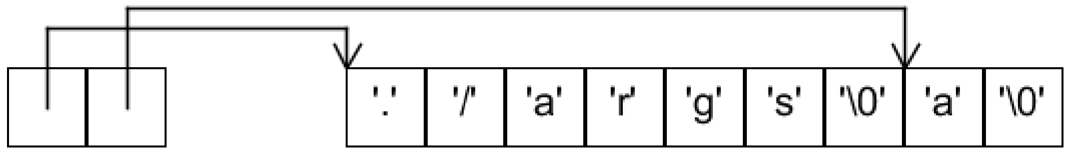


Array of pointers

Instead, `main` is:

```
... main(int n, char *args[n]) ...
```

So `args` is an array of pointers to strings, which can have different lengths but can be packed close together:



Other notations

You might also see:

```
... main(int n, char **args) ...
... main(int argc, char *argv[argc]) ...
... main(int argc, char *argv[]) ...
```

The first unnecessarily uses pointer notation twice

The second uses traditional (poor?) names ('argument count' and 'argument vector')

The third leaves out the array size

Structs without pointers

Reminder: when you pass a struct directly to a function, it is copied, so any updates have to be returned and put back in the original structure:

```
struct bird move(struct bird b, ...) {
    ... b.x = ...
}
...
b = move(b, ...);
```

This is very fussy, so let's tidy it up

The struct keyword

Let's use `typedef` like before to get rid of the `struct` keyword:

```
typedef struct bird bird;

bird move(bird b, ...) {
    ... b.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

If a function `move` updates the struct, then it is only updating its local copy (in its local argument variable `b`) so the updated struct has to be returned

```
bird jay;
...
jay = move(jay, ...);
```

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

The copying problem

The fields in the struct are all copied across into the function's argument variable, and then 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 some changes to the calling functions

Changing called functions

A called function is passed a pointer to a struct:

```
void move(bird *b, ...) {
    ... b->x = ...
}
```

The function no longer needs to return the struct

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

Local allocation

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

```
int main(...) {
    bird jaydata = { ... };
    bird *jay = &jaydata;
    ...
    move(jay);
}
```

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

In my opinion, this is inferior:

```
bird jay = { ... };
...
move(&jay);
```

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

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

Returning

Never do this:

```c
bird *newBird(...) {  // BAD
    bird bdata = { ... };
    bird *b = &bdata;
    ...
    return b;
}
```

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

Example

bird.c

```
/* Passing structures using pointers */
#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;
}

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