C Programming
Function Pointers and Object-Oriented C

Adam Sampson (based on material by Greg Michaelson)
School of Mathematical and Computer Sciences
Heriot-Watt University

`funcp.c`

Function pointers

Function addresses

- Functions are sequences of instructions, stored in memory...
- ... so functions have memory addresses
- The function name is an alias for the address

```
int sq(int x)
{
    return x * x;
}

int main(int argc, char *argv[])
{
    printf("sq: %ld, &sq: %ld\n",
           (long) sq, (long) &sq);
    return 0;
}

→ sq: 134513572, &sq: 134513572
```

Function pointers

rettype (* *name*)(*argtypes*);

- Declares *name* as a pointer to a function that takes *argtypes* and returns *rettype*

```
int main(int argc, char *argv[])
{
    int (*f)(int);
    f = sq;
    printf("%d\n", f(3));
    return 0;
} → 9
```

Higher-order functions

- A HOF is a function that takes another function as an argument – for example...
- `map` applies a function to every element of an array

```
void map(int (*f)(int), int a[], int n)
{
    for (int i = 0; i < n; i++)
        a[i] = f(a[i]);
}
...
    map(sq, my_array, 10);
```

extra

`map.c`

Higher-order function: map

Object-oriented C

- OO techniques – used carefully – are a good way of structuring large, complex programs
- But C doesn't have OO facilities – classes, inheritance...
  - **C++** does – but C++ is a much more complex language, with greater reliance on standard library support, so it's not as widely used for embedded or kernel development
- Many C programs and libraries **implement** OO using the facilities that C provides
  - e.g. Linux kernel drivers are objects providing an interface

A class

```
class Counter {
    public Counter() {
        count = 0;
    }
    public void tick() {
        count++;
    }
    private int count;
};
```

Java/C++-ish pseudocode,
not a real language...

Translation

```
class Counter {
    public Counter() {
        count = 0;
    }
    public void tick() {
        count++;
    }
    private int count;
};
```

```
typedef struct {
    int count;
} counter;
```

Translation

```
class Counter {
    public Counter() {
        count = 0;
    }
    public void tick() {
        count++;
    }
    private int count;
};
```

```
typedef struct {
    int count;
} counter;

void counter_init(counter *this) {
    this->count = 0;
}
```

We could also write a destructor function

Translation

```
class Counter {
    public Counter() {
        count = 0;
    }
    public void tick() {
        count++;
    }
    private int count;
};
```

```
typedef struct {
    int count;
} counter;

void counter_init(counter *this) {
    this->count = 0;
}

void counter_tick(counter *this) {
    this->count++;
}
```

Translation in use

```
Counter ctr;                        counter ctr;
                                    counter_init(&ctr);

ctr.tick();                         counter_tick(&ctr);
```

Private data is put into a `struct`

Each method becomes a function, taking the `struct` as the first argument

Inheritance

```
class FancyCounter : Counter {
    public void noisyTick() {
        print(msg);
        tick();
    }
    private String msg;
};
```

Inheritance by composition

```
class FancyCounter : Counter {
    public void noisyTick() {
        print(msg);
        tick();
    }
    private String msg;
};
```

```
typedef struct {
    counter super;
    const char *msg;
} fancy_counter;
```

If we inherit from another class...

... include that class's data as a field

Inheritance by composition

```
class FancyCounter : Counter {
    public void noisyTick() {
        print(msg);
        tick();
    }
    private String msg;
};
```

```
typedef struct {
    counter super;
    const char *msg;
} fancy_counter;

void fancy_counter_noisy_tick
        (fancy_counter *this) {
    puts(msg);
    counter_tick(&this->super);
}
```

We can refer to the superclass data when calling its methods

Inheritance by composition, in use

```
FancyCounter fc;

fc.noisyTick();
```

```
fancy_counter fc;
counter_init(&fc->super);

fancy_counter_noisy_tick(&fc);
```

This is a bit ugly from the caller's point of view, though...

Abstract class

```
class Tickable {
   public abstract void tick();
   int count;
};

class LoudTicker : Tickable {
   public void tick() {
      print(“TICK ” + count);
      count++;
   }
};
```

This is an **abstract class** (or **interface**): each subclass of `Tickable` must implement `tick`

Abstract class

```
typedef struct _tickable {
    void (*tick)(struct _tickable *this);
    int count;
} tickable;



tickable t;
loud_ticker_init(&t);

t.tick(&t);
```

We translate the abstract method into a **function pointer**

Abstract class

```
typedef struct _tickable {
    void (*tick)(struct _tickable *this);
    int count;
} tickable;
```

We translate the abstract method into a **function pointer**

```
tickable t;
loud_ticker_init(&t);

t.tick(&t);
```

... which we can use when calling the method (yes, we write `t` twice!)

Abstract class

```
typedef struct _tickable {
    void (*tick)(struct _tickable *this);
    int count;
} tickable;

void loud_ticker_tick(tickable *this) {
    printf(“TICK %d”, this->count);
    this->count++;
}
```

Method definition, matching the function pointer type

Abstract class

```
typedef struct {
    void (*tick)(void *this);
    int count;
} tickable;

void loud_ticker_tick(tickable *this) {
    printf(“TICK %d”, this->count);
    this->count++;
}

void loud_ticker_init(tickable *this) {
    this->tick = loud_ticker_tick;
    this->count = 0;
}
```

In the constructor, set the function pointer(s)

Basics of OO in C

- So we can now have an interface (`tickable`) which is implemented by multiple different classes
  - This is used widely in the Linux kernel – e.g. to have drivers for different “char” devices provide a common set of read/write functions
- What if the subclasses need to add data members?
  - Have a `void *data` pointer in the structure, which subclasses can use to point to a private data structure...
  - (... or make all the methods take `void *this`, casting to the right type internally, and use composition to put the function pointers at the start of all classes' structures – this is subtle! See GObject.)

Basics of OO in C

- In practice, we often separate the function pointers for an object from its data members
  - The methods are the same for all objects of a class, and a complex interface may have many of them
- So you will see **two** `structs`:
  - `foo_ops` containing the function pointers
  - `foo` containing the data members, plus a pointer to the appropriate `foo_ops` structure
- This is how C++'s OO features are compiled