B

Objective-C in five minutes

This appendix considers a very brief introduction to Objective-C (Obj-C) which is an extension of the C language that builds-in some object orientated programming facilities. If you are familiar with C++ you should have little difficulty adapting to Obj-C. There are few subtle changes from C++ and some different terminology.

Obj-C introduces the concept of a class that extends the C `struct` in a way that is similar to the way in which C++ extends the C `struct` into the C++ `class`. An Obj-C class allows member variable (*fields*) to be declared as public or private, and object specific functions (*methods*) to be defined. Like C++, an object is an instance of a class.

- Obj-C defines its classes through its `interface` keyword;

- Just as a C++ class contains *members* an Obj-C interface contains *fields*. A field can be any C data type or a **pointer** to an object of an Obj-c class. Only pointers to Obj-C classes can be inlcuded as fields in a class, instances of Obj-C classes are never included.

- Just as a C++ class contains *methods* an Obj-C interface contains *methods*

- An Obj-C class can contain two types of method:

    1. Instance methods behave similarly to C++ class methods. An instance method is identified by a ”-” at the start of its definition.

    2. Class methods behave similarly to C++ static methods. A class method is identified by a ”+” at the start of its definition.

- An interface is declared with an `interface` statement. This is typically contained in a header file. The header file is included using an `#import "filename"` directive when needed

```
@class my_classX; // forward class definition

@interface my_interface {
 // declare fields
 @public   // same as C++
  char field3[5];
 @protected //  same as C++
  int field1;
  float field2;
@private
 my_classX *pX; // reference to a class
}
//declare method prototypes
 -(void)method1;
@end
```

- The `interface` (class) is usually implemented in a separate file and is called an `implementation`. The implementation defines the methods of the class :

```
#import "my_interface.h"

@implementation my_interface

-(void) method1 { // no arguments
 return;
}
// other methods
@end
```

- A class may be derived from a parent class:

```
@interface my_interface : derived_from_this_class  {
 // fields and methods declaration
}
```

- A class may implement a `protocol`. A protocol is a collection of methods that a class can use to present a common appearance to is user. The class should implement the protocol's methods in a way that is appropriate for that class. A protocol can be used to archieve the same sort fo things that is done using *pure virtual functions* in C++ and with Java's *interface* facility.

  A class that implements a protocol includes the protocol name in the class declaration. For example if a protocol is called `my_protocol` then, if class `my_class`, which is derived from `my_parent`, implements the protocol it would be specified as:

```
#import "my_protocol.h"
@interface my_class : my_parent <my_protocol> {
..
}
..
@end
```

Classes can implement several protocols at the same time.

```
@interface my_class : my_parent <my_protocol1, my_protocol2>
```

Protocols are often used in IOS to implement the delegate system, for example the `UIApplicationDelegate`

- An interface's instance method is invoked using square brackets, with a pointer to the object and the name of the method and arguments.

  ```
  [object_pointer method_name];
  ```

  In Obj-C one speaks of sending a *message* to an object to perform a method, which is exactly the same as calling a function in C. If the method has any arguments they are passed after a set of ":"s. An interface's class method is invoked using the name of the class:

  ```
  [class_name method_name];
  ```

  There is a slight difference between the first argument and any other arguments(the first argument is attached to the method name by a ":"). Arguments are separated by spaces.

- A method has a single return type and none or more arguments. For example:

  ```
  @interface testClass {
    int field1;
  }
  - (void) Method1;    // instance method,  no arguments
  +(id)Method2;      // class method, no arguments
  -(int)Method3:(int)arg1;  // instance method, one argument
  // instance method, 3 arguments
  -(int)Method4:(int)arg1  a2:(int)arg2  : a3:(float)arg3;
  @end
  ```

In the case of the last method above, the text to the right of the argument is regarded as part of the method name, so this method is called `Method4 a2 a3`. For example to pass messages(call) to methods: Method1, Method3 and Method4 of an object of class `testClass` and *pointed to* by `pX` use:

```
// call Method1 of an objectthat is  pointed to by pX
[pX Method1];
// call Method3 (one argument)
[pX Method3:arg1_value];
// call Method 4 (3 arguments)
[pX Method4:arg1_value a2:arg2_value a3:arg3_value];
// to send a message to a  class method, use:
[testClass Method2];
```

- Refer to a *field* in a class name just be referring to the field by name.
- Special variables `self` and `super` Refer the the object itself and its direct parent. (Similar to the `this` pointer) Can be used to send messages (call) methods in a parent class.

In our examples we are only going to be concerned with Obj-C classes derived from other OSX or IOS classes, in this case most classes are derived (possibly in a chain) from `NSObject` . In neither of these situations most Obj-C classes are derived from `Object`.

The N͡SObject and `Object` class provide class methods to allocate and deallocate memory for the object. For example to created an object of class `MyClass` which is derived from one of the base classes use:

```
 MyClass *pc; // pointer to object
 // allocate the memory for the object (class method)
 pc=[MyClass alloc];
 pc=[pc init];  // initialize the fields (members)
..
 // this could all be done in one line
MyClass *pc=[[MyClass alloc] init];
 ..
 // release the object by calling its parent deallocator
 // for NSObject
 [super dealloc];
 // or
 [pc free];  // for "Object" parent class
```

To access *fields* (members) of an object, accessor methods are written. In OSX and IOS accessor methods can be generated automatically, with the use of `@property ...  ;` and `@synthesize ...  ;`