Literate Object Oriented Library.

Brandon Koepke

February 12, 2016

Contents

Abstract

The following is a simple object library for C. An object has a class name, a to_string method, and a free method. We also included default implementations of to_string and free that will be suitable for most implementations.

1 Class Definition

We start by defining a basic class structure. We are forward declaring our *Class* to ensure that the implementation details can be changed at a later date.

1a ⟨*class-typedef-h* 1a⟩≡ (2)

```
typedef struct _Class Class;
```

The class vtable contains a single operation, the ability to get the class name. The vtable should also be considered opaque to other objects, manually manipulating this from outside a class (including invoking it) is considered undefined behavior. In addition, by defining the name as a function as opposed to including a string reference enables us to change the implementation in child classes.

1b ⟨*class-vtable-h* 1b⟩≡ (2)

```
typedef struct {
  const char * const (*name)(const Class * const);
} class_vtable;
```

To get the class name safely, use the following method.

1c ⟨*class-name-h* 1c⟩≡ (2)

```
const char * const class_name(const Class * const c);
```

The class implementation only contains the class vtable, everything else is deferred to implementers.

1d ⟨*class-typedef-c* 1d⟩≡ (1e)

```
struct _Class {
  class_vtable *vtable;
};
```

In the *class_name* implementation we invoke the name function from the vtable against the class instance passed.

1e ⟨*class.c* 1e⟩≡

```
#include "class.h"
⟨class-typedef-c 1d⟩
const char * const class_name(const Class * const c) { return c->vtable->name(c); }
```

Finally we have the completed header.

2 ⟨*class.h* 2⟩≡

```
#pragma once
⟨class-typedef-h 1a⟩
⟨class-vtable-h 1b⟩
⟨class-name-h 1c⟩
```

2 Object Definition

An object inherits all the abilities of a class. In addition, an object can also have it's memory freed and can be converted to a string representation.

3a ⟨*object-h* 3a⟩≡ (3g)

```
typedef struct _Object Object;
typedef struct {
  class_vtable class;
  const char * const (*to_string)(const Object * const);
  void (*free)(Object *);
} object_vtable;
void object_free(Object *o);
const char * const object_to_string(const Object * const o);
```

3b ⟨*object-typedef-c* 3b⟩≡ (3h)

```
struct _Object {
  object_vtable *vtable;
};
```

We also have default implementations of *free* and *to_string*.

3c ⟨*object-default-h* 3c⟩≡ (3g)

```
void _object_free(Object *o);
const char * const _object_to_string(const Object * const o);
```

We define the interface method for *object_free* where we invoke the vtable implementation and also the default implementation which runs the stdlib free.

3d ⟨*object-free-c* 3d⟩≡ (3h)

```
void object_free(Object *o) { o->vtable->free(o); }
void _object_free(Object *o) { free(o); }
```

Next we define the *to_string* method for objects. Make sure you free the string once it goes out of scope.

3e ⟨*object-to-string-c* 3e⟩≡ (3h) 3f▷

```
const char * const object_to_string(const Object * const o) {
  return o->vtable->to_string(o);
}
```

The default *to_string* implementation is the same as the default Java implementation. We get the name of the class and the hex representation of the pointer. From here we print '*class_name*@pointer'.

3f ⟨*object-to-string-c* 3e⟩+≡ (3h) ◁3e

```
const char * const _object_to_string(const Object * const o) {
  const char * const name = class_name((Class *)o);
  size_t size = sizeof(void *) + sizeof('@') + strlen(name) + sizeof('\n');
  char *buffer = malloc(size);
  snprintf(buffer, size, "%s@%x", name, (unsigned int)o);
  return buffer;
}
```

The final header and object implementations.

3g ⟨*object.h* 3g⟩≡

```
#pragma once
#include "class.h"
⟨object-h 3a⟩
⟨object-default-h 3c⟩
```

3h ⟨*object.c* 3h⟩≡

```
#include "object.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
⟨object-typedef-c 3b⟩
⟨object-free-c 3d⟩
⟨object-to-string-c 3e⟩
```