C!

Implementing Interfaces

Marwan Burelle

marwan.burelle@lse.epita.fr
http://www.lse.epita.fr/

July 20, 2012

Outline

Introduction

Quick Overview of **C!**

- System oriented programming language
- Variant of C with a rationnalized syntax.
- Syntactic Sugar.
- Object Oriented extensions.
- **C!** is a *compiler-to-compiler* producing standard C code

Motivation

- **C!** was designed for kernel programming
- The aim of the project is to provide modern extensions to C (like *OOP*) without breaking the aspects that make C a good kernel programming language.
- **C!** has no run-time depencies: make it easier to integrate in a non-standard environment.
- We try to minimize the amount of *hidden code*: almost all expensive operations are explicit (no hidden object copy for example.)
- Since we produce C code, it is quite easy to integrate **C!** pieces of code in existing C projects.

Objects in C!

Object Model

- The *OOP* model provides:
  - Class definitions with virtual methods (true methods)
  - Simple inheritance and subtyping based polymorphism
  - Interfaces and abstract methods
  - Method overriding
  - No overloading
  - No visibility control
- In addition, objects instances follow some rules:
  - Objects are always pointer
  - You can define local object on the stack
  - You have to provide your own allocation scheme

Interfaces and Abstract Methods

- You can define interfaces: pure abstract classes with no state and no implementation for methods.
- Interfaces can inherit other interfaces and unlike classes inheritance, you can have multiple inheritance.
- Class can explicitely *support* interfaces (no structural typing)
- When implementing an interface you (as usual) must provides an implementation for each methods in the interface.
- You can have abstract methods in class: methods with no implementation
- Classes with abstract methods can't be instantiated

The Inner Side

Objects

Objects in C

- While C does not provides objects, we can use pure C constructions to represent them.
- We only need structure and function pointers.
- Thus, **C!** *OOP* extensions can be encoded as (almost) pure syntactic sugar.
- Since we produce C code, we take advantage of automatic offset computation in structure.
- Using structures rather than annonymous table generate more readable code.

Basic Objects

- An object is the association of some data (attributes) and operations on this data (methods)
- We just put function pointers in a structure.

Example:

```
class obj {
 get() : int;
 set(int x) : void;
 int content;
}
```

```
struct s_obj {
 int (*get)(struct s_obj*);
 void (*set)(struct s_obj*,int);
 int content;
};
```

- To call methods, you just have to find the pointer in the object
- The only things that you need is to provide the `this` pointer to the method.
- For cascade of methods call you will need temporary variables to cache results of method's calls to avoid multiple calls to the same method.

Example:

```
o.set(42);                    o->set(o,42);
```

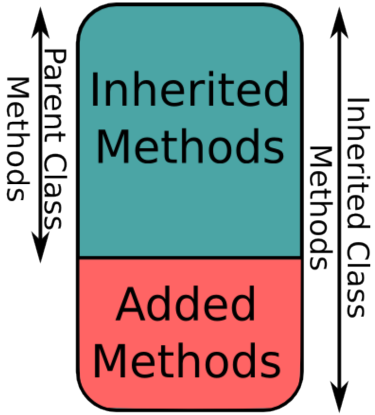
Classes and inheritance

- Defining classes is straightforward: a classes is a structure together with the initialization code to fill the structure.
- To avoid unnecessary duplication of pointers to methods, we build an external structure with only methods that will be shared by all instances.
- Inheritance is also straightforward: structures (state and methods) are ordered so that methods described in parent class come first.

Example:

```c
struct s_obj_vtbl {
  int (*get)(struct s_obj*, void);
  void (*set)(struct s_obj*, int);
};
struct s_obj {
  struct s_obj_vtbl *vtbl;
  int content;
};
```

Classes And Inheritance



Layout For Table Of Methods

Interfaces

Issues

- Unlike class inheritance, you can implement several interfaces.
- You can inherit from a class and implement interface.
- $\Rightarrow$ **We can't rely on structures layout.**
- *What if we use more than one table of methods ?*

Insufficient information

- Interfaces are used to pass objects to function without knowing their real type.
- So, we won't be able to access specific object content that is not described in the interface.
- So, we need to found a way to access some specific fields of an object without knowing the object layout.

Dynamic Dispatcher

- A possible solution is to have a function that select the right table of methods.
- This function is just a a big switch using a hash of the interface name to select the right table.
- Now an object will look like:

Example:

```
struct s_obj {
  void *(interface_table)(int);
  struct s_obj_vtbl *vtbl;
  // object content
};
```

Avoiding Dispatch Overhead

- The previous method induces an overhead at each method call (a function call)
- The idea to avoid this overhead by *transforming* the object when we still have the needed information.
- Once the object is transformed, we don't need the dynamic dispatcher that select the correct method's table.
- Let's take a look at the context first:

Example:

```
interface A {
  get() : cint;
}
class B support A {
  get() : cint { return 42; }
}
f1(o : A) : cint { return o.get(); }
f2(o : B) : cint { return f1(o); }
```

Transforming Objects

- While there's various ways to implement objects transformation, the way we inject information is usually always the same.
- The transformation must be injected where we still know the real type of the object.

Example:

The previous function:

```
f2(o : B) : cint { return f1(o); }
```

Into the following:

```
f2(o : B) : cint { return f1( CAST(o,A) ); }
```

How to transform an object

- There's various way to transform an object:
  - You can change the pointer to the table of methods
  - You can build an helper object around your object
  - You can have all interfaces in your object and replace the pointer of the real object
- First two methods has a lot of drawbacks (overhead, inconsistency in concurrent context ...)
- The last method need some refinement

Hosting All Interfaces

- The idea is to have inside the object a block with, for each supported interface, a mini-object with a pointer to corresponding table of methods.

Example:

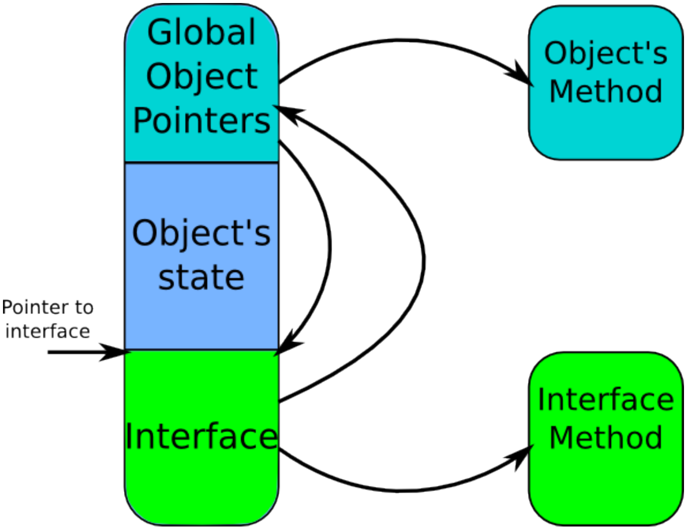
```
struct s_A_vtbl { int (*get)(void); };
struct s_obj_interface_table {
  struct { s_A_vtbl *vtbl; } A;
};
struct s_obj {
  struct s_A_vtbl *interfaces;
  struct s_obj_vtbl *vtbl;
  // object state
  struct s_A_vtbl _interfaces;
};
#define CAST(_OBJ, _INTER_) \
  (&((_OBJ)->interfaces->_INTER_))
```

- We need to pass a `this` pointer to our methods.
- But the pointer we have in the calling context does not correspond to the real object.
- So, we provide a pointer to the real object inside the structure associated to the interface.

Example:

```
struct s_obj_interface_table {
  struct {
    s_A_vtbl *vtbl;
    void     *real_this;
  } A;
};
// Calling a method from interface context
o->vtbl->myMethod(o->real_this);
```

Full Object Layout



Transitive Inheritance

- When you support an interface that is itself derived from various interfaces, you can be transformed to any of these interfaces.
- So we need to provide access to these interfaces, but we can be transformed in context where we don't know how:

Example:

```
interface A { /* Some defs */ }
interface B : A { /* More defs */ }
class C support B { /* implem */ }

f1(o : A) { /* some code */ }
f2(o : B) { f1(o); }
f3(o : C) { f2(o); }
```

Solutions ?

- Provide a function (similar to the dynamic dispatcher) that provide a pointer to the good interfaces (*dynamic cast*.)
- Embed recursively interfaces description (probably a lot of space overhead.)
- We also need conversion for comparison operators.

Of Structures And Classes

Why Pointers To Objects ?

- C++ offers to manipulate object as pointer or directly.
- I choose to **not** follow this approach.
- There's almost no reason to transmit objects by copy.
- When you need copy, you should obviously do it **explicitly**.
- Copy also introduce new issues: when using an object throught one of its parent types or throught an interface, you can't simply rely on C to correctly copy the struct since it doesn't know the size.

Why Do We Have Classes And Structs in **C!**

- You may use structures for other reason than structured data
- A good example is when you want a bunch of contiguous heterogeneous data
- For a lot of cases you don't want meta-data in your structure.
- There's also cases where you want data to be copied to called functions without overhead.
- So, classes and structures serve different purposes and we need both.
- If you want *member functions* (or non-virtual methods) acting on structures you can use *macro-class* in **C!**

More on C!

- Git repository:
  http://git.lse.epita.fr/?p=cbang.git
- Redmine Project:
  http://redmine.lse.epita.fr/projects/cbang
- Global Overview Blog's article:
  http://blog.lse.epita.fr/articles/
  12-c---system-oriented-programming.html
- **C!** syntax overview:
  http://blog.lse.epita.fr/articles/
  23-c---system-oriented-programming---syntax-explanati.
  html

QUESTIONS ?

Figs/questionMark