C!

A System Oriented Programming Language

Marwan Burelle

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

Outline

Introduction

Kernel Programming?

What do we use in kernel programming?

- Low-level code:
  - Direct memory access (reading/writing to in-memory registers and other adress-based data access);
  - Architecture-specific code (ASM inlining);
  - Low-level execution flow manipulation.
- More classical code:
  - Data structures;
  - Algorithms.
- Control over binary building.

Languages for Kernel Programming?

Programming in kernel space requires some features in the language and its toolchain.

- Address manipulation (pointers. . .);
- Function pointers;
- Transparent memory mapping of data structures;
- Interface with lower-level code (ASM inlining. . .);
- Bitwise operations;
- Control over the linking and how the output binary is produced;
- Very small or inexistant runtime dependencies.

Why Only C?

- Why not a *High Level Language*:
  - Most of the modern languages are oriented towards userland applications;
  - *High level languages* (such as Java) often rely on specific runtime or worse (Java programs runs on a Virtual Machine);
  - *High level languages* often forbid low level manipulations;
  - Even languages such as C++ or Google's Go requires a specific runtime (Go uses a garbage collector, has specific calling conventions and various userland only native features) incompatible with kernel programming, or at least difficult to adapt.
- And C?
  - C was meant to code kernel!
  - Produced code is almost a direct translation of the original source code;
  - Most of the current C compilers offer ASM inlining.

Why would we need another language?

- The official C (ISO) is complex and full of *"things that used to be that way and won't change"*;
- The language has some odd or ambiguous syntax;
- The type system of the language is too permissive on some subjects while being too restrictive on others;
- The language lacks modern programming idioms (like objects) that can be useful;
- Some features of the language are outdated and do not fit well in today's context (*what is the size of an* `int`*? what really does the keyword* `register`*? ...*).

C!

- C! (pronounced *c-bang*) is our attempt to have a new programming language for kernel and low-level programming;
- Our main goals were:
  - Rationalization of the C language syntax;
  - Minor but useful extensions (mainly syntactic sugar);
  - Attempting for a better type system (with possibly some extensions such as *Generics*);
  - Simple but usable objects;
  - Language idioms dedicated to kernel/low-level; programming
  - Introducing a module formalism (redundant header files, namespaces, . . .);
  - Keeping what makes C a good language for kernel.
- Currently C! is more a proof of concept rather than a production language, but the compiler prototype can already be used;
- Most of our goals are (at least partially) achieved.

Global Language Overview

Compiler to Compiler

- We chose to use a Compiler to Compiler model;
- The C! compiler produces C code that can be compiled using an almost standard C compiler (we use C99 and GNU C features);
- Why Compiler to Compiler?
  - Hard work on native code producing is obtaind for *free*;
  - Most C! specific features are syntaxic sugar over pure C;
  - Advanced features like object can directly be expressed in C;
  - The C programming language can be seen as high level portable ASM;
  - It enforces our idea of a new native general purpose system programming language: a strong low level base and non-intrusive higher level extensions.

Syntax Rational

- The basic idea of C! is to globaly preserve C syntax with rational modifications;
- The first important point was to design the general syntax in a way that can be expressed using a standard parser generator (like yacc);
- Modifications to the original syntax should be globaly coherent;
- The main difference is unambigous *typenames*: type identifiers must be identified syntactically: we do not want lexing trick!
- Modifications on the syntaxic localization of type identifiers completely modifies variable and function declarations (and thus structures' fields are also modified);
- Other syntax differencies are syntaxic sugar and pure C! extensions (objects, macro-class, ...);

Integer Rational

- In C integer types are loosely defined, the `int` type obeys two contradictory definitions:
  - `int` type should be of the size of the machine word;
  - `int` type should be 2 or 4 bytes (16 or 32 bits) long.
- Some others types have a fixed size (`char`, `short`, ...) and other are machine dependant (`size_t` ...);
- C99 introduces sized types but these definitions are *fixed* (*i.e.* the standard defines a fixed set of types not a way to define integer type by its size);
- We directly introduce size (and signedness) in type declaration: a 32 bits signed integer will decalared as `int<32>` and the unsigned version as `int<+32>`;
- For homogeneity we use a similar scheme for floating point numbers;
- Integer with unusal size (less than 64 bits) can also be expressed, they will be stored in usual integer and we introduce management code to handle overflow.

Minor Extensions

- Structure definitions are type defitions (no need to use the *struct* keywords);
- Unsigned integers can be used as bit arrays;
- Bitfields are natural extension of sized integer definitions;
- Packing of structures is a language feature;
- Smarter attempt to handle enumeration's identifier conflict (a local variable can't hide an enum value);
- Several *work in progress* for eliminating unused (or badly named) variable qualifiers;

Objects

- C! has a simple object oriented syntax extension;
- The object extensions is provided for convenience, it is not a major paradigm in C!;
- One should use objects to simplify data structuring, not code structuring;
- This extension is build as a syntaxic sugar over *hand-made* objects in C;
- We have simple inheritance, only virtual methods and method overriding (but not overloading);
- For unambigous manipulations, objects are always pointers (no implicit copy mechanism);
- Construction is classical but memory allocation is up to the programmer (the constructor take a pointer as first parameter);
- Object obey to a single syntax (only `obj.method()` and no horrible `obj->method()`);

Modules

We introduce a simple *modules model* to avoid naming ambiguity in multiple files compilation:

- Symbols and type names are isolated between modules using (simplified) namespaces;
- Symbols and type names are exported specifiying a namespace:
  - `global_variable : int<32>;` won't be exported;
  - `my_module::global_variable : int<32>;` will be exported.
- A module can use another module's exported resources the `import` statement: `import my_module;;`
- The backend to C uses mangling to avoid name conflicts between modules during the linking;
- Unlike in C, a module M2 cannot use a module M3's resources without importing it even if both are used by a module M1 (this error is detected with splitted compilation).

Other Extensions

- C! offers a concept of *macro class*: object oriented like syntax for operations call on non-object types;
- Various attempts for *external code* inlining (asm or pure C) are *(still work in progress)*;
- Genrated C code is (or we hope it is) supposed to be human-readable and directly usable by a C programmer (simpler integration in existing code base);
- Some newer syntax (still in design phase) will try to solve classical low level issue such as automation of register like (adress based) data structures.
- We also try to have a better type system (there's still some issues);

Syntax and Sugar

Syntax Rational: type position

- The main C's grammar issue is about type position:

```
struct my_struct[5] *my_var;   →   my_var : my_struct[5] *;
```

- The former style forces to hack the lexer in order to memorize typedefs. The last one introduces no ambiguity.
- This syntax is used everytime one associates a type with a name:
  - Parameters and return type in function declarations:

```
double my_function(long prm1, float prm2);
↓
my_function(prm1 : int<32>, prm2: float<32>) : float<64>;
```

  - Field declaration for boxed types (structures, classes, . . .):

```
struct my_struct { long field; ... };
↓
struct my_struct { field : int<32>; ... }
```

Syntax Rational: pointer to function

- Pointer to function types are made more readable:

```
int (*my_func)(int, char *)
            ↓
my_func : <(int<32>, int<8> *)> : int
```

- Once more, the trick is to isolate the type and the identifier.

Syntax Rational: casting

- In C, can you determine if `(x)(y)` is:
  - a cast of `y` to the `x` type, or. . .
  - a call of the function `x` with `y` as an argument?
- Casts in C! obey the unambigous type identifier rules!

```
string = (char *) my_ptr;
                ↓
string = (my_ptr : int<8> *);
```

- This syntax is coherent with previous changes.

Syntax Rational: minor changes

- Trailing `';'` at the end of structure and class declarations is no longer needed;
- Function calls have a single syntax: a functional expression (direct identifier, function pointer, array or structure element) is applied to a vector of arguments;

Syntactic sugar: bit access

- System code often deals with bit fields;
- One can handle it playing with bitwise operations (masks, shifts, . . .);
- Or using bit fields in packed C's structures;
- C! brings another possibility: use unsigned integer as a bit array:

```
switch_feature(flags : int<+32> *) : int<+32>
{
    (*flags)[3] = !(*flags)[3];
    return (*flags)[3];
}
```

Objects in C!

Object Model

- Object in C! is a class-based OOP model
- Intuitively, we design objects in C and then provide a syntaxic sugar for C!
- We choose to keep the object model as simple as we can
- Basicaly we have:
  - Usual class definintions
  - All object's code is in class definintion
  - Methods are *virtual* (in C++ terminology)
  - There's no visibility control (everything is *public*)
  - Methods can be overriden but not overloaded
  - Objects creation is divided in two parts: allocation (delegated to the programmer) and object initialization
- We try to avoid most C++ annoying aspects: no implicit copy, no syntax hell (object, reference, pointer ...), no syntax noise.

Inner Representation

- In C! objects are structures with special inner structure containning function pointers:

```
struct my_object {
  // vtable
  struct my_object_methods *_methods;
  // attributes
  int x,y;
};

struct my_object_methods {
  int (*getX)(void);
  int (*getY)(void);
};
```

Objects Syntax

```
class A {
  x : int<32>;
  get() : int<32> { return x; }
  set(_x : int<32>) : void { x = _x; }
  double() : void
  {
    this.set(2 * this.get());
  }
}

class B {
  x : A;
  get() : A { return x; }
}
```

Objects Syntax

- The syntax embeds all the cooking for objects manipulation in C:
  - Method call such as `a.get()` will be rewritten as `a->_methods->get(a)`
  - Cascading methods call is managed using compound expressions and temporary local variables: the call `b.get().set(42)` will be translated as:

```
({
   A _cbtmp1 = b->_methods->get(b);
   _cbtmp1->_methods->set(_cbtmp1, 42) ;
})
```

  - Actual constructors are build using class variables (class definition can take parameters) visible only at creation time.
  - The constructor always take as first a pointer to a suitable memory location to store the object (of type `void*` to avoid typing issues.)

The Macro Class Concept

Macro Class ?

- The idea is to provide an object like syntax to manipulate non-object data.
- When defining a macro-class we give a base type (the storage type) and a set of *operations*.
- A fake *this* is available in operations code, it represents the inner value
- operations can be *const* (no modifications to the inner value) or not.
- Normally methods calls on macro-class are replace by C macro.
- Macro-class are a simple syntax extensions to handle typed macro over specific data.

Macro class usage

- Macro class can be used where object-oriented syntax are convenien but full-object implementation is useless.
- A classical example is evolved integer flags such as status return of a `wait` syscall. One can define setting and testing of various properties of the status as method call that in fact will implement the usual macro.
- The purpose is to provide convenient syntax for low level behaviour code (for exemple, it will possible to include assembly code in a macro operations) without any overhead or runtime code.

Example

```
macro class MC : int<+32>
{
  get() const : int<+32>
  {
    return (this * 2);
  }

  set(x : int<+32>) : void
  {
    this = x/2;
  }
}

f() : void
{
  x : MC;
  // x is really an int<+32>
  x = 0;
  // but it can be used like an object
  x.set(42);
}
```

State of The Prototype

State of The Prototype

- Lexing/Parsing : check;
- Type checking : check, even if a few improvements are scheduled;
- Module support: very partial... but coming soon! (the identifiers mangling is still to be decided)
- Object support: check;
- Macro class support: partial;
- Stay tuned!

State of The Prototype

- Lexing/Parsing : check;
- Type checking : check, even if a few improvements are scheduled;
- Module support: very partial... but coming soon! (the identifiers mangling is still to be decided)
- Object support: check;
- Macro class support: partial;
- Stay tuned!

State of The Prototype

- Lexing/Parsing : check;
- Type checking : check, even if a few improvements are scheduled;
- Module support: very partial... but coming soon! (the identifiers mangling is still to be decided)
- Object support: check;
- Macro class support: partial;
- Stay tuned!

State of The Prototype

- Lexing/Parsing : check;
- Type checking : check, even if a few improvements are scheduled;
- Module support: very partial… but coming soon! (the identifiers mangling is still to be decided)
- Object support: check;
- Macro class support: partial;
- Stay tuned!

State of The Prototype

- Lexing/Parsing : check;
- Type checking : check, even if a few improvements are scheduled;
- Module support: very partial… but coming soon! (the identifiers mangling is still to be decided)
- Object support: check;
- Macro class support: partial;
- Stay tuned!

State of The Prototype

- Lexing/Parsing : check;
- Type checking : check, even if a few improvements are scheduled;
- Module support: very partial... but coming soon! (the identifiers mangling is still to be decided)
- Object support: check;
- Macro class support: partial;
- Stay tuned!

State of The Prototype

- Lexing/Parsing : check;
- Type checking : check, even if a few improvements are scheduled;
- Module support: very partial... but coming soon! (the identifiers mangling is still to be decided)
- Object support: check;
- Macro class support: partial;
- Stay tuned!

Future Directions

Work in progress

- Finish and fix work in the compiler
- Fix syntax for inlining
- Fix the object constructor troll
- Solve various typing issues
- Find a better way to implement class macro ops (inlined functions ?)
- Have a tool chain ...

Future Works ?

- Data mapping: automatic access to register like data
- Method Overloading
- Plugin and annotation (external processing of code)
- Generics and/or Template
- Global static constructions
- Native compiler (does-it make sense ?)
- Self-host (C! in C!)
- Front-end integration in gcc/clang
- Code validation and static checking

QUESTIONS ?