Objects? No, thanks! (Using C++ effectively on small systems)

Wouter van Ooijen, Hogeschool Utrecht - February 15, 2014

My field of interest is using C++ on small microcontrollers. Such chips are traditionally written in C or even assembler, but in my opinion C++ has much to offer, if used appropriately. In this article I will show how using static class templates can be used to get compile time flexibility without paying a price in size or speed.

I will use the blinking of an LED as an example, because it is the very first thing a microcontroller programmer does on a new chip, and it has two parts: the blinking, and the manipulation of the IO pin. These two aspects beg to be handled separately, otherwise writing N such algorithms for M chips would require N x M specific programs. Separating the two aspects reduces the effort to N algorithms plus M pieces of pin access code.

As an example, Iâ€™ll use an LPC1114FN28 chip with a Cortex M0 CPU, 4K RAM, 32K ROM. To blink an LED on pin 0 of port 1 of this chip a C programmer might write something like the following code. Various details, like configuring the pin for GPIO, setting its direction to output, and realizing the delay() function, have been omitted to concentrate on the blinking and the handling of the pin. The GPIOREG macro provides access to an IO pin related register; the PIN_SET macro does the actual setting of the pin. The main() calls these two macroâ€™s in a loop, interspaced with delay() calls.

```
// typical blink-an-LED in C

   // return the pointer to an IO port related register
#define GPIOREG( port, offset ) \
   (*(volatile int *)( 0x50000000 + (port) * 0x10000 +
(offset)))

   // set the pin port.pin to value
#define PIN_SET( port, pin, value ) \
   { GPIOREG( (port), 0x04 << (pin) ) = (value) ? -1 : 0;
}

int main(){
   for(;;){
      PIN_SET( 1, 0, 1 );
      delay();
      PIN_SET( 1, 0, 0 );
      delay();
   }
}
```

Both macros might seem pretty complicated, but the compiler can translate a call of PIN_SET with literal arguments into just a few machine instructions. Note in particular that the address arithmetic and the choice implied by the ? operator have been eliminated because the input values are constants.

```
// assembler listing fragment for the blink-an-LED
for(;;){ . . . } loop

.L97:
    ldr    r4, .L98       // get the I/O address
    mov    r3, #1         // get -1 into r3
    neg    r0, r3
    str    r0, [r4]       // store -1 in the I/O register
    bl     delay
    mov    r1, #0
    str    r1, [r4]       // store 0 in the I/O register
    bl     delay
    b      .L97
```

Now consider how a C++ library might help programmers of such a chip by providing a nice OO abstraction for a pin of a microcontroller. A function could be provided that does essentially the same thing as the PIN_SET macro (and by providing it in a header, the same optimizations would be possible), but this does not make a pin something that can be treated as an abstraction. For instance, the fact that a pin on the LPC1114 is characterized by its port and pin number does not hold for all IO pins on all chips. Instead a pin is, in true OO fashion, represented by an object that implements an abstract interface. For this simple case the interface contains only one method: set(bool), which makes the pin either high or low, depending on the value passed. A concrete class derives from this interface class and implements the method.

```
// an OO interface for a pin, and an LPC1114
implementation

struct pin_out {
   virtual void set( bool ) = 0;
};

volatile int & gpioreg( int port, int offset ){
   return * (volatile int *) ( 0x50000000 + port *
0x10000 + offset );
}

struct lpc1114_gpio : public pin_out {
   const int port, pin;

   lpc1114_gpio( int port, int pin )
      : port( port ), pin( pin ){}

   void set( bool x ){
      *gpioreg( port, 0x04 << pin ) = x ? -1 : 0;
   }
};
```

With this abstraction, code can be written that manipulates a pin without knowing the details of how this is done. The blink function below takes a pin_out as parameter. It is fully decoupled from the concrete pin and the actual operations needed to make the pin high or low. This nicely breaks the N x M problem. The blink application now only has to create the pin object and pass it to the blink function. This could of course be done in a single line, but it is good style to give the object a meaningful name.

```
// blinking the LED using a pin object

void blink( pin_out & pin ){
   for(;;){
      pin.set( 1 );
      delay();
      pin.set( 0 );
      delay();
   }
}

int main(){
   lpc1114_gpio led( 1, 0 );
   blink( led );
}
```

If this code would be as efficient as the C code shown earlier, the decoupling of the pin access from the algorithm would probably be welcomed by microcontroller programmers. Or, at the very least, they would not discard this approach, and C++ with it, on first sight. But alas, there is a price to be paid for this abstraction in RAM use, code size, and speed.

Each concrete object that inherits from pin_out has at least a virtual function table pointer (4 bytes on a 32-bit CPU), and probably some more information that identifies the point. In the LPC1114 implementation, two integers are used (4 bytes each). This could be easily reduced to one or two bytes, but on a Cortex CPU objects must be 4-byte aligned, so the practical minimum size for a pin object is 8 bytes.

The LPC1114DFN28 chip has 4K RAM and 22 pins that can be used as output. When objects are created for all 22 pins, this would claim 8 x 22 = 176 bytes, which is 4% of the available RAM. To some this might seem insignificant, but it is a good argument for C programmers that C++ is wasteful of resources.

The use of RAM can be avoided by creating the pin objects in ROM, of which a microcontroller generally has more. A problem with this is that it forces all pin-using algorithms to work with const objects, which makes it difficult to do interesting things in a pin that require state - for instance optimizing writes by writing out only changes.

```
// OO pin abstraction and implementation with const
```

```
objects

struct pin_out {
   virtual void set( bool ) const = 0;
};

struct lpc1114_gpio : public pin_out {
   int port, pin;

   constexpr lpc1114_gpio( int port, int pin )
      : port( port ), pin( pin ){}

   void set( bool x ) const {
      gpioreg( port, 0x04 << pin ) = x ? -1 : 0;
   }
};

void blink( pin_out const & pin ){
   . . .
}

int main(){
   constexpr lpc1114_gpio led = lpc1114_gpio( 1, 0 );
   blink( led );
}
```

The example pin_out abstraction has only one method. A realistic one would have many more: for configuring the pin as GPIO, for setting the direction, for reading the level, for enabling and disabling the weak pull-ups, etc. These methods would all be virtual in the abstract pin class, and implemented (probably differently) in each concrete pin class. Current C++ compilers are not good at optimizing away unused virtual method implementations, so all methods for all pins are going to be present in the ROM image. For a desktop, code size is almost a non-issue these days, but for a small microcontroller it can be a killer. This issue would discourage a library author from providing a rich interface for a pin. Alternatively, it would force a microcontroller programmer not to use such a library, because the compiled code would not fit in his target chip.

The fact that led.set() is a virtual method call means that it is somewhat slower than a plain method or function call. The overhead is only a little arithmetic and an indirect call, but when the method is just a few assembler instructions this overhead is significant. Much worse, because the compiler in general canâ€™t know which method is called, it canâ€™t inline the method code, and hence canâ€™t apply constant folding, dead code elimination, and other nice optimizations. The simple action of making a pin high or low now takes much more code and hence more CPU cycles than it ought to compared to the C version.

Now take a step back. WHY does the OO version have so much overhead compared to the C version? The crucial difference is that in the OO version blink() can be called with any pin object, whereas the C version is for a specific pin. This might sound like a big advantage to an

OO programmer, but for a small microcontroller the use of its pins is generally fixed by the hardware it is part of. It would take at least a soldering iron to connect the LED to a different pin. If we are going to do that, we might as well re-compile the software for the new pin too.
Page 2 of 2
What is needed is a way to configure the blink() function for a specific pin in a way that is as decoupled from the actual pin implementation as the OO version, but that allows the compiler to do the same optimizations the C version allowed. To make this possible the coupling between the blink() function and the actual pin can and must be made at compile time. The obvious C++ realization is to make blink() a function template, with the pin supplied as template parameter.

A template parameter can't be a user-defined object, but it can be a type. In the next static-class-template based version of our blink-an-LED example, a pin is represented by a type: an instantiation of the lpc1114_pin class template. Instead of the virtual method of the pin_out object, the class template has a static method. This enables the compiler to 'see' which method is called, so it can do its optimizations magic. Note that a particular compiler might not do all optimizations, but, contrary to the situation with virtual methods, it has all the information, so at least it could.

```
// template based blink-an-LED

template< int port, int pin >
struct lpc1114_pin {
   static void set( bool x ) const {
      gpioreg( port, 0x04 << pin ) = x ? -1 : 0;
   }
};

template< class pin >
void blink(){
   for(;;){
      pin::set( 1 );
      delay();
      pin::set( 0 );
      delay();
   }
}

int main(){
   typedef lpc1114_pin< 1, 0 > led;
   blink< led >();
}
```

The result is as expected: the compiler inlines the pin::set calls and optimizes the loop to essentially the same machine code as the C version.

```
.L65:
    ldr    r4, .L66
    mov    r3, #1
    neg     r3, r3
    str     r3, [r4]
    bl      _Z5delayv
    mov     r3, #0
    str     r3, [r4]
    bl      _Z5delayv
    b       .L65
```

This static class template approach has other advantages compared to the objects-with-virtual-methods approach: Static methods are handled by the compiler and linker as global functions, so only the ones that are used need to be kept. And there are no objects, so there is no chance to create dangling references.

There are some (potential) problems too. Each template instantiation (with different arguments) is a completely new class, with methods that are distinct from those of other instantiations. If the amount of duplicated code in a method is substantial, this can lead to code bloat. This can be solved by migrating common code to a non-template class, or even to global functions. Note that such code will no longer benefit from the optimizations that are made possible by knowing the actual method parameters, so there is a tradeoff to make. In our case the code is very small and benefits significantly from inlining, so there is no need to migrate code to a non-template class.

Another problem is that a template by itself does no checking on its arguments. Each template instantiation is in effect re-compiled with the supplied template parameters, at which time any problems that surface will be reported by the compiler, pointing to the location in the template where the instantiation caused a problem. This is of course not very helpful for the user, who wants to know which template parameter he supplied caused the problem, and what he should do about it. The message below is what I get when I am in a funny mood and try to blink an int.

```
int main(){
   blink< int >();
}

main.cpp: In instantiation of 'void blink() [with pin =
int]':
main.cpp:733:17:   required from here
main.cpp:687:7: error: 'set' is not a member of 'int'
main.cpp:689:7: error: 'set' is not a member of 'int'
```

In this case the error message is not that bad, because the template has only one parameter, and the error occurs in the one template itself, not in a deeply nested sub-template. But it is still a lot worse than what one would get for passing a parameter of the wrong type to a method.

There are ways to do compile time checking on the properties of template arguments. The approach I favor so far is to have an identification in each type that in effect states that the type conforms to some defined interface. A template must check, for each argument, that it provides the interface that it expects, or give an appropriate (compile-time) error message. This is not fool-proof: a user could write a class that has the identification type but not the required static methods, which would cause an error message from somewhere “deep down”.

On the other hand, with the identification type approach a user must consciously insert the identification type, reducing the chance that a class that happened to have a set(bool) method (with completely inappropriate semantics) is inadvertently accepted, which would be the case when the template (only) checked for the presence of that method.

Separation of the algorithm (blink) from the object it uses (the pin) is nice, but it is only a first step. Templates can be written that act as a pin_out, and take a pin_out as a template parameter, but add a twist to the semantics - this is known in OO land as the Decorator pattern. A very simple but useful decorator is one that behaves like a pin, but writes to both argument pins. This would be handy when there are LEDs on two different pins that must be blinked together.

```
// blink two LEDs in unison

template< class pin1, class pin2 >
struct both {
   static void set( bool x ){
      pin1::set( x );
      pin2::set( x );
   }
};

int main(){
   typedef lpc1114_pin< 1, 0 > led0;
   typedef lpc1114_pin< 1, 1 > led1;
   typedef both< led0, led1 > leds;
   blink< leds >();
}
```

The ‘pins’ that can be supplied to both<> are not limited to real pins: it can be any class that has a static set(bool) operation. It could for instance log what is written to the pin, as shown in the next code.

```
// log all writes to the LED

struct logger {
   static void set( bool x ){
      std::cout << "LED set to " << x << "\n";
   }
};

int main(){
   typedef lpc1114_pin< 1, 0 > led;
   typedef both< led, logger > logged_led;
   blink< logged_led >();
}
```

With some template magic, both<> can be extended to all<>, which takes an arbitrary number of pin parameters and writes to them all.

```
// all< list >::set(x) calls set(x) on all pins in list

template< typename... > struct all {};

template<> struct all<> {
   static void set( unsigned int x ){}
};

template< typename pin, typename... tail > struct all <
pin, tail... >{
   static void set( unsigned int x ){
      pin::set( x & 0x01 );
      all< tail... >::set( x >> 1 );
   }
};
```

The important point is not so much that with tools like all<> it is easy to do things like adding logging, but that it is possible to do so without touching either the blink code or the pin_out code.
A more complex example is shown below. The 74HC595 is a shift-register chip with serial input and an 8-bit parallel output. Three pins of a microcontroller are needed to control this chip. Hence such a chip can be used when a design needs more output pins than the microcontroller can provide. In fact, the 3 microcontroller pins can be used to control a number of daisy-chained 74HC595 chips.
The code below uses the hc595 template. It is instantiated with three pins, and provides 8 pins (q0 . . . q7, as they are called in the chips datasheet). These 8 pins offer the normal pin_out interface, so they can be used interchangeably with output pins of the microcontroller, for instance as argument to blink<>.

```
// blink an LED that is connected to a 74HC595 chip

template< class data, class shcp, class stcp >
class hc595 {
   . . .
   typedef . . . q0;
   . . .
   typedef . . . q7;
}

typedef hc595<
   lpc1114_pin< 1, 0 >,
   lpc1114_pin< 1, 1 >,
   lpc1114_pin< 1, 2 >
> hc595_chip;
```

```
int main(){
   blink< hc595_chip::q2 >::blink();
}
```

The hc595 template requires three pins to provide the pin_out interface. The pins provided by this macro provide that same interface. Hence they can be used to interface to a second (cascaded) hc595, as shown below.

```
// blink an LED that is connect to a cascaded 74HC595
chip

typedef hc595<
   lpc1114_pin< 1, 0 >,
   lpc1114_pin< 1, 1 >,
   lpc1114_pin< 1, 2 >
> hc595_chip1;

typedef hc595<
   hc595_chip1::q0,
   hc595_chip1::q1,
   hc595_chip1::q2
> hc595_chip2;

int main(){
   blink< hc595_chip2::q2 >::blink();
}
```

This shows how the static-class-template approach can offer the same ‘mix-and-match’ flexibility that a standard OO approach would have offered, without the costs. But note that the mixing-and-matching can be done only at compile time. What this approach sacrifices is the ability to decide at run time how a pin is used. In my opinion this is a worthwhile sacrifice for the gains in code and data size and run-time performance.

The examples were built with G++ 4.7.4 (-Os) and run on an LPC1114FN28. Note that using a pin of this chip as output might require configuring it as GPIO and setting its direction, which is shown.

***Wouter van Ooijen** is a software engineer by profession and a hardware tinkerer by passion. He got his degree in Informatics from the Delft University of Technology. He has worked on embedded systems for industry, space, and military applications. Currently he teaches at the Technical Informatics section of the Hogeschool Utrecht, Netherlands. His main interest is the borderline between hardware and software. He is working on a library that uses the principles described in this article.*