Flamebait

Paul M. Bendixen

February 3, 2015

A good flamebait

- Vim vs. Emacs

A good flamebait

- Vim vs. Emacs
  Too equal, too many uses other stuff (like sublime???)

A good flamebait

- Vim vs. Emacs
  Too equal, too many uses other stuff (like sublime???)
- Tabs vs. Spaces

A good flamebait

- Vim vs. Emacs
  Too equal, too many uses other stuff (like sublime???)
- Tabs vs. Spaces
  Too intern, too logical

A good flamebait

- Vim vs. Emacs
  Too equal, too many uses other stuff (like sublime???)
- Tabs vs. Spaces
  Too intern, too logical
- C++ in embedded!!

Embedded (my definition)

Embedded systems are characterised by:

- Bare Metal (No operating system)
- Limited RAM
- Limited ROM
- no MMU
- no standard console (perhaps UART or LCD)

C++ (my definition)

The C++ subset I use in this talk:

- C++03
- No dynamic memory (new/delete)
- → No or very limited STL
- No RTTI
- No exceptions

This is *not* your desktop's C++

A little detour

Embedded C++ removes the following:

- mutable
- Exceptions
- RTTI
- Namespaces
- Templates
- Multiple Inheritance
- Virtual base classes
- C++ style casts

source:`http://www.caravan.net/ec2plus/rationale.html`

Common arguments

- C++ leads to bloat
- C++ is slower (less effective) than C
- Some things you need C (or assembly) for

Clearing out the FUD

With a typical “Hello, world” program, compiled with GCC:

C	8511 bytes
C++	9128 bytes
C++ -O3	8994 bytes
C with g++	8511 bytes

Clearing out the FUD

With a typical “Hello, world” program, compiled with GCC:

C	8511 bytes
C++	9128 bytes
C++ -O3	8994 bytes
C with g++	8511 bytes

Case closed C++ == bloat.

Clearing out the FUD

With a typical “Hello, world” program, compiled with GCC:

C	8511 bytes
C++	9128 bytes
C++ -O3	8994 bytes
C with g++	8511 bytes

Case closed C++ == bloat.

- no standard console (perhaps UART or LCD)

Clearing out the FUD

C++ adds overhead

Clearing out the FUD

C++ adds overhead so you don't have to

Clearing out the FUD

C++ adds overhead so you don't have to

- Virtual functions are really just optimized function pointers
- Templated functions adds compile time polymorphism
- Classes are just structs, with more.

Why C++ over C

```
template<typename Callable>
struct call_helper_unfold<Callable , std::type_array<>,
typename std::enable_if<
!internal_worker::is_function_pointer<Callable >::callable >::type
>
: assert_is_function_pointer<decltype(&Callable::operator())>
{
using type = call_helper_data<decltype(&Callable::operator())>;
};
```

Why C++ over C

```
template<typename Callable>
struct call_helper_unfold<Callable, std::type_array<>,
typename std::enable_if<
!internal_worker::is_function_pointer<Callable>::callable>::type
>
: assert_is_function_pointer<decltype(&Callable::operator())>
{
using type = call_helper_data<decltype(&Callable::operator())>;
};
```

Just. . . don't.

Why C++ over C

```
class GPIO
{
        public:
                inline void turnOn( uint8_t bit )
                {
                        DOUTSET = 1U << ( bit & 0x0F );
                }
        private:
                volatile uint32_t CTRL;
                volatile uint32_t MODEL;
                volatile uint32_t MODEH;
                volatile uint32_t DOUT;
                volatile uint32_t DOUTSET;
                /* ... */
                volatile uint32_t DIN;
                /* ... */
}
```

Why C++ over C

```
class GPIO
{
        public:
                inline void turnOn( uint8_t bit )
                {
                        PORT |= 1U << ( bit & 0x07 );
                }
        private:
                volatile uint8_t PORT;
                volatile uint8_t DDR;
                volatile uint8_t PIN;
}
```

The init function

```
void init()
{
initA();
initB();
initB();
}
int main( void )
{
init();
useC();
}
```

constructors

```
int main( void )
{
GPIO portA __attribute__(section(.portA));
LedDriver led( portA, 3 );
led.turnLedOn();
}
```

Using templates (with care)

Templated code can be used for compile time polymorphism

```
template< typename HW >
class LedDriver
{
        public:
                LedDriver( HW& ledPort, uint8_t pinNum )
                        : gpio( ledPort ), pin( pinNum )
                {}
                void turnLedOn( void )
                {
                        gpio.turnOn( pin );
                }
        private:
                uint8_t pin;
               HW& gpio;
};
```

Using templates (with care)

Templated code can be used for compile time polymorphism

```
template< typename HW >
class LedDriver
{
        public:
                LedDriver( HW& ledPort, uint8_t pinNum )
                        : gpio( ledPort ), pin( pinNum )
                {}
                void turnLedOn( void )
                {
                        gpio.turnOn( pin );
                }
        private:
                uint8_t pin;
                HW& gpio;
};
```

Allows for multiple different hardware drivers. Even mocks.

What you can't do in C++

C is almost a subset of C++

What you can't do in C++

C is almost a subset of C++

"But what about interrupts?"

What you can't do in C++

C is almost a subset of C++

"But what about interrupts?"

`http://www.drdobbs.com/implementing-interrupt-service-routines/184401485`

Caveat emptor

C++ is great for embedded programming but:

- You are going to write a lot of code yourself
- Know what you are doing
- measure
- look in the assembly
- be aware of the new keyword