Embedded System Design Techniques™

Transitioning from C to C++

Class 4: Real-time C++

October 12th, 2017
Jacob Beningo

Presented by:

DesignNews

CEC CONTINUING EDUCATION CENTER

Digi-Key ELECTRONICS

Course Overview

Topics:

- C++ Fundamentals
- Designing a C++ Application
- Beginning the Transition
- **Real-Time C++**
- Getting into the Bits and the Bytes

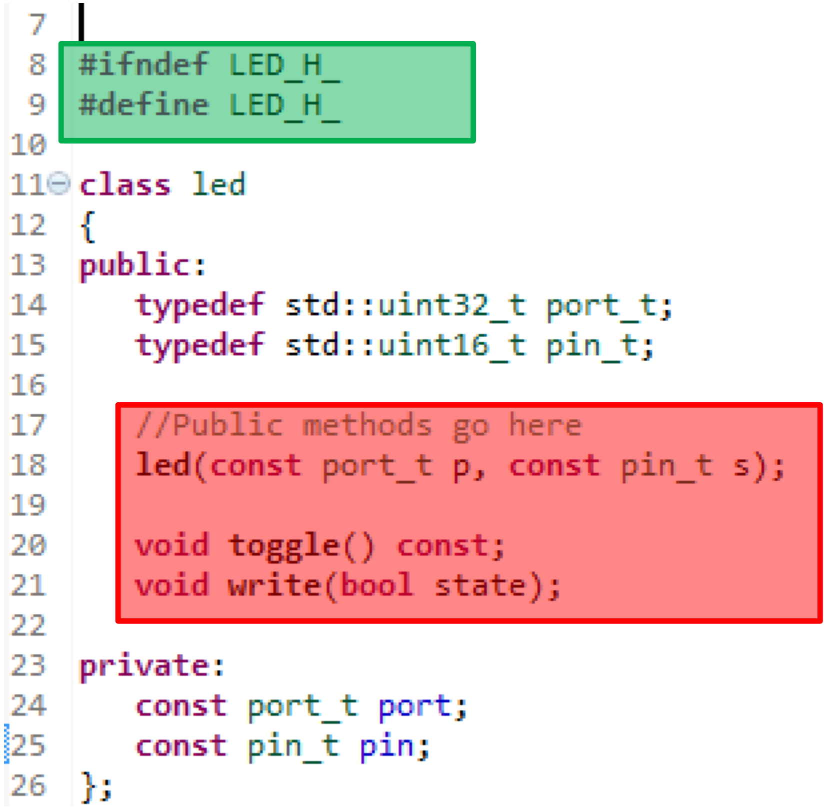
Session Overview

- Inline Definitions
- Templates
- Inheritance
- Polymorphism
- Virtual Functions

Presented by:

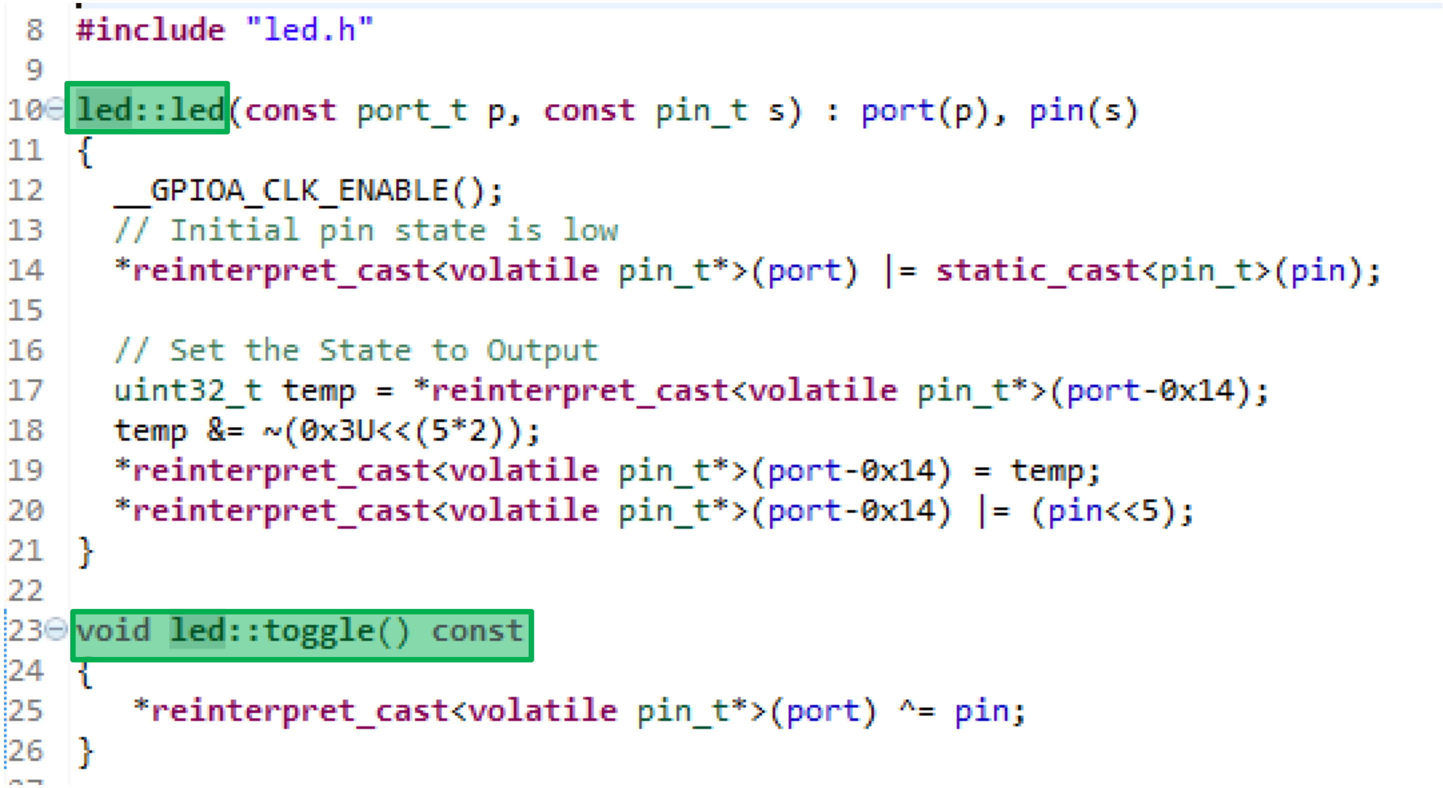
Inline definitions

Led.h



```

#ifndef LED_H_
#define LED_H_

class led
{
public:
    typedef std::uint32_t port_t;
    typedef std::uint16_t pin_t;

    //Public methods go here
    led(const port_t p, const pin_t s);

    void toggle() const;
    void write(bool state);

private:
    const port_t port;
    const pin_t pin;
};
```

Inline Definitions



```
#include "led.h"

led::led(const port_t p, const pin_t s) : port(p), pin(s)
{
  __GPIOA_CLK_ENABLE();
  // Initial pin state is low
  *reinterpret_cast<volatile pin_t*>(port) |= static_cast<pin_t>(pin);

  // Set the State to Output
  uint32_t temp = *reinterpret_cast<volatile pin_t*>(port-0x14);
  temp &= ~(0x3U<<(5*2));
  *reinterpret_cast<volatile pin_t*>(port-0x14) = temp;
  *reinterpret_cast<volatile pin_t*>(port-0x14) |= (pin<<5);
}

void led::toggle() const
{
   *reinterpret_cast<volatile pin_t*>(port) ^= pin;
}
```

Templates

A template allows a developer to use the same code for different types.

- Improves code flexibility
- Easier program maintenance

Syntax

```
template<typename T>
T add(const T& a, const T& b)
{
   return a + b;
}
```

Use

```
int add(2,3);
```

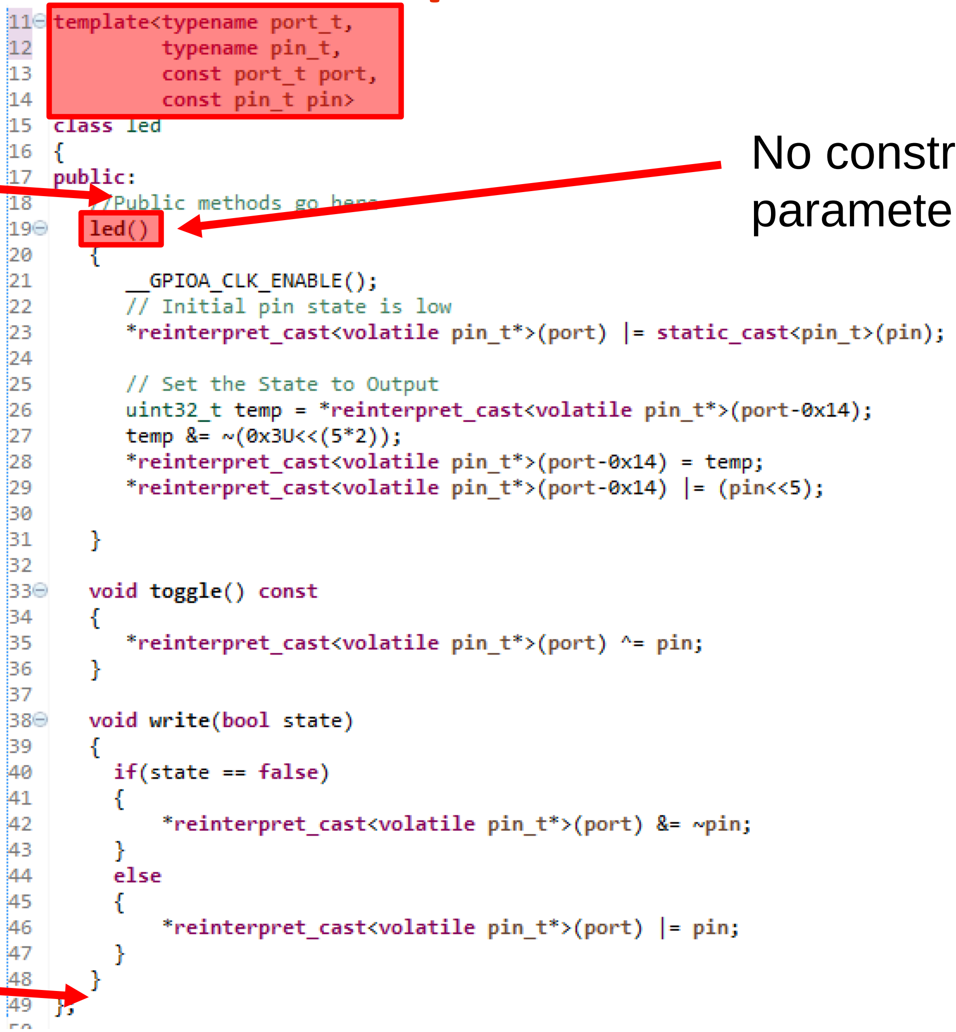
Result: 5

Use

```
uint8_t add(4,3);
```

Result: 7

Templates



```
template<typename port_t,
         typename pin_t,
         const port_t port,
         const pin_t pin>
class led
{
public:
    //Public methods go here
    led()
    {
        __GPIOA_CLK_ENABLE();
        // Initial pin state is low
        *reinterpret_cast<volatile pin_t*>(port) |= static_cast<pin_t>(pin);

        // Set the State to Output
        uint32_t temp = *reinterpret_cast<volatile pin_t*>(port-0x14);
        temp &= ~(0x3U<<(5*2));
        *reinterpret_cast<volatile pin_t*>(port-0x14) = temp;
        *reinterpret_cast<volatile pin_t*>(port-0x14) |= (pin<<5);

    }

    void toggle() const
    {
        *reinterpret_cast<volatile pin_t*>(port) ^= pin;
    }

    void write(bool state)
    {
      if(state == false)
      {
          *reinterpret_cast<volatile pin_t*>(port) &= ~pin;
      }
      else
      {
          *reinterpret_cast<volatile pin_t*>(port) |= pin;
      }
    }
};
```

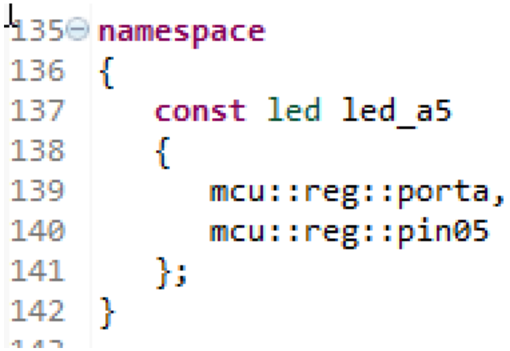
No typedefs!

No constructor parameter lists!

No private variables!

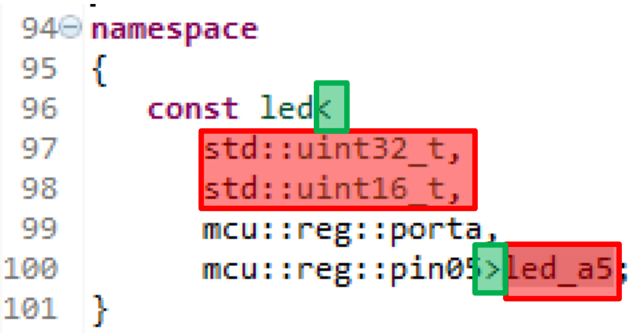
Presented by:

Templates



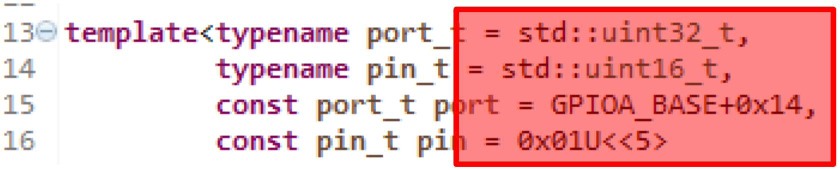
```
namespace
{
    const led led_a5
    {
        mcu::reg::porta,
        mcu::reg::pin05
    };
}
```

Becomes >



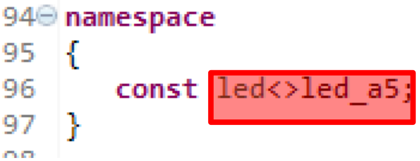
```
namespace
{
    const led<
        std::uint32_t,
        std::uint16_t,
        mcu::reg::porta,
        mcu::reg::pin05>led_a5;
}
```

Default values and types



```
template<typename port_t = std::uint32_t,
         typename pin_t = std::uint16_t,
         const port_t port = GPIOA_BASE+0x14,
         const pin_t pin = 0x01U<<5>
```

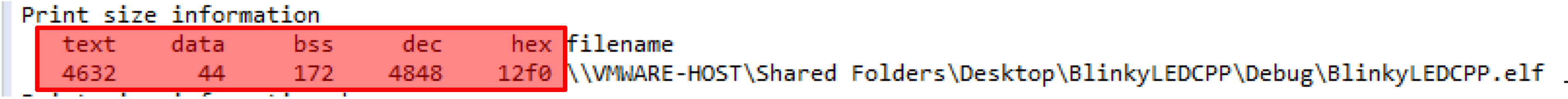
Default values and types



```
namespace
{
    const led<>led_a5;
}
```

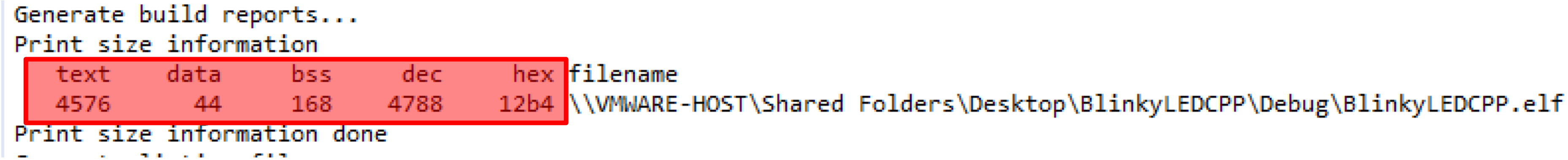
Templates

- Non-templated LED Program



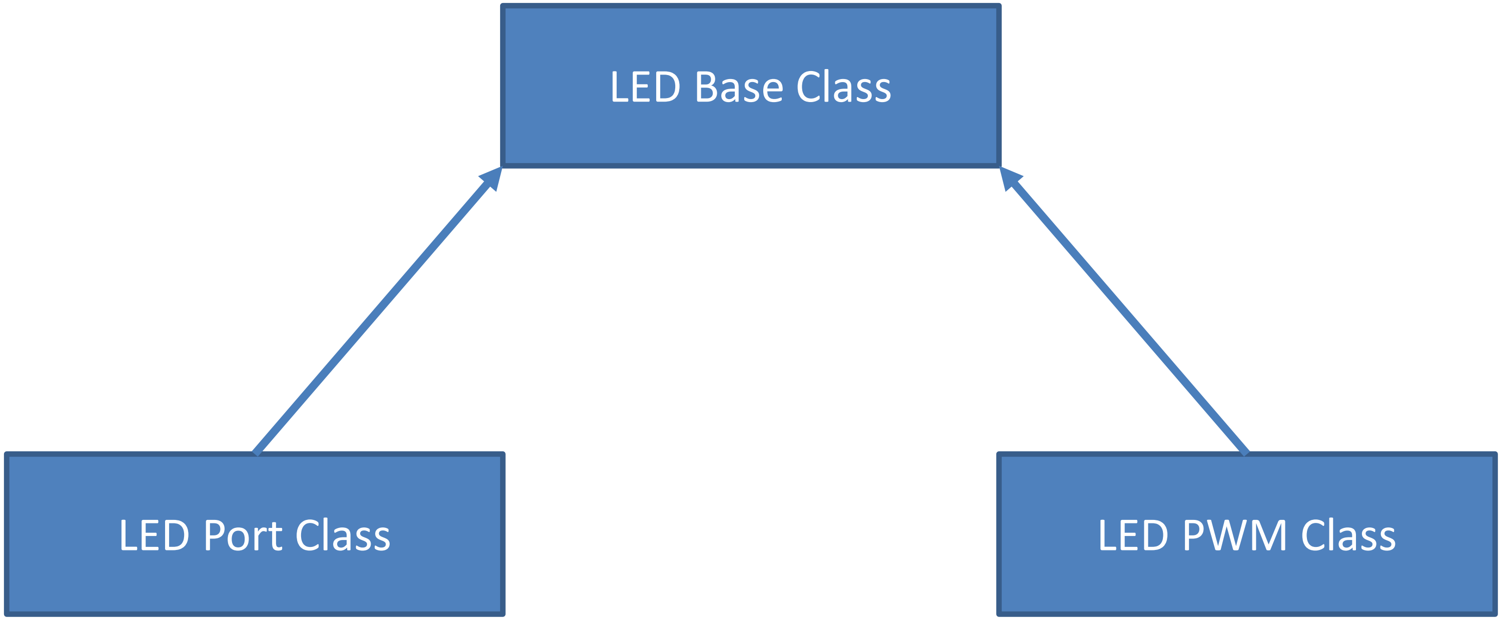
```
Print size information
   text    data     bss     dec     hex filename
   4632      44     172    4848    12f0 \\VMWARE-HOST\Shared Folders\Desktop\BlinkyLEDCPP\Debug\BlinkyLEDCPP.elf
```

- Templated LED Program

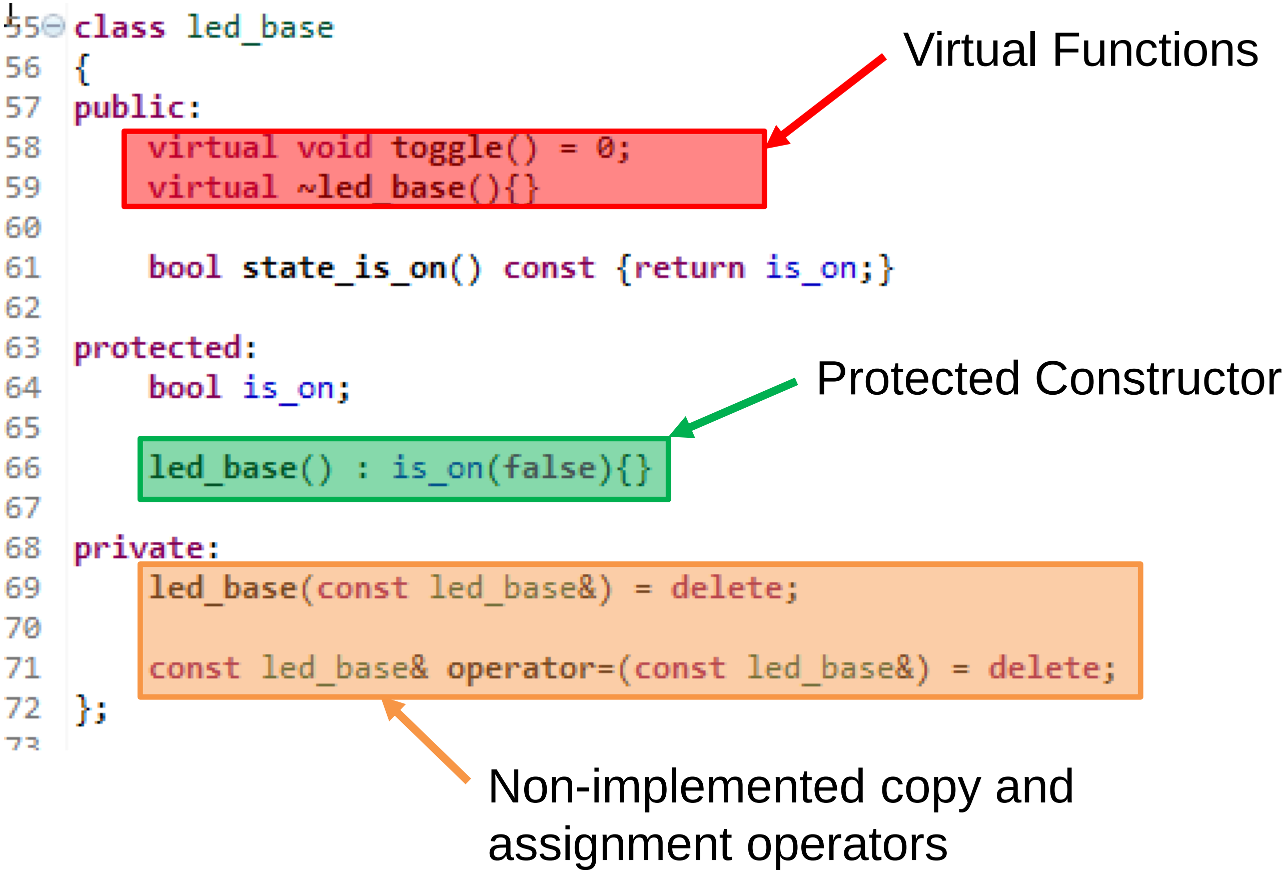


```
Generate build reports...
Print size information
   text    data     bss     dec     hex filename
   4576      44     168    4788    12b4 \\VMWARE-HOST\Shared Folders\Desktop\BlinkyLEDCPP\Debug\BlinkyLEDCPP.elf
Print size information done
```

Inheritance



Inheritance



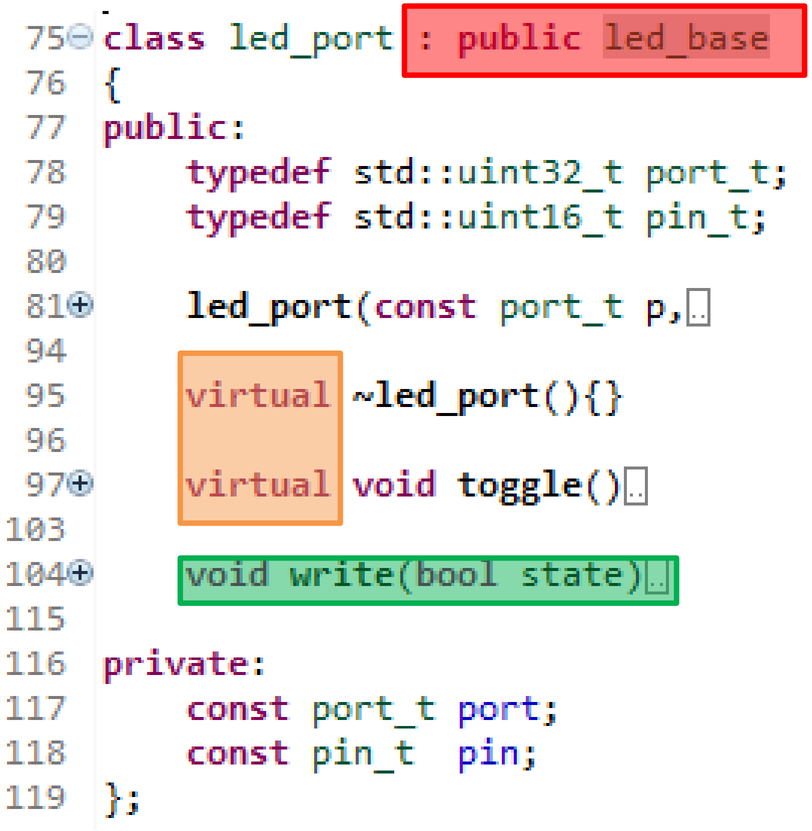
```
class led_base
{
public:
    virtual void toggle() = 0;
    virtual ~led_base(){}

    bool state_is_on() const {return is_on;}

protected:
    bool is_on;

    led_base() : is_on(false){}

private:
    led_base(const led_base&) = delete;

    const led_base& operator=(const led_base&) = delete;
};
```

Virtual Functions

Protected Constructor

Non-implemented copy and assignment operators

Inheritance



```
class led_port : public led_base
{
public:
    typedef std::uint32_t port_t;
    typedef std::uint16_t pin_t;

    led_port(const port_t p,..

    virtual ~led_port(){}

    virtual void toggle()..

    void write(bool state)..

private:
    const port_t port;
    const pin_t  pin;
};
```

Polymorphism

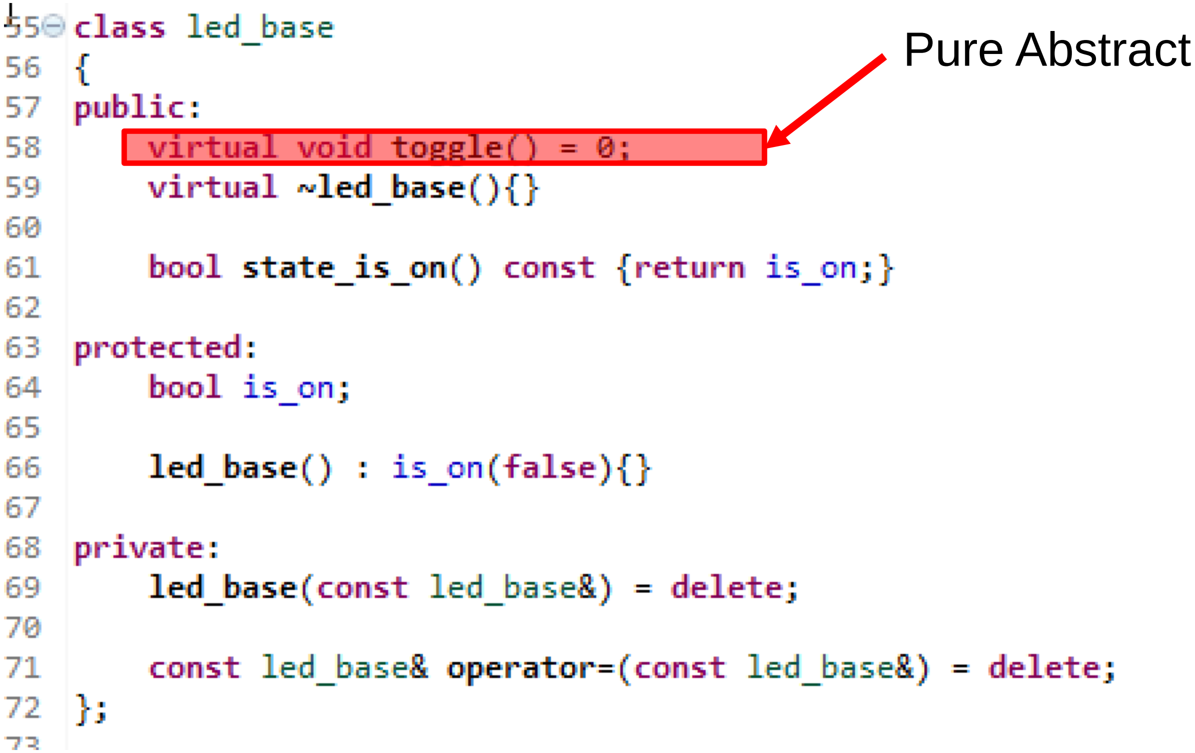
Polymorphism – providing a single interface to entities of different types.

Dynamic Polymorphism – uses a runtime virtual function mechanism to call methods of a derived class by accessing them from a base class pointer or reference.

Dynamic Polymorphism

```
void led_toggler(led_base* led)
{
  // Toggle LED by dynamic polymorphism
  led->toggle();
}

void do_something()
{
  led_toggler(&led_a5);  // LED Port Object
  led_toggler(&led_b7);  // LED PWM Object
}
```

Virtual Functions



```
class led_base
{
public:
    virtual void toggle() = 0;
    virtual ~led_base(){}

    bool state_is_on() const {return is_on;}

protected:
    bool is_on;

    led_base() : is_on(false){}

private:
    led_base(const led_base&) = delete;

    const led_base& operator=(const led_base&) = delete;
};
```

Pure Abstract

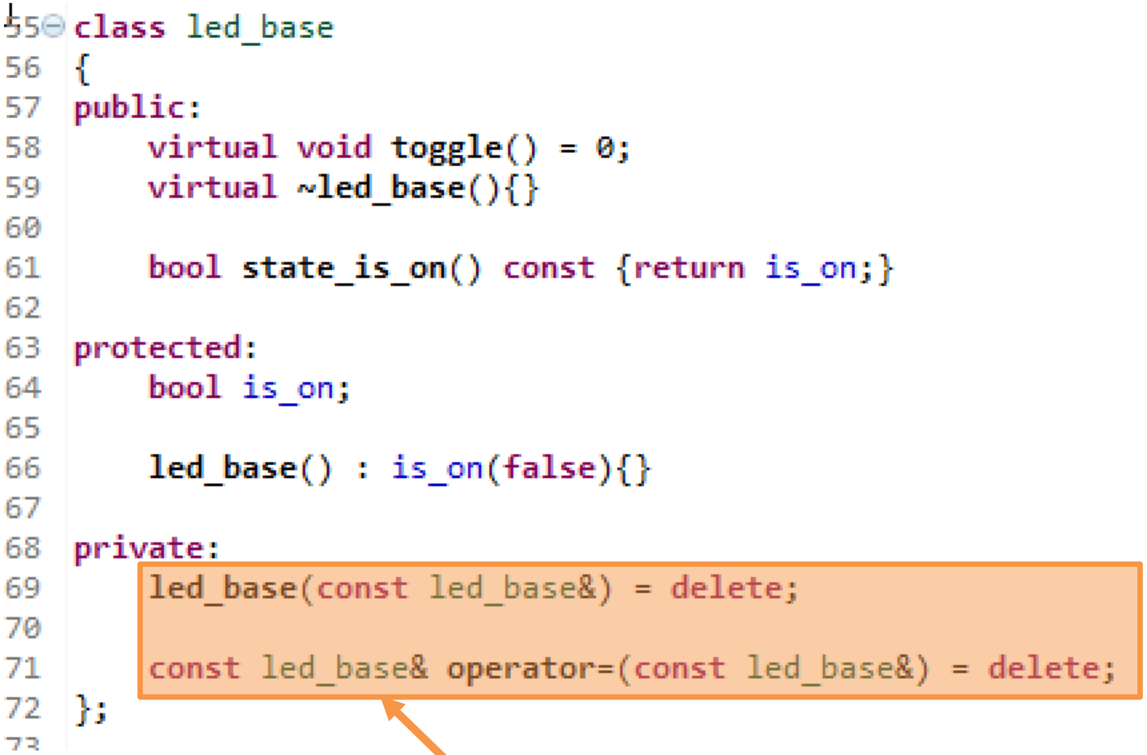
Class Relationships

- **is-a** relationship – derived class is-a subclass of the base class

- **has-a** relationship – class has something, such as a relationship with a member variable. ie. **has-a** pwm object

- **uses-a** relationship – class uses something such as a pwm object

Non-copyable Classes



```
class led_base
{
public:
    virtual void toggle() = 0;
    virtual ~led_base(){}

    bool state_is_on() const {return is_on;}

protected:
    bool is_on;

    led_base() : is_on(false){}

private:
    led_base(const led_base&) = delete;

    const led_base& operator=(const led_base&) = delete;
};
```

Non-implemented copy and assignment operators

Best Practices

- Use constexpr or enum to create constants, don't use #define
- Carefully monitor code size, memory usage and execution overhead as you develop
- Use protect for abstract class constructors and other data that should be accessible to derived classes
- If methods do not need to write data, make them const (there is no penalty!)
- Make class non-copyable for low-level hardware

Additional Resources

- Download Course Material for
  - C/C++ Doxygen Templates
  - Example source code
  - Blog
  - YouTube Videos
- Embedded Bytes Newsletter
  - http://bit.ly/1BAHYXm

From www.beningo.com under

- Blog > CEC – Designing IoT Sensor Nodes using the ESP8266

The Lecturer – Jacob Beningo

Jacob Beningo

Principal Consultant

BENINGO EMBEDDED GROUP

Social Media / Contact

- E : jacob@beningo.com
- T : 810-844-1522
- : Jacob_Beningo
- f : Beningo Engineering
- in : JacobBeningo
- EDN : Embedded Basics

ARM Connected Community

Consulting

- Advising
- Coaching
- Content
- Consulting
- Training

www.beningo.com