Using Modern C++ to Improve Code Clarity

David Rowland
Tracktion

Recap From Last Year

Things a Programmer has to Consider

- Performance
  - Battery life
  - Responsiveness
  - Graphics for better UI/UX
  - Future improvements
- Readability
  - Coding styles
  - Clear intent
  - Other developers (and your future self)
- Maintainability
  - Time to change/refactor
- Reusability
  - Generic
  - Level of abstraction
- Robustness
  - Withstand future uses (threading)
- Security
  - Connections
  - Storing data
- Portability
  - Time to adapt to other platforms
  - Different UI form factors
- Compatibility
  - Fit with existing/future code
- Scaleability
  - From test cases to real world uses
  - Potential future uses

Solutions?

- Write less code
- Write simpler code

How?

- Type deduction (auto, decltype)
- Threads (std::async, std::future etc.)
- Lambdas (std::function)
- Variadic templates (parameter packs)
- Range based for loops
- Braced initialisers (std::initializer_list)

Part 1: Braced Initialisers Recap

Braced Initialisers Recap

- Last years talk "Using C++11 to Improve Code Clarity: Braced Initialisers"
- Object constructor deduction
- Aggregate initialisation
- Default member initialisers

Returning Objects

```
// C++98
std::pair<Path*, float> getPathForRow (int row)
{
    PredefinedGraphics& pg = *PredefinedGraphics::getInstance();

    switch (owner.getType (row))
    {
        case vstType:       return std::pair<Path*, float> (&pg.vstPath, 0.9f);
        case vst3Type:      return std::pair<Path*, float> (&pg.vst3Path, 0.9f);
        case auType:        return std::pair<Path*, float> (&pg.auPath, 0.9f);
        case rackType:      return std::pair<Path*, float> (&pg.rackPath, 0.7f);
        case internalType:  return std::pair<Path*, float> (&pg.internalPath, 0.5f);
    }

    return std::pair<Path*, float> (nullptr, 0.0f);
}
```

Returning Objects

```
// C++11
std::pair<Path*, float> getPathForRow (int row)
{
    PredefinedGraphics& pg = *PredefinedGraphics::getInstance();

    switch (owner.getType (row))
    {
        case vstType:       return { &pg.vstPath, 0.9f };
        case vst3Type:      return { &pg.vst3Path, 0.9f };
        case auType:        return { &pg.auPath, 0.9f };
        case rackType:      return { &pg.rackPath, 0.7f };
        case internalType:  return { &pg.internalPath, 0.5f };
    }

    return {};
}
```

Constructing Objects

```
// C++98
p.addLineSegment (Line<float> (Point<float> (-0.25f, 0.5f), Point<float> (1.25f, 0.5f)), 0.1f);


// C++11
p.addLineSegment ({{ -0.25f, 0.5f }, { 1.25f, 0.5f }}, 0.1f);
```

```
// C++98
p.addTriangle (1.2f, 0.3f, 1.6f, 0.3f, 1.4f, 0.6f);


// C++11
p.addTriangle ({ 1.2f, 0.3f }, { 1.6f, 0.3f }, { 1.4f, 0.6f });
```

Initialiser Lists

```
// C++98
static int notes[] = { 35, 38, 42, 46, 51, 41 };

for (int i = 0; i < numElementsInArray (notes); ++i)
    addChannel (notes[i], MidiMessage::getRhythmInstrumentName (notes[i]));
```

```
// C++11
for (auto n : { 35, 38, 42, 46, 51, 41 })
    addChannel (n, MidiMessage::getRhythmInstrumentName (n));
```

Default Member Initialisers

```
// C++98
class MidiNote
{
public:
    MidiNote()
        : startBeat (0.0), lengthInBeats (0.0),
          noteNum (0), chan (0), velocity (0),
          colourIndex (0),
          noteID (getNextNoteID())
    {
    }

    MidiNote (double startBeat_, double lengthInBeats_,
              int noteNum_, int chan_, int velocity_)
        : startBeat (startBeat_), lengthInBeats (lengthInBeats_),
          noteNum (noteNum_), chan (chan_), velocity (velocity_),
          colourIndex (0),
          noteID (getNextNoteID())
    {
    }

    MidiNote (const ValueTree& v)
        : startBeat (v[IDs::s]), lengthInBeats (v[IDs::l]),
          noteNum (v[IDs::n]), chan (v[IDs::c]), velocity (v[IDs::v]),
          colourIndex (0),
          noteID (getNextNoteID())
    {
    }

    // accessors/mutators
    // ...

private:
    double startBeat, lengthInBeats;
    int noteNum, chan, velocity;
    int colourIndex;
    int noteID;
};
```

```
// C++11
class MidiNote
{
public:
    MidiNote() = default;

    MidiNote (int noteNum_, int chan_, int velocity_,
              double startBeat_, double lengthInBeats_)
        : startBeat (startBeat_), lengthInBeats (lengthInBeats_),
          noteNum (noteNum_), chan (chan_), velocity (velocity_)
    {
    }

    MidiNote (const ValueTree& v)
        : startBeat (v[IDs::s]), lengthInBeats (v[IDs::l]),
          noteNum (v[IDs::n]), chan (v[IDs::c]), velocity (v[IDs::v])
    {
    }

    // accessors/mutators
    // ...

private:
    double startBeat = 0.0, lengthInBeats = 0.0;
    int noteNum = 0, chan = 0, velocity = 0;
    int colourIndex = 0;
    int noteID { getNextNoteID() };
};
```

Part 2: Lambdas & std::function

Lambdas and std::function

- A lambda is a quick way to define a class and create an instance of it
- std::function is used to store, copy and invoke a callable object
- Lambdas are callable objects so can be stored in std::function
- auto lambda = [capture-list] (Args a) { functionBody; }

Lambdas as Callbacks

```
// C++98

class ObjectWithCallback
{
public:
    ObjectWithCallback() {}

    void changeObject()
    {
        listeners.call (&Listener::listenerCallback, *this);
    }

    struct Listener
    {
        virtual ~Listener() {}
        virtual void listenerCallback (ObjectWithCallback&) = 0;
    };

    void addListener (Listener* l)      { listeners.add (l); }
    void removeListener (Listener* l)   { listeners.remove (l); }

private:
    ListenerList<Listener> listeners;
};
```

```
class ListeningObject  : private ObjectWithCallback::Listener
{
public:
    ListeningObject (ObjectWithCallback& obj)
        : objectWithCallback (obj)
    {
        objectWithCallback.addListener (this);
    }

    ~ListeningObject()
    {
        objectWithCallback.removeListener (this);
    }

private:
    ObjectWithCallback& objectWithCallback;

    void listenerCallback (ObjectWithCallback&) override
    {
        // Do something...
    }
};
```

```
// Usage
ObjectWithCallback objectWithCallback;
ListeningObject listeningObject (objectWithCallback);
objectWithCallback.changeObject();
```

Lambdas as Callbacks (2)

```
// C++11

class ObjectWithLambdaCallback
{
public:
    ObjectWithLambdaCallback() {}

    void changeObject()
    {
        if (callback)
            callback();
    }

    std::function<void()> callback;
};
```

```
class LambdaListeningObject
{
public:
    LambdaListeningObject (ObjectWithLambdaCallback& obj)
        : objectWithCallback (obj)
    {
        objectWithCallback.callback = []
            {
                // Do something...
            };
    }

    ~LambdaListeningObject()
    {
        objectWithCallback.callback = nullptr;
    }

private:
    ObjectWithLambdaCallback& objectWithCallback;
};
```

```
// Usage
ObjectWithLambdaCallback objectWithLambdaCallback;
LambdaListeningObject lambdaListeningObject (objectWithLambdaCallback);
objectWithLambdaCallback.changeObject();
```

Timers

```
// C++98
struct TimerSubclass    : private Timer
{
    TimerSubclass()
    {
        startTimerHz (1);
    }

private:
    void timerCallback() override
    {
        // Do some repetitive task
    }
};
```

Multi-Timers

```
struct OldTimerExample  : public Component,
                          private MultiTimer
{
    enum
    {
        repetitiveTimer,
        deferredTimer
    };

    OldTimerExample()
    {
        startTimer (repetitiveTimer, 1000);
    }

    void triggerDeferred()
    {
        startTimer (deferredTimer, 500);
    }

    void someDeferredMethod()
    {
        // Do some stuff
        stopTimer (deferredTimer);
    }

private:
    void timerCallback (int timerID) override
    {
        switch (timerID)
        {
            case repetitiveTimer:
            {
                DBG ("repetitive callback");
                break;
            }
            case deferredTimer:
            {
                someDeferredMethod();
            }
            default:
                break;
        }
    }
};
```

Lambda Timers

```
// C++11
struct LambdaTimer  : public Timer
{
    LambdaTimer() = default;

    LambdaTimer& setCallback (std::function<void()> callback)
    {
        callbackFunction = std::move (callback);
        return *this;
    }

    void timerCallback() override
    {
        if (callbackFunction)
            callbackFunction();
    }

    std::function<void()> callbackFunction;
};
```

Using a Lambda Timer

```
// C++11
struct TimerExample
{
    TimerExample()
    {
        repetitiveTimer.setCallback ([] { DBG ("repetitive callback"); });
        repetitiveTimer.startTimerHz (1);

        deferredTimer.setCallback ([this] { someDeferredMethod(); });
    }

    void triggerDeferred()
    {
        deferredTimer.startTimer (500);
    }

private:
    void someDeferredMethod()
    {
        // Do some stuff
        deferredTimer.stopTimer();
    }

    LambdaTimer repetitiveTimer, deferredTimer;
};
```

Comparison

```
// C++98
struct OldTimerExample  : public Component,
                          private MultiTimer
{
    OldTimerExample()
    {
        startTimer (repetitiveTimer, 1000);
    }

    void triggerDeferred()
    {
        startTimer (deferredTimer, 500);
    }

private:
    enum
    {
        repetitiveTimer,
        deferredTimer
    };

    void someDeferredMethod()
    {
        // Do some stuff
        stopTimer (deferredTimer);
    }

    void timerCallback (int timerID) override
    {
        switch (timerID)
        {
            case repetitiveTimer:
            {
                DBG ("repetitive callback");
                break;
            }
            case deferredTimer:
            {
                someDeferredMethod();
            }
            default:
                break;
        }
    }
};
```

```
// C++11
struct TimerExample
{
    TimerExample()
    {
        repetitiveTimer.setCallback ([] { DBG ("repetitive callback"); });
        repetitiveTimer.startTimerHz (1);

        deferredTimer.setCallback ([this] { someDeferredMethod(); });
    }

    void triggerDeferred()
    {
        deferredTimer.startTimer (500);
    }

private:
    void someDeferredMethod()
    {
        // Do some stuff
        deferredTimer.stopTimer();
    }

    LambdaTimer repetitiveTimer, deferredTimer;
};
```

Lambda AsyncUpdater

```
// C++11
struct LambdaAsyncUpdater   : public AsyncUpdater
{
    LambdaAsyncUpdater() = default;

    ~LambdaAsyncUpdater()
    {
        cancelPendingUpdate();
    }

    LambdaAsyncUpdater& setCallback (std::function<void()> callback)
    {
        callbackFunction = std::move (callback);
        return *this;
    }

    void handleAsyncUpdate() override
    {
        if (callbackFunction)
            callbackFunction();
    }

    std::function<void()> callbackFunction;
};
```

Lambda AsyncUpdater (2)

```
// C++11
struct AsyncUpdateExample
{
    AsyncUpdateExample()
    {
        updaterOne.setCallback ([]      { DBG ("updaterOne callback"); });
        updaterTwo.setCallback ([this]  { someDeferredMethod(); });
    }

    void triggerDeferred()
    {
        updaterOne.triggerAsyncUpdate();
    }

    void someDeferredMethod()
    {
        // Do some stuff long
    }

    LambdaAsyncUpdater updaterOne, updaterTwo;
};
```

More Complex Example

```
/**
    Holds a list of function objects and enables you to call them asynchronously.

    1. Add function with an associated ID
    2. Call updateAsync with the ID
    3. When the update get triggered, any pending updates are called
    4. Useful to coalesce functionality from synchronous callbacks e.g. resizes or properties
 */
struct AsyncFunctionCaller  : private juce::AsyncUpdater
{
    AsyncFunctionCaller() = default;
    ~AsyncFunctionCaller();

    void addFunction (int functionID, std::function<void()>);
    void updateAsync (int functionID);

private:
    std::unordered_map<int, std::pair<bool, std::function<void()>>> functions;

    void handleAsyncUpdate() override;
};
```

```
struct AsyncFunctionCaller  : private juce::AsyncUpdater
{
    AsyncFunctionCaller() = default;

    ~AsyncFunctionCaller()
    {
        cancelPendingUpdate();
    }

    void addFunction (int functionID, std::function<void()> f)
    {
        functions[functionID] = { false, std::move (f) };
    }

    void updateAsync (int functionID)
    {
        auto found = functions.find (functionID);

        if (found != functions.end())
        {
            found->second.first = true;
            triggerAsyncUpdate();
        }
    }

private:
    std::unordered_map<int, std::pair<bool, std::function<void()>>> functions;

    void handleAsyncUpdate() override
    {
        auto compareAndReset = [] (bool& flag) -> bool
        {
            if (! flag)
                return false;

            flag = false;
            return true;
        };

        for (auto&& f : functions)
            if (compareAndReset (f.second.first))
                f.second.second();
    }
};
```

```
/**
    - Essentially coalesces synchronous ValueTree callbacks into other method calls
    - Almost all logic is now simply responding to property changes and triggering a flag
    - Not even any callback code
    - No member flags (e.g. bool needsToUpdateTracks)
    - No reset methods (e.g. needsToUpdateTracks = false)
*/
struct TreeWatcher  : private ValueTree::Listener
{
    enum
    {
        tracksFlag = 0,
        muteSoloFlag,
        nameFlag,
        updateLayoutFlag
    };

    TreeWatcher (TrackOwner& o, ValueTree& v) : owner (o), state (v)
    {
        state.addListener (this);

        updater.addFunction (tracksFlag,        [this] { owner.updateTracks(); } );
        updater.addFunction (muteSoloFlag,      [this] { owner.updateMuteSolo(); } );
        updater.addFunction (nameFlag,          [this] { owner.updateNames(); });
        updater.addFunction (updateLayoutFlag,  [this] { owner.updateLayout(); } );
    }

    TrackOwner& owner;
    ValueTree state;
    AsyncFunctionCaller updater;

    void childAddedOrRemoved (ValueTree&, ValueTree& c)
    {
        if (c.hasType (IDs::TRACK))
            updater.updateAsync (tracksFlag);
    }

    void valueTreePropertyChanged (ValueTree& v, const Identifier& i) override
    {
        if (v.hasType (IDs::VIEWSTATE))
        {
            if (i == IDs::viewLeft || i == IDs::viewRight)
                updater.updateAsync (updateLayoutFlag);
        }
        else if (v.hasType (IDs::TRACK))
        {
            if (i == IDs::mute || i == IDs::solo)
                updater.updateAsync (muteSoloFlag);
            else if (i == IDs::name)
                updater.updateAsync (nameFlag);
        }
    }

    void valueTreeChildAdded   (ValueTree& p, ValueTree& c) override       { childAddedOrRemoved (p, c); }
    void valueTreeChildRemoved (ValueTree& p, ValueTree& c, int) override  { childAddedOrRemoved (p, c); }
    void valueTreeParentChanged (ValueTree&) override {}

    void valueTreeChildOrderChanged (ValueTree& p, int, int n) override
    {
        if (p.getChild (n).hasType (IDs::TRACK))
            updater.updateAsync (tracksFlag);
    }
};
```

Popup Menu Callbacks

```
// Define a lambda based PopupMenuCallback
struct PopupMenuCallback    : public PopupMenu::CustomCallback
{
    PopupMenuCallback (std::function<void()> fn)
        : function (std::move (fn))
    {
        jassert (function);
    }

    bool menuItemTriggered() override
    {
        function();
        return false; // Don't pass on to general PopupMenu callback
    }

    std::function<void()> function;
};

// Helper method to add the callback to the menu
static inline void addCustomCallback (PopupMenu& m, const String& text, std::function<void()> f)
{
    PopupMenu::Item mi;
    mi.text << text;
    mi.itemID = -1;
    mi.customCallback = new PopupMenuCallback (std::move (f));
    m.addItem (mi);
}

// Use
PopupMenu m;
addCustomCallback (m, "Item 1", [] { DBG("item one clicked"); });
addCustomCallback (m, "Item 2", [] { DBG("item two clicked"); });
m.showMenuAsync ({}, nullptr);
```

ButtonCallbacks

```
struct ButtonClickCallback : public ReferenceCountedObject,
                             private Button::Listener
{
    /** Helper method to attach a callback to a button. */
    static void create (Button& b, const Identifier& callbackName, std::function<void()> callbackToUse)
    {
        auto& props = b.getProperties();
        jassert (props.indexOf (callbackName) == -1);
        props.set (callbackName, new ButtonClickCallback (b, std::move (callbackToUse)));
    }

    ~ButtonClickCallback()
    {
        button.removeListener (this);
    }

private:
    Button& button;
    std::function<void()> callback;

    ButtonClickCallback (Button& b, std::function<void()> callbackToUse)
        : button (b), callback (std::move (callbackToUse))
    {
        button.addListener (this);
    }

    void buttonClicked (Button*) override
    {
        if (callback)
            callback();
    }
};
```

ButtonCallbacks (2)

```
addAndMakeVisible (textButton);
ButtonClickCallback::create (textButton, "clickedCallback", []
                             {
                                 PopupMenu m;
                                 addCustomCallback (m, "Item 1", [] { DBG("item one clicked"); });
                                 addCustomCallback (m, "Item 2", [] { DBG("item two clicked"); });
                                 m.showMenuAsync ({}, nullptr);
                             });
```

std::function: By Reference or Value?

- Don’t pass by non-const reference (disables passing rvalues)
- Usually storing std::functions
- Not cheap to copy so avoid copying
- At minimum receive by value, move from parameter (1 move)
- Move call site lvalues in to arguments (2 moves)
- OR, provide copy and move constructor and assignment operators

Part 2: Smart Pointers, auto and some Pitfalls

Stealing by Copying

- juce::ScopedPointer<> was developed before move semantics
- This makes it possible to steal and delete a ScopedPointer's contents by type deduction

- This can happen inadvertently when exposing scoped pointers as class member variables

```
juce::ScopedPointer<Slider> slider (new Slider());

if (auto p = slider)
{
    // Here p is juce::ScopedPointer<Slider> created with the copy
    // constructor thus nulling the original
}
```

Stealing by Copying (2)

- Never expose raw juce::ScopedPointers, prefer accessors
- Make ScopedPointers const where possible to prevent stealing
- Prefer std::unique_ptr to only allow explicit stealing by invoking std::move

```
auto slider = std::make_unique<Slider>();

if (auto p = slider)
{
    // Error: Call to implicitly-deleted copy constructor
}

if (auto p = slider.get())
{
    // Correct, p is now Slider*
}

if (auto p = std::move (slider))
{
    // Explicit transfer of ownership
    // p is now std::unique_ptr<Slider>, slider is nullptr
}
```

Keeping Objects Alive through Reference Counting

- juce::ReferenceCountedObjectPtr<ObjType> will increment the reference count of the object it refers to
- This can lead to surprising reference counts in certain situations
- Consider a garbage collection scenario

ReferenceCountedObjectPtr

```
// C++98
struct GarbageCollectedObject   : juce::ReferenceCountedObject
{
    using Ptr = ReferenceCountedObjectPtr<GarbageCollectedObject>;

    GarbageCollectedObject() {}
    ~GarbageCollectedObject() {}
};


struct GarbageCollectedObjectFactory
{
    GarbageCollectedObjectFactory() {}

    ...

private:
    ReferenceCountedArray<GarbageCollectedObject> objects;

    void collect()
    {
        for (int i = objects.size(); --i >= 0;)
            if (GarbageCollectedObject* obj = objects.getUnchecked (i))
                if (obj->getReferenceCount() == 1)
                    objects.remove (i);

        // If we're the only one keeping the object alive, delete it
    }
};
```

```
// Consider a small change to the previous slider where the type of obj is now deduced
for (int i = objects.size(); --i >= 0;)
    if (auto obj = objects.getUnchecked (i))
        if (obj->getReferenceCount() == 1)
            objects.remove (i);

// Now obj is deduced to be a ReferenceCountedObjectPtr
// This increments the ref count to 2
// Thus the object is never removed from the array and deleted
// This can have wider ranging consequences


// Solution #1
// Use ReferenceCountedArray::getObjectPointer to return a raw pointer



// Solution #2
// Use std::shared_ptr to improve readability

struct GarbageCollectedObjectFactory
{
    GarbageCollectedObjectFactory() = default;

private:
    std::vector<std::shared_ptr<GarbageCollectedObject>> objects;

    void collect()
    {
        objects.erase (std::remove_if (objects.begin(),
                                       objects.end(),
                                       [] (const auto& element) { return element.unique(); }),
                       objects.end());

        // If object is unique, we're the only one referencing so remove it thus deleting it
    }
};
```

Taking a Reference to a SafePointer

- Capturing this in asynchronous functions can be dangerous
- There is the potential for the object to be deleted before the callback happens
- This can often be solved by capturing a weak reference to the object

```
// Consider this:
MessageManager::getInstance()->callAsync ([this] { someFunction(); });

// this object is deleted
// MessageManager then calls someFunction() with a dangling this pointer

// Solve by capturing a weak reference BY VALUE:
Component::SafePointer<MainContentComponent> sp (this);
MessageManager::getInstance()->callAsync ([sp]
                                          {
                                              if (sp != nullptr)
                                                  sp->someFunction();
                                          });

// The unnamed object created by the lambda creates its own member copy of sp
// This will then be nullptr if the object gets deleted
// If you capture sp by reference, the reference will be dangling once it goes out of scope
```

Part 3: std::async

std::async Basics

- std::async runs a function asynchronously
- Returns a std::future which will eventually hold the result of the function
- std::future waits in the destructor for the result*
- Can be used to parallelise non-dependant tasks

*It's actually a bit more complicated that that, it will only block if all of the following are true:

- the shared state was created by a call to std::async
- the shared state is not yet ready
- and this was the last reference to the shared state

```
// Load a file and return the buffer
static AudioBuffer<float>* loadFileInToBuffer (const void* data, size_t size)
{
    if (auto reader = std::unique_ptr<AudioFormatReader> (WavAudioFormat().createReaderFor (new
MemoryInputStream (data, size, false), true)))
    {
        std::unique_ptr<AudioBuffer<float>> buffer (new AudioBuffer<float>());
        buffer->setSize (reader->numChannels, (int) reader->lengthInSamples);
        reader->read (buffer.get(), 0, (int) reader->lengthInSamples, 0, true, true);

        return buffer.release();
    }

    return {};
}

// Load several resources: ~20 ms
void loadResourcesSerial()
{
    OwnedArray<AudioBuffer<float>> buffers;

    buffers.add (loadFileInToBuffer (hiphop01_wav, hiphop01_wavSize));
    buffers.add (loadFileInToBuffer (hiphop02_wav, hiphop02_wavSize));
    buffers.add (loadFileInToBuffer (hiphop03_wav, hiphop03_wavSize));
    buffers.add (loadFileInToBuffer (hiphop04_wav, hiphop04_wavSize));
}

// Load resources concurrently ~10 ms
void loadResourcesMultithreaded()
{
    OwnedArray<AudioBuffer<float>, CriticalSection> buffers;

    auto b1 = std::async ([&buffers] { buffers.add (loadFileInToBuffer (hiphop01_wav, hiphop01_wavSize)); });
    auto b2 = std::async ([&buffers] { buffers.add (loadFileInToBuffer (hiphop02_wav, hiphop02_wavSize)); });
    auto b3 = std::async ([&buffers] { buffers.add (loadFileInToBuffer (hiphop03_wav, hiphop03_wavSize)); });
    auto b4 = std::async ([&buffers] { buffers.add (loadFileInToBuffer (hiphop04_wav, hiphop04_wavSize)); });
    // b# deduced to std::future<void>
}
```

Generic Usage

- Can be combined with STL to make generic methods

```
void loadResourcesMultithreadedGeneric()
{
    OwnedArray<AudioBuffer<float>> buffers;
    std::vector<std::future<AudioBuffer<float>*>> futures;

    forEachAudioResource ([&futures] (auto data, auto size)
                          {
                              futures.emplace_back (std::async ([=]
                                  { return loadFileInToBuffer (data, size); })
                              );
                          });

    for (auto& f : futures)
        buffers.add (f.get());
}
```

Possible Side Effects

- Many uses of std::async/std::thread/std::future etc.
- Some will speed up code, some will slow it down
- Profile!
- Best utilised when there are:
  - No shared resources
  - Little thread contention
  - CPU is under-utilised (e.g. at start-up)
- Use C++17 Execution Policies (Parallelism TS)???

Summary

- Use Modern C++ to reduce the amount of code
  - Use braces to reduce boilerplate and default initialise objects
  - Use std::function to reduce listener boilerplate
  - Prefer composable objects (over inheritance)
  - Use lambdas to improve code locality
  - Use std smart pointers to avoid pitfalls
  - Run tasks simply in parallel using std::async
- Less code means:
  - Quicker to write
  - Quicker to read
  - There's less to reason about
  - Clearer intent
  - More robust, maintainable
  - Likely to be more optimisable

Questions?

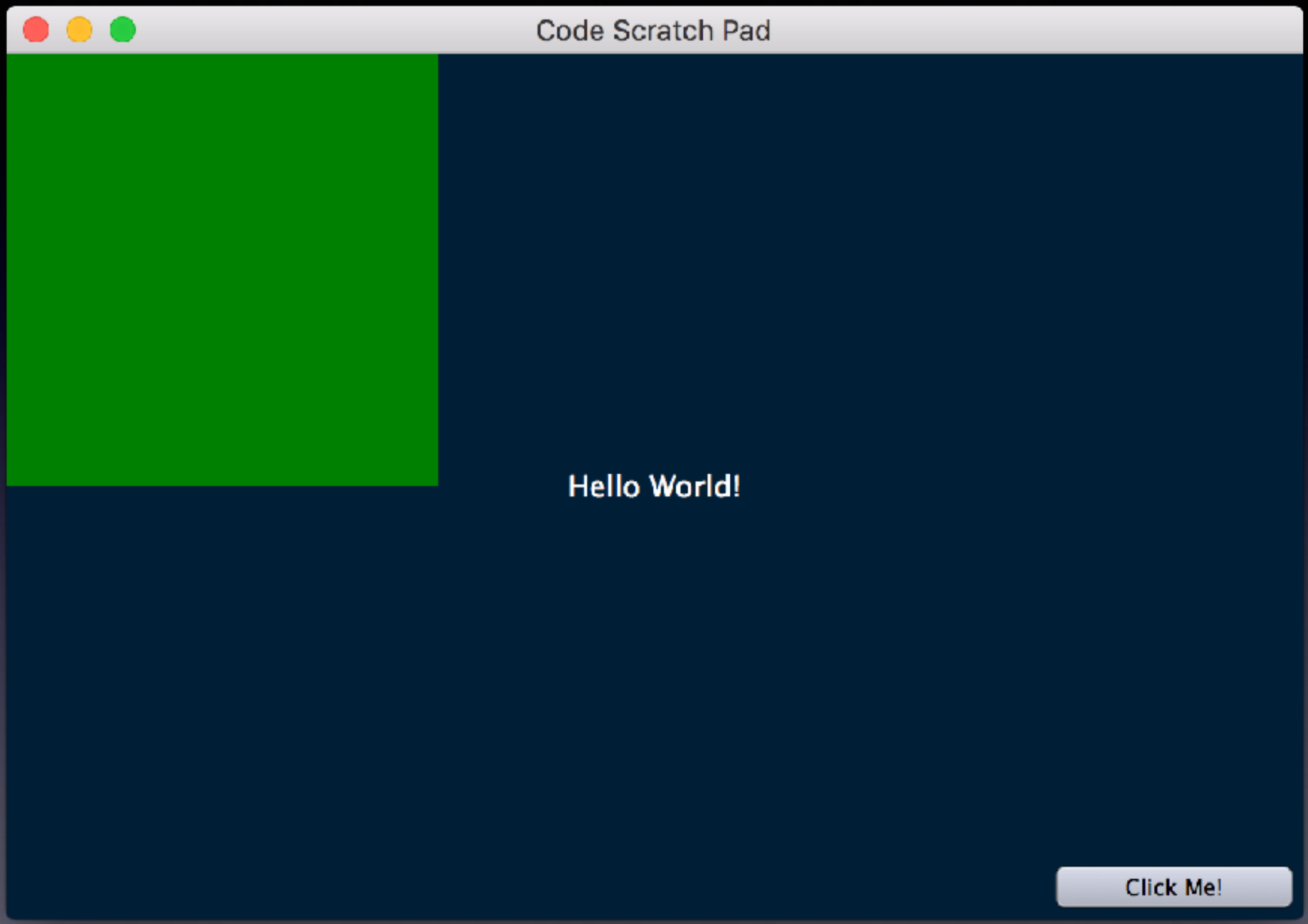
But Wait, There's More…

Composable Components

```
struct LambdaComponent  : public Component
{
   LambdaComponent() = default;

   LambdaComponent (std::function<void (Component&, Graphics&)> g)
        : paintFunction (std::move (g))
    {}

    void setCallback (std::function<void (Component&, Graphics&)> callback)
    {
        paintFunction = std::move (callback);
        repaint();
    }

    void paint (Graphics& g) override
    {
        if (paintFunction)
            paintFunction (*this, g);
    }

    std::function<void (Component&, Graphics&)> paintFunction;
};
```

Simple Example

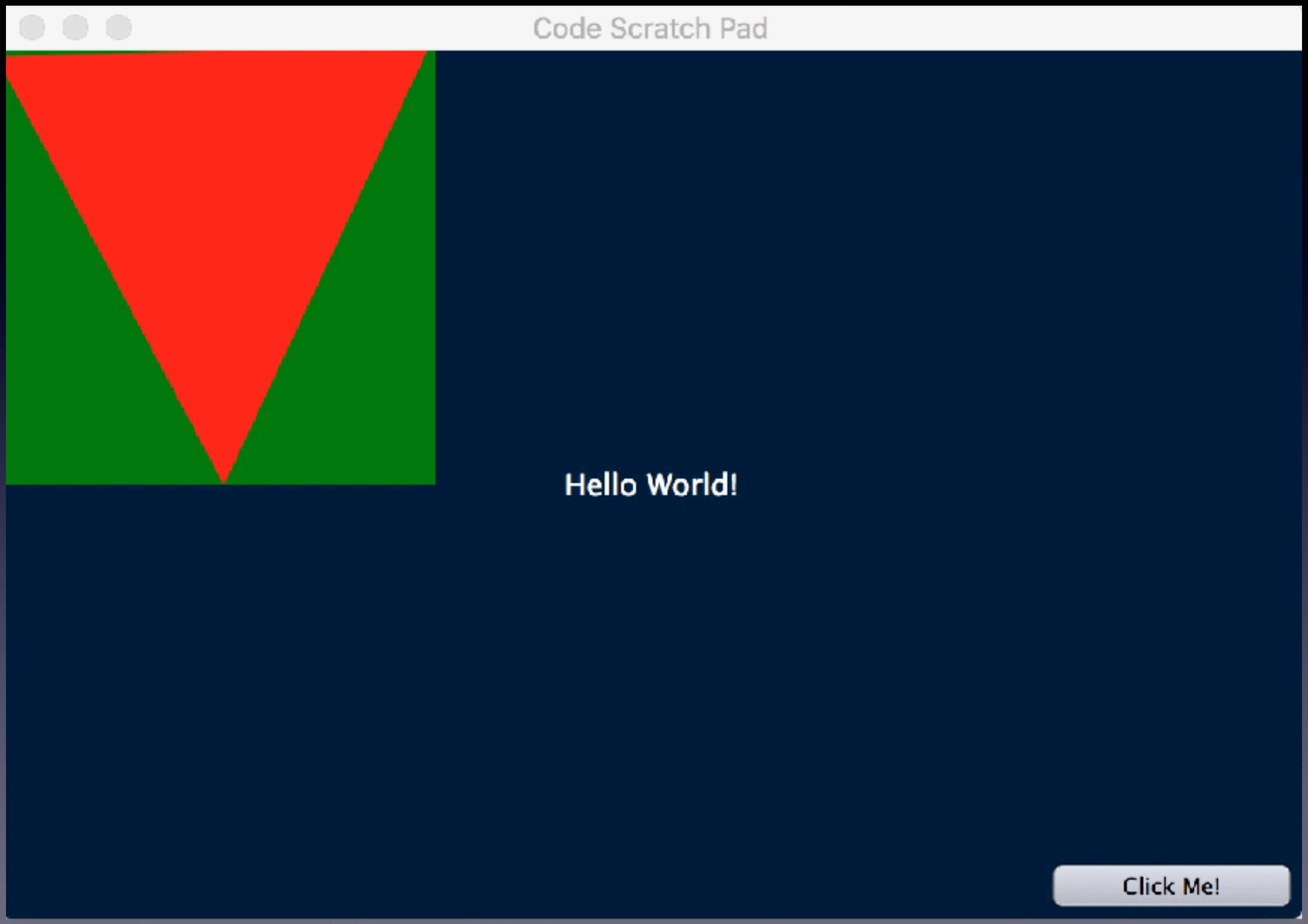
```
struct MainComponent : public Component
{
    MainComponent()
    {
        lambdaComponent.reset (new LambdaComponent ([] (Component&, Graphics& g)
                                                    {
                                                        g.fillAll (Colours::green);
                                                    }));
        addAndMakeVisible (lambdaComponent.get());
    }

    void resized() override
    {
        auto r = getLocalBounds();
        auto compBounds (r.removeFromLeft (200).removeFromTop (200));
        lambdaComponent->setBounds (getLocalBounds());
    }

    std::unique_ptr<LambdaComponent> lambdaComponent;
};
```


Code Scratch Pad
Hello World!
Click Me!

Animating a Composable Component

```
MainComponent()
{
    // Create a LambdaComponent and a timer to repaint it
    auto localLambdaComp = std::make_unique<LambdaComponent>();
    auto timer = std::make_shared<LambdaTimer> ([comp = localLambdaComp.get()] { comp->repaint(); });
    timer->startTimerHz (25);

    // Set the paint method, moving the timer's ownership and creating a counter
    localLambdaComp->setCallback ([timer = std::move (timer)] (Component& c, Graphics& g)
    {
        const double period = 3.0;
        const double prop = std::fmod (Time::getApproximateMillisecondCounter() * 0.001, period) / period;
        const double angle = (2.0 * double_Pi) * prop;

        auto r = c.getLocalBounds().toFloat();
        Path p;
        p.addTriangle (r.getBottomLeft(), r.getBottomRight(), { r.getCentreX(), 0.0f });
        const auto centre = r.getCentre();
        p.applyTransform (AffineTransform::rotation (angle, centre.x, centre.y));

        g.setColour (Colours::red);
        g.fillPath (p);
    });

    // Move ownership of the LambdaComponent
    lambdaComponent = std::move (localLambdaComp);
    addAndMakeVisible (lambdaComponent.get());
}
```

```
// Abstract complex logic into a helper function
std::unique_ptr<LambdaComponent>
createAnimatedComponent (std::function<void (Component&, Graphics&)> paintFunction)
{
    auto localLambdaComp = std::make_unique<LambdaComponent>();
    auto timer = std::make_shared<LambdaTimer> ([comp = localLambdaComp.get()] { comp->repaint(); });
    timer->startTimerHz (25);

    localLambdaComp->setCallback (
        [timer = std::move (timer), callback = std::move (paintFunction)] (Component& c, Graphics& g)
        {
            callback (c, g);
        });

    return localLambdaComp;
}

// Which leaves just the paint logic in the parent Component constructor
MainComponent()
{
    lambdaComponent = createAnimatedComponent ([] (Component& c, Graphics& g)
    {
        const double period = 3.0;
        const double prop = std::fmod (Time::getApproximateMillisecondCounter() * 0.001, period) / period;
        const double angle = (2.0 * double_Pi) * prop;

        auto r = c.getLocalBounds().toFloat();
        Path p;
        p.addTriangle (r.getBottomLeft(), r.getBottomRight(), { r.getCentreX(), 0.0f });
        const auto centre = r.getCentre();
        p.applyTransform (AffineTransform::rotation (angle, centre.x, centre.y));

        g.setColour (Colours::red);
        g.fillPath (p);
    });

    addAndMakeVisible (lambdaComponent.get());
}
```


Code Scratch Pad
Hello World!
Click Me!

More Questions?