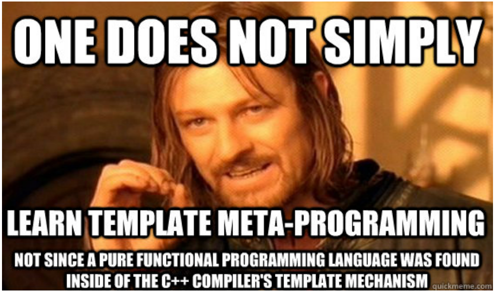
Managing Parameters in JUCE Plug-Ins

Sean Enderby

Birmingham City University

8th March 2017

Some C++11 Concepts



C++11?

- In 2011 lots of cool new features were added to the C++ standard.
- These include things like: smart pointers, multi-threading, lambda expressions and automatic type deduction.
- They make programming in C++ easier, safer and faster.
- There has been the C++14 standard since (and this year we'll get C++17), but neither are as colossally awesome as C++11.

Automatic Type Deduction

- C++11 introduced a new use for the *auto* keyword.
- When declaring a variable the type specifier can be replaced by the word *auto*.
- This tells the compiler to pick what type the variable should be.
- *auto** and *auto&* can be used to specify automatic type deduction for pointers and references.

```
// x is an int
auto x = 1;

// rx is a reference
// to an int
auto &rx = x;

// px is a pointer
// to an int
auto *px = &x;

// y is a double
auto y = 2.0;

// z is a float
float wickedFunction();
auto z = wickedFunction();
```

Range Based for Loop

- Range based for loops are a new syntax for iterating over containers. They are awesome!

```
// std::array is another C++11 toy
// C style arrays are a thing of the past!
std::array <int, 10> tenFives;

// remember to use a reference
// if you want to alter the elements
for (auto &i : tenFives)
        i = 5;

// if you are only accessing the elements
// you can use a normal variable
for (auto i : tenFives)
        std::cout << i << ",";
```

Lambda Expressions

- Lambda expressions allow you to create anonymous functions in the body of your code.
- They are really useful when you need to create callback functions for certain tasks.
- Lambda expressions take the form of three sets of brackets:
  - [*capture list*] (*parameters*) {*function body*};
- The parameters and function body are the same as for a normal function.
- The capture list declares the variables to capture from the surrounding scope.

Lambda Expressions Cont.

```
// some variables
double x = 34.2, y = 12.3;

// auto makes using lambdas so easy
// we capture x by reference and y by value
// the lambda can edit the value of x
// but only read the value of y
auto coolFunction = [&x, y] (int z)
                    {x *= y + z;};

// call the lambda like a normal function
coolFunction (13);
// x is now equal to x(y + z)
// 34.2(12.3 + 13)
```

std::function

- When you write a lambda expression you can use *auto* to have the compiler work out the type for you.
- If you need to specify the type of function yourself things can get more tricky.
- C++11 has the <*functional*> header to help in these situations.
- The *std::function* type can be used to declare variables which are functions (such as lambda expressions).
- The return type and parameter types for the function are defined using the template parameter.

std::function Cont.

- *std::function* objects can be set equal to any function of the correct type.
- Here we use a use a function and a *std::function* as an extremely convoluted way of setting a variable.

```
#include <functional> // to get access to std::function

// a really useful function
int returnX (int x) {return x;}

// make a std::function which uses returnX()
std::function <int (int)> usefulFunction = returnX;

// use the function via our std::function object
int y = usefulFunction (34);
```

std::function Cont.

- *std::function* is really useful for passing functions as arguments to another function.

```
// a function which calls the passed in function twice
void callTwice (std::function <void()> f)
{
        f();
        f();
}

// we can call it with a regular function
void printEcho() {std::cout << "Echo!\n";}
callTwice (printEcho);

// and with a lambda expression
callTwice ([](){std::cout << "Echo!\n";});
```

Parameter Management

More Complex Parameters

- So far we have used the pre made classes derived from JUCE's *AudioProcessorParameter* to manage our plug-in's parameters.
- These are great for simple parameters but are quite limited.
- We might need to do some more complex work whenever a parameter is changed, e.g.:
  - Update filter coefficients.
  - Clear a delay buffer.
  - Control multiple processing parameters with a single plug-in parameter.

More Complex Parameters Cont.

- We need some mechanism by which we can trigger the work we need to do on parameter updates.
- There are a number of ways to achieve this.
- We will cover two here:
  - Making a new child class of *AudioProcessorParameter* which allows us to register a callback function which will be called when the parameter is changed.
  - Using JUCE's *AudioProcessorValueTreeState* class.

Using Callback Functions

Parameter Class with a Callback Function

- We write a child class of *AudioProcessorParameter* which:
  - Keeps track of the parameter value and scaling (like *AudioParameterFloat* etc.).
  - Has a *std::function* member in which to store the callback function.
  - Allows the user to register a callback function.
  - Calls the callback function when the parameter value is changed.
- The plug-in can then use instances of this parameter class and use lambda expressions to register callbacks.
- See the *ParameterWithCallback* class in the Useful_Bits directory of the git repo for an implementation.

Advantages / Disadvantages

- Very flexible, can quickly create individual lambda expressions for the callback of each parameter.
- Can be set up to allow changing of callbacks if necessary.
- Makes you look like a C++11 ninja!
- More code to manage yourself.
- Can be tricky to make sure that your callback doesn't get called before the relevant parts of your class are initialised.

Using AudioProcessorValueTreeState

AudioProcessorValueTreeState

- JUCE provides the *AudioProcessorValueTreeState* class to manage the state of a plug-in.
- We create an *AudioProcessorValueTreeState* object and pass a pointer to our plug-in's processor to its constructor.
- Parameters are then added to the *AudioProcessorValueTreeState* instead of the processor.
- We can inherit the *AudioProcessorValueTreeState::Listener* class to listen for parameter changes.
- See the week 10 example code for example usage.

Advantages / Disadvantages

- Requires less code to be written.
- You can add an *UndoManager* which will automatically handle parameter change history for you.
- Very generalised (perhaps too generalised for medium complexity applications).
- Parameters have to be accessed via their ID strings which can be a little cumbersome.
- Have to use the same listener callback for every parameter or add individual listener classes for each parameter.
- Contender for the worst class name in JUCE.

Thanks For Listening!

Any Questions?

Let's Manage Some Dank Parameters!