C++11 Signals and Slots!

I've been asked multiple times how I would implement a signal / slot mechanism in modern C++. Here is the answer!

Newer post: C++11 Properties!
Older post: Color Splash!

C++11 Signals and Slots!

I've been asked multiple times how I would implement a signal / slot mechanism in modern C++. Here is the answer!

Newer post:
C++11 Properties!

Older post:
Color Splash!

Published
20 September 2015

C++11 Signals and Slots!

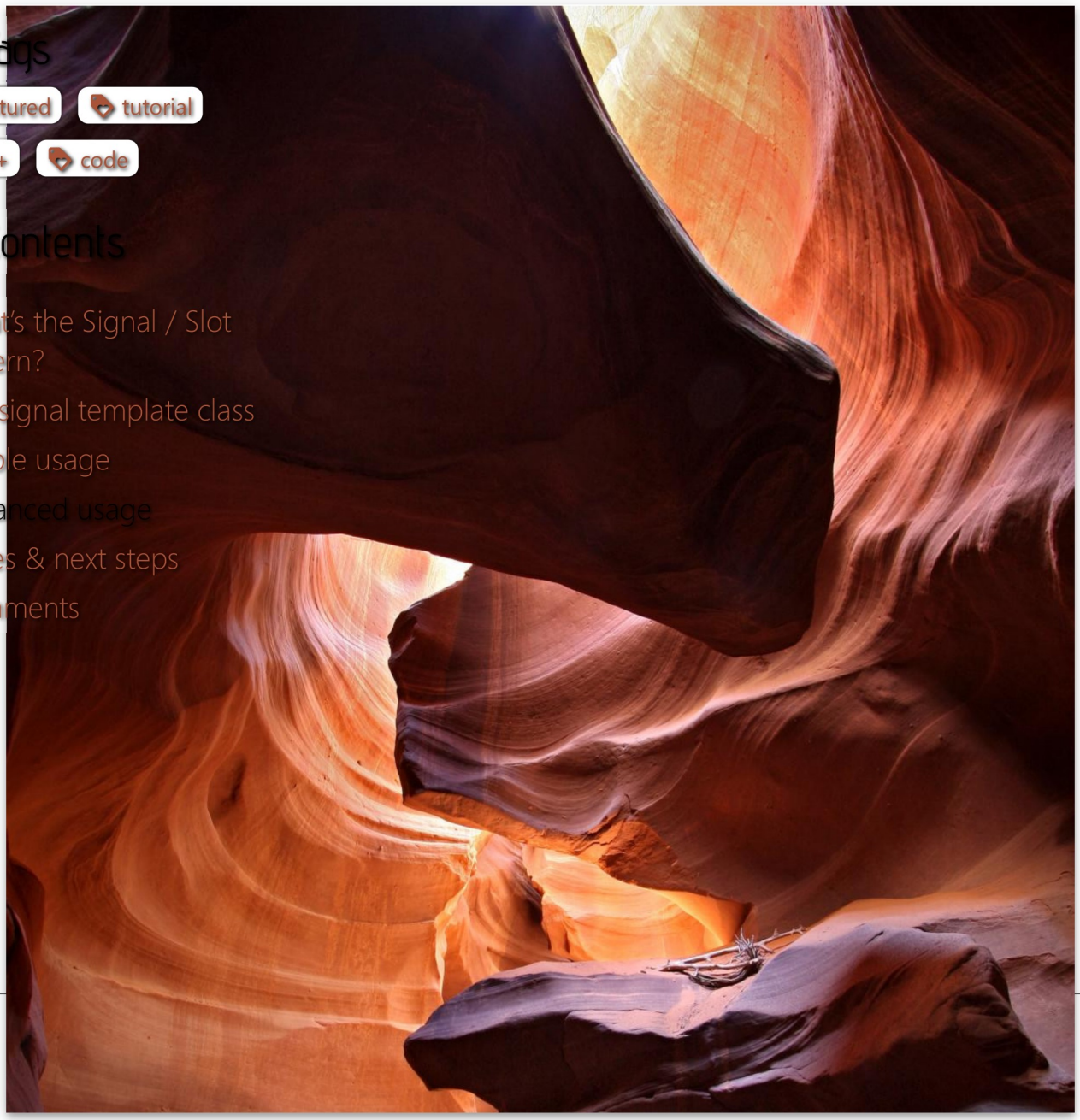
Author
Simon Schneegans

I've been asked multiple times how I would implement a signal / slot mechanism in modern C++. Here is the answer!

Tags

featured tutorial c++ code

Contents



Published
20 September 2015

Author
Simon Schneegans

Tags
featured tutorial c++ code

Contents

What's the Signal / Slot Pattern?

> [...] a language construct [...] which makes it easy to implement the Observer pattern while avoiding boilerplate code. The concept is that GUI widgets can send signals containing event information which can be received by other controls using [illegible]ns known as slots. - Wikipedia

So basically it allows for event based inter-object communication. In my opinion it's intuitive to use and produces easily readable code when used in a moderate amount. And the big plus: It can be added to your program with one simple template class!

There are many libraries around (refer to the linked Wikipedia article) implementing this pattern, but it's so easy to implement on you own that I would recommend to do this without an additional dependency. All you need is the header I posted below. And it's a good exercise.

The signal template class

Below you can find the entire class. Because this class is using variadic templates you can define signals which pass any kind of data to their slots. Basically you can create signals which allow for arbitrary slot signatures. The emit method will accept the same argument types you declared as template parameters for the Signal class. The class is documented with comments and should be quite understandable. Further below you will find two usage examples.

Collapse

```
#ifndef SIGNAL_HPP
#define SIGNAL_HPP
```

Published

20 September 2015

Author

Simon Schneegans

Tags

featured tutorial c++ code

Contents


```
#include <functional>
#include <map>

// A signal object may call multiple slots with the
// same signature. You can connect functions to the signal
// which will be called when the emit() method on the
// signal object is invoked. Any argument passed to emit()
// will be passed to the given functions.
template <typename... Args>
class Signal {

public:

  Signal() : current_id_(0) {}

  // copy creates new signal
  Signal(Signal const& other) : current_id_(0) {}

  // connects a member function to this Signal
  template <typename T>
  int connect_member(T *inst, void (T::*func)(Args...)) {
    return connect([=](Args... args) {
      (inst->*func)(args...);
    });
  }

  // connects a const member function to this Signal
  template <typename T>
  int connect_member(T *inst, void (T::*func)(Args...) const) {
    return connect([=](Args... args) {
      (inst->*func)(args...);
    });
  }

  // connects a std::function to the signal. The returned
  // value can be used to disconnect the function again
  int connect(std::function<void(Args...)> const& slot) const {
    slots_.insert(std::make_pair(++current_id_, slot));
    return current_id_;
```

Published

20 September 2015

Author

Simon Schneegans

Tags

featured tutorial c++ code

Contents


```
    }

    // disconnects a previously connected function
    void disconnect(int id) const {
      slots_.erase(id);
    }

    // disconnects all previously connected functions
    void disconnect_all() const {
      slots_.clear();
    }

    // calls all connected functions
    void emit(Args... p) {
      for (auto it : slots_) {
        it.second(p...);
      }
    }

    // assignment creates new Signal
    Signal& operator=(Signal const& other) {
      disconnect_all();
    }

  private:
```

Simple usage

The example below creates a simple signal. To this signal functions may be connected which accept a string and an integer. A lambda is connected and gets called when the emit method of the signal is called.

Collapse

```
#include "Signal.hpp"
#include <iostream>

int main() {
```

```
  // create new signal
  Signal<std::string, int> signal;

  // attach a slot
  signal.connect([](std::string arg1, int arg2) {
      std::cout << arg1 << " " << arg2 << std::endl;
  });

  signal.emit("The answer:", 42);

  return 0;
```

When you saved the Signal class as `Signal.hpp` and the above example as `main.cpp` you can compile the example with:

```
g++ --std=c++0x main.cpp
```

And if you execute the resulting application you will get the following output:

```
The answer: 42
```

Advanced usage

This example shows the usage with classes. A message gets displayed when the button is clicked. Note that neither the button knows anything of a message nor does the message know anything about a button. That's awesome! You can compile this example in the same way as the first.

Collapse

```
#include "Signal.hpp"
#include <iostream>

class Button {
 public:
  Signal<> on_click;
};
```

Published

20 September 2015

Author

Simon Schneegans

© License

public domain

Tags

featured tutorial c++ code

≡ Contents

Published

20 September 2015

Author

Simon Schneegans

Tags

featured tutorial c++ code

Contents


```
class Message {
  public:
    void display() const {
      std::cout << "Hello World!" << std::endl;
    }
};

int main() {
  Button  button;
  Message message;

  button.on_click.connect_member(&message, &Message::display);
  button.on_click.emit();

  0;
```

You may also connect member functions which take arguments (Thank you FlashingChris for pointing out how to do this without std::placeholders!). In the following example Alice and Bob may say something and the other will hear it:

Expand

```
#include "Signal.hpp"
#include <iostream>
```

Issues & next steps

There are two drawbacks in this simple implementation: It's not threadsafe and you cannot disconnect a slot from a signal from within the slot callback. Both problems are easy to solve but would make this example more complex.

Using this Signal class other patterns can be implemented easily. In a follow-up post I'll present another simple class: the Property. This will allow for a clean implementation of the observer pattern.

Have some fun coding events in C++!

- Last updated on 15 Februray 2017 -

Published

20 September 2015

Author

Simon Schneegans

Comments

blog comments powered by Disqus

© License

public domain

Tags

featured tutorial c++ code

Older post: Properties!

Color Splash! ›

Contents