# Synapse

Non-Intrusive Publish-Subscribe Library | Emil Dotchevski

# Abstract

Synapse is a non-intrusive C++ publish–subscribe library for C++11. Features:

- Any C or C++ object of any type whatsoever can be used as a signal emitter.
- A system of meta signals for interoperability with other publish–subscribe, signal programming libraries, and callback APIs. Lambda expressions / closures can be easily installed as C API callbacks!
- In a multi-thread environment signals can be emitted asynchronously and scheduled for synchronous execution when polled from other threads.
- No dependencies.

# Tutorial

## Signals, Emitters, Receivers

Signal programming libraries allow *signals* to be associated with emitter objects. Like function types, each signal has a signature. Emitting a signal is similar to function invocation, except it may call multiple functions that currently connect that particular signal from that particular emitter object. Naturally, the signature of any connected function must match the signature of the signal.

In Synapse, signals are defined as function pointer typedefs. When a signal is emitted, the value returned from any connected function is discarded, but the return type of the signal definition is still important: it is used as an identifier of the signal, a way to tell apart different signals that have otherwise identical signatures. For example, the following typedefs define two different signals, even though they both take one `int` argument:

```
struct this_signal_;
typedef this_signal_(*this_signal)(int);

struct that_signal_;
typedef that_signal_(*that_signal)(int);
```

The two defined signals are different because they use different return types (`this_signal_` vs. `that_signal_`). By convention, the return types are defined implicitly within each `typedef`. This makes the signal definitions more compact:

```
typedef struct this_signal_(*this_signal)(int);
typedef struct that_signal_(*that_signal)(int);
```

To emit a Synapse signal, we instantiate the `emit` function template with a pre-defined signal `typedef` (e.g. `this_signal`), passing the emitter object as the first argument. The rest of the arguments follow, as defined by the signal signature (in this case a single `int` argument). Note that the emitter object passed as the first argument to `emit<S>` is not forwarded implicitly to the connected functions; its only purpose is to specify the *emitter*, that is, which object is emitting the signal `S`.

As a less abstract example, let's define a type `button` that emits a Synapse signal `clicked` (which takes no arguments) when the member function `click` is called:

```
typedef struct clicked_(*clicked)();

class button {

    public:

    void click() {
        synapse::emit<clicked>(this);
    }
};
```

emit

It is possible to define the `clicked typedef` as a member of `class button`, but this coupling is usually not appropriate when using Synapse. It is better to treat signals as types with independent semantics that can be used with any appropriate object. In this case, anything clickable could emit the `clicked` signal.

Next, let's connect the signal `clicked` to the `accept` member function of a dialog box object:

```
class dialog {

    public:

    void accept();

};

....
shared_ptr<button> emitter=make_shared<button>();
shared_ptr<dialog> receiver=make_shared<dialog>();
synapse::connect<clicked>(emitter, receiver, &dialog::accept);
```

connect

Or we could use a lambda instead:

```
synapse::connect<clicked>(emitter, receiver,
    [ ](dialog * d) {
        d->accept();
    } );
```

connect

With this setup, `click()`-ing the `button` will `accept()` the `dialog`.

The `receiver` argument to `connect` is optional. If it is specified, a pointer to the receiver object is passed implicitly as the first argument to the connected function, followed by all other arguments as specified by the signal signature.

## Emitting Signals from Objects of 3rd-Party Types

The `button`/`dialog` example from the previous section could have been similarly implemented using any signal programming library, because the `button` type is specifically designed to be able to emit the `clicked` signal.

But in Synapse, *any* object whatsoever can be used as an emitter. This makes it possible to `emit` non-intrusively even if the type of the emitter object was not designed to support signals. For example, a function that processes a file can use a standard `FILE` pointer as a Synapse emitter to report on its progress:

```
typedef struct report_progress_(*report_progress)(int progress);

void process_file( FILE * f ) {

    for( int progress=0; !feof(f); ) {
        ....
        progress += fread(buf,1,nread,f);
        ....
        synapse::emit<report_progress>(f,progress);
    }

}
```

`emit`

Outside of `process_file` the `report_progress` signal can be connected to some user interface function that updates a progress bar. For example, using Synapse, we could easily connect it to a Qt `QProgressBar` object:

```
if( FILE * f=fopen("file.dat","rb") ) {
    QProgressBar pb(....);
    auto c=synapse::connect<report_progress>(f, &pb, &QProgressBar::setValue);
    process_file(f);
    fclose(f);
}
```

`connect` | `connection`

Notice that `process_file` is not coupled with `QProgressBar`: the `report_progress` signal could be connected to a different function or not connected at all, in which case the call to `emit` in `process_file` would be a no-op.

The observant reader has surely noticed that in the above example we had to capture the return value of `synapse::connect<report_progress>` in the local variable `c`, while we didn't have to do this in the previous `button`/`dialog` example. This is explained below.

# Managing Connection Lifetime

In Synapse there are two types of connection objects: `connection` and `pconnection`:

- `shared_ptr<connection>` objects are returned by overloads of `connect` which take the emitter (and, if specified, the receiver) as a raw pointer. The user is required to keep the `connection` object alive; the function passed to `connect` will be disconnected when that object expires.
- `weak_ptr<pconnection>` objects are returned by overloads of `connect` which take at least one of the emitter or the receiver as a `weak_ptr` or `shared_ptr`. The user is *not* required to keep `pconnection` objects alive; the connected function is disconnected when Synapse detects that the emitter or the receiver have expired.

The `release` function can be used to convert a `weak_ptr<pconnection>` object to a `shared_ptr<connection>` object, in case we need to disconnect the function before the receiver or the emitter have expired.

# Blocking of Signals

It is possible to temporarily block a specific signal for a specific emitter. This allows users to dynamically disable functionality implemented by emitting signals — without having to disconnect them.

Blocking affects pre-existing as well as future connections.

*Example*

```
#include <boost/synapse/connect.hpp>
#include <boost/synapse/block.hpp>
#include <string>
#include <iostream>

namespace synapse = boost::synapse;

typedef struct print_(*print)(std::string const & s);

int main() {

    int emitter;

    auto c = synapse::connect<print>(&emitter,
        [ ](std::string const & s) {
            std::cout << s;
        } );

    synapse::emit<print>(&emitter,"Hello World"); ①

    shared_ptr<synapse::blocker> b = synapse::block<print>(&emitter); ②
    synapse::emit<print>(&emitter,"no-op");

    b.reset();
    synapse::emit<print>(&emitter,"Hello World"); ③
}
```

emit | connect | connection | block | blocker

① `emit` calls the connected lambda, printing `Hello World`.

② The `print` signal will be blocked until `b` expires, therefore the call to `emit` on the next line is a no-op, even though the signal is still connected.

③ At this point `b` has expired, so the call to `emit` will call the connected lambda, printing `Hello World`, again.

# Meta Signals

Synapse features a special global emitter that emits meta signals to notify connected functions about user interactions with other signals. It can be accessed by the `meta::emitter` function.

When a signal `S` is blocked or unblocked, the meta emitter emits the `meta::blocked<S>` signal. Connecting this `meta::blocked<S>` signal allows the blocked state of the signal `S` to be automatically reflected in other systems, for example in user interface.

Similarly, when a signal `S` is connected or disconnected, the meta emitter emits the `meta::connected<S>` signal, which is useful when integrating Synapse with 3rd-party callback systems; see Synapsifying C Callbacks.

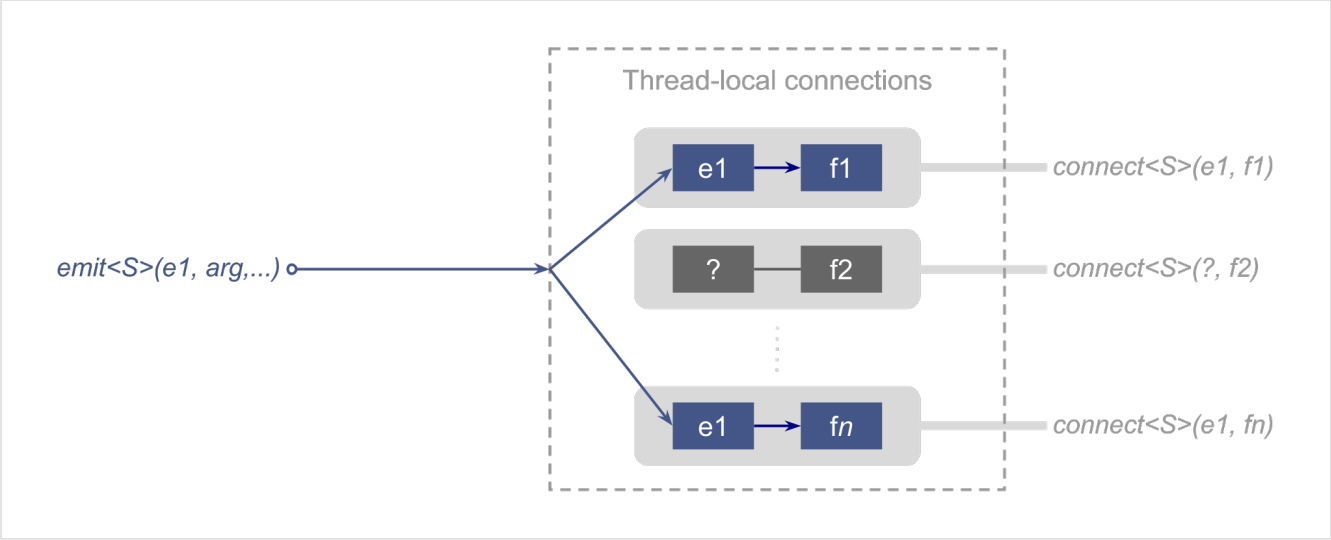
To further facilitate interoperability between Synapse and other callback APIs, `connection`/`pconnection` objects (returned by overloads of `connect`) can store arbitrary user data.

# Interthread Communication

## Emitting Signals Across Thread Boundaries

Synapse can be used to implement interthread communication using signals. The data structures created by `connect` (or `translate`) use thread-local storage, so by default calling `emit` will call only functions connected by the calling thread (and will not return until all such functions have been called in order, or one of them throws.)

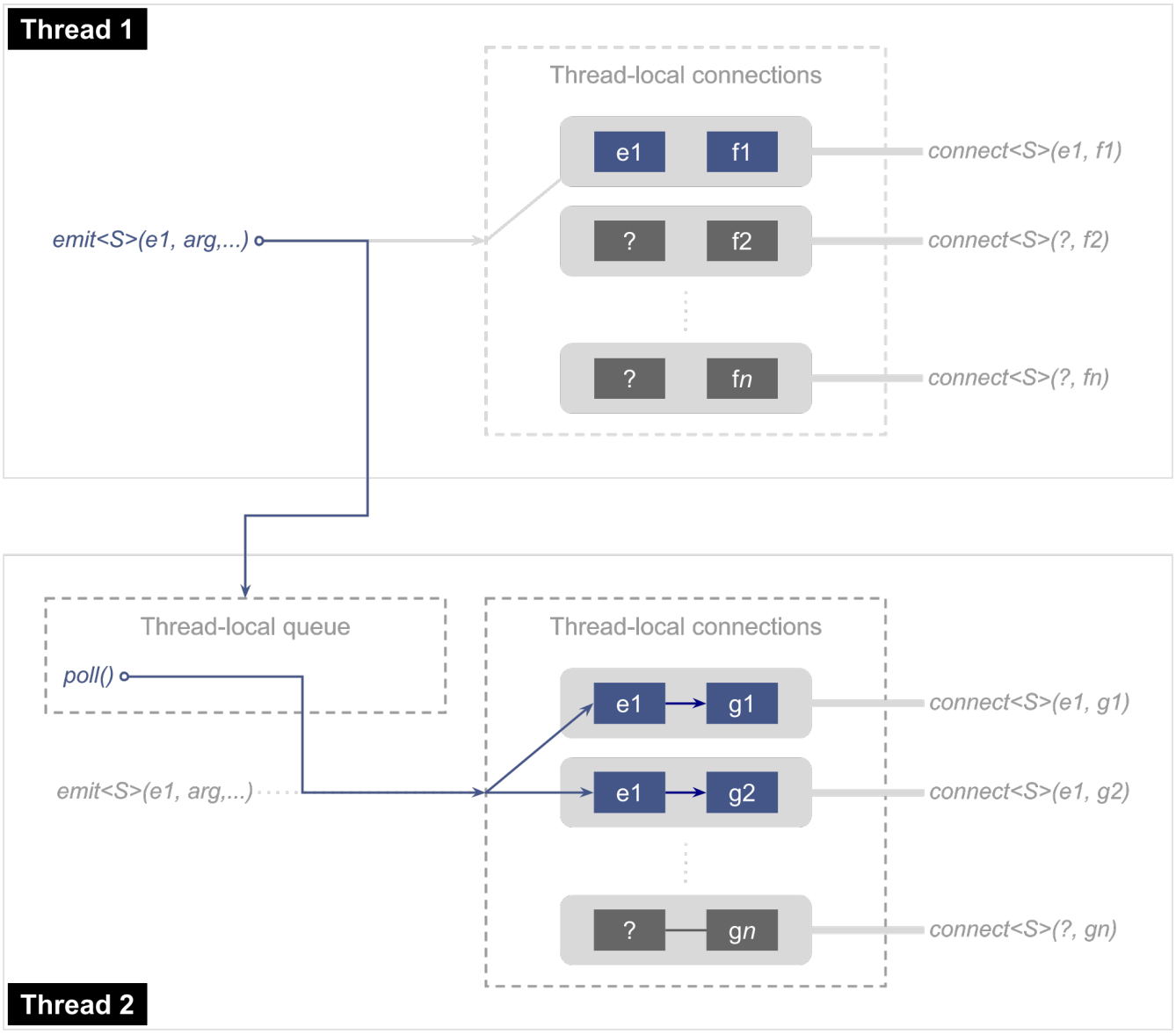
The following diagram shows the connections created (by calls to `connect<S>`) in a single thread for a given signal type `S`, each connecting an emitter to a function. When `emit<S>(e1,arg,⋯)` is called, all functions connecting the signal `S` from the given emitter `e1` are called in the order in which the connections were created:




It is also possible for any thread to request to receive all signals emitted by other threads, by creating its own `thread_local_queue` object.

In this case, *in addition* to the behavior described above, `emit<S>(e1,arg,⋯)` will capture its arguments (depending on the signature of `S`) and queue them into the `thread_local_queue` object created by any thread *other* than the calling thread. Each such thread must poll its own `thread_local_queue` regularly; this "emits" the queued objects locally and removes them from the queue (note that `poll` is not given an emitter or a signal type, it emits locally all queued objects, regardless of signal type or emitter).

This is illustrated by the following diagram:




A typical use case for this system is to update user interface objects with data generated by one or multiple worker threads: the user interface objects themselves need not be thread-safe, because they will be updated only synchronously, at the time `poll` is called.

Special care must be taken to ensure that any objects referred to by arguments passed to `emit` will remain valid at least until all other threads have polled their `thread_local_queue` objects. For example, the following code is incorrect in the presence of `thread_local_queue` objects:

```
typedef struct my_signal_(*my_signal)( int * );

void emit_my_signal( void * emitter, int x ) {
    emit<my_signal>(emitter,&x); //Undefined behavior in the presence of
thread_local_queues!
}
```

`emit`

The problem is that the address of `x` may be queued into other threads' queues, and since `x` is local to `emit_my_signal`, it may be destroyed by the time these threads call `poll`.

## Scheduling Function Calls Across Thread Boundaries

The `post` function can be used to queue into a `thread_local_queue` arbitrary functions for execution at the time `poll` is called. This feature allows critical worker threads to minimize the amount of time they consume by offloading expensive non-critical computations to another, non-critical thread. This also removes the need for synchronization, since the queued functions are executed synchronously in the thread that owns the `thread_local_queue` object.

# Synopsis

*#include <boost/synapse/emit.hpp>*

```
namespace boost { namespace synapse {

    template <class Signal, class... A>
    int emit( void const *, A... );

} }
```

emit

*#include <boost/synapse/connect.hpp>*

```
namespace boost { namespace synapse {

    class connection;

    template <class Signal, class Emitter, class F>
    shared_ptr<connection> connect( Emitter * e, F f );

    template <class Signal, class Emitter, class Receiver, class F>
    shared_ptr<connection> connect( Emitter * e, Receiver * r, F f );

    class pconnection;

    template <class Signal, class Emitter, class F>
    weak_ptr<pconnection> connect( <<Emitter>> e, F f ); ①

    template <class Signal, class Emitter, class Receiver, class F>
    weak_ptr<pconnection> connect( <<Emitter>> e, <<Receiver>> r, F f ); ②

    shared_ptr<connection const> release( weak_ptr<pconnection const> const & c );
    shared_ptr<connection> release( weak_ptr<pconnection> const & c );

    namespace meta {

        weak_ptr<void const> emitter();

        namespace connect_flags {

            unsigned const connecting=1;
            unsigned const first_for_this_emitter=2;
            unsigned const last_for_this_emitter=4;

        }

        template <class Signal>
        struct connected {
            typedef connected<Signal>(*type)( connection & c, unsigned
connect_flags cf );
        };
    }

} }
```

① Two overloads are provided: `<<Emitter>>` is `weak_ptr<Emitter> const &` or, equivalently, `shared_ptr<Emitter> const &`;

② Multple overloads are provided: `<<Emitter>>` is `Emitter *`, `weak_ptr<Emitter> const &` or, equivalently, `shared_ptr<Emitter> const &`; at least one of `e` and `r` is *not* a raw pointer.

*#include <boost/synapse/translate.hpp>*

```
namespace boost { namespace synapse {

    template <
        class OriginalSignal,  class TranslatedSignal,
        class OriginalEmitter, class TranslatedEmitter>
    shared_ptr<connection> translate( OriginalEmitter * e1, TranslatedEmitter * e2
);

    template <
        class OriginalSignal,  class TranslatedSignal,
        class OriginalEmitter, class TranslatedEmitter>
    weak_ptr<pconnection> translate( <<OriginalEmitter>> e1, <<TranslatedEmitter>>
e2 ); ①

} }
```

① Multiple overloads are provided: `<<OriginalEmitter>>` is either `OriginalEmitter *`, `weak_ptr<OriginalEmitter> const &` or, equivalently, `shared_ptr<OriginalEmitter> const &`; `<<TranslatedEmitter>>` by analogy. At least one of `e1` and `e2` is *not* a raw pointer.

translate

*#include <boost/synapse/connection.hpp>*

```
namespace boost { namespace synapse {

    class connection { //abstract base

    protected:

        connection();
        ~connection();

    public:

        template <class T>
        void set_user_data( T const & );

        template <class T>
        T * get_user_data() const;

        template <class T>
        shared_ptr<T> emitter() const;

        template <class T>
        shared_ptr<T> receiver() const;
    };

    class pconnection: protected connection { //abstract base

    protected:

        pconnection();
        ~pconnection();

    public:

        using connection::set_user_data;
        using connection::get_user_data;
        using connection::emitter;
        using connection::receiver;

    };

} }
```

connection | pconnection | set_user_data | get_user_data | emitter | receiver

*#include <boost/synapse/block.hpp>*

```
namespace boost { namespace synapse {

    class blocker;

    template <class Signal, class Emitter>
    shared_ptr<blocker> block( <<Emitter>> e ); ①

    namespace meta {

        template <class Signal>
        struct blocked {
            typedef blocked<Signal>(*type)( blocker &, bool is_blocked );
        };

    }

} }
```

① Multiple overloads are provided: `<<Emitter>>` is `Emitter *`, `weak_ptr<Emitter> const &` or, equivalently, `shared_ptr<Emitter> const &`.

`blocker` | `block` | `meta::blocked`

*#include <boost/synapse/blocker.hpp>*

```
namespace boost { namespace synapse {

    class blocker { //abstract base

    protected:

        blocker();
        ~blocker();

    public:

        template <class T>
        shared_ptr<T> emitter() const;
    };

} }
```

`blocker` | `emitter`

*#include <boost/synapse/thread_local_queue.hpp>*

```
namespace boost { namespace synapse {

    struct thread_local_queue;
    shared_ptr<thread_local_queue> create_thread_local_queue();

    int poll( thread_local_queue & q );
    int wait( thread_local_queue & q );
    void post( thread_local_queue & q, function<void()> const & f );

} }
```

thread_local_queue | create_thread_local_queue | poll | wait | post

# Reference

## emit

*#include <boost/synapse/emit.hpp>*

```
namespace boost { namespace synapse {

    template <class Signal, class... A>
    int emit( void const *, A... );

} }
```

**Effects:**

Calls all function objects that are connected to the specified `Signal` from the emitter `e`, in the order in which they were connected by `connect` or `translate`, passing the specified arguments depending on the `Signal` signature, subject to the connection lifetime/blocking restrictions.

**Returns:**

The count of the connected function objects that were called. Signals that are currently blocked are not included in the count returned by `emit`.

It is the responsibility of the caller to ensure that the emitter object `e` does not expire before `emit` returns, otherwise the behavior is undefined.

**Throws:**

Any exception thrown by one of the connected function objects, in which case the remaining function objects are not called.

**Notes:**

- Values returned by the connected function objects are discarded.
- If before `emit` returns `connect` is called on the same signal and the same emitter, any newly connected functions are not called during the same `emit`.
- If before `emit` returns a `connection` object expires, it may or may not get called during the same `emit`.
- If `e` is `0`, `emit` simply returns `0` without calling any functions. Because of this feature, if the emitter is held by a `shared_ptr` object `sp`, there is no harm in calling `emit<Signal>(sp.get(),⋯)` even if `sp` is empty. Similarly, if the caller holds a `weak_ptr` reference `wp` to an emitter object which has expired, calling `emit<Signal>(wp.lock().get(),⋯)` will simply return `0`.
- `emit` takes its arguments by value. Use `std::ref` to pass by reference (but beware of `thread_local_queue` objects).

**Thread safety:**

By default `emit` will only call functions connected from the calling thread. In addition, the signal is pushed onto any `thread_local_queue` objects created by other threads, but only if those threads

currently have at least one active connection for the specified `Signal`. In this case `emit` captures its arguments similarly to `std::bind`, and it is the responsibility of the caller to ensure that they remain valid until the posted signal is processed in all other threads, by a call to `thread_local_queue::poll` or `thread_local_queue::wait`.

---

## connect

*#include <boost/synapse/connect.hpp>*

```
namespace boost { namespace synapse {

    class connection;

    template <class Signal, class Emitter, class F>
    shared_ptr<connection> connect( Emitter * e, F f );

    template <class Signal, class Emitter, class Receiver, class F>
    shared_ptr<connection> connect( Emitter * e, Receiver * r, F f );

    class pconnection;

    template <class Signal, class Emitter, class F>
    weak_ptr<pconnection> connect( <<Emitter>> e, F f ); ①

    template <class Signal, class Emitter, class Receiver, class F>
    weak_ptr<pconnection> connect( <<Emitter>> e, <<Receiver>> r, F f ); ②

} }
```

① Two overloads are provided: `<<Emitter>>` is `weak_ptr<Emitter> const &` or, equivalently, `shared_ptr<Emitter> const &`;

② Multiple overloads are provided: `<<Emitter>>` is `Emitter *`, `weak_ptr<Emitter> const &` or, equivalently, `shared_ptr<Emitter> const &`; at least one of `e` and `r` is *not* a raw pointer.

Overloads of `connect` that return `shared_ptr<connection>` create connections whose lifetime is explicitly managed by the user. Such connections require that the caller keeps the returned `connection` object alive for as long as the connection should persist.

Overloads of `connect` that return `weak_ptr<pconnection>` take at least one of `e` and `r` as a `weak_ptr` or `shared_ptr`. They create *persistent* connections which expire automatically when either `e` or `r` expire.

**Effects:**

1. Connects the specified `Signal` from the emitter `e` to the function object `f`. The arguments of `F` must match the arguments of `Signal`, except that if `r` is specified, a pointer to the receiver object is passed as the first argument to `F`, followed by the rest of the arguments as specified by the `Signal` signature. The signal is considered disconnected when either of the following occurs:

- The returned `shared_ptr<connection>` object expires (this applies only to overloads that return `shared_ptr<connection>`);
- `e` (passed as either `weak_ptr<Emitter>` or `shared_ptr<Emitter>`) expires;
- `r` (passed as either `weak_ptr<Emitter>` or `shared_ptr<Emitter>`) expires.

The returned object does not assume ownership of `e` or `r`: passing `shared_ptr` to `connect` is equivalent to passing `weak_ptr`.

If either the emitter or the receiver, if passed as raw pointers, expire before the returned `connection` object has expired, the behavior is undefined.

2. The `meta::emitter` emits the `meta::connected<Signal>` signal:

```
namespace boost { namespace synapse {

    namespace meta {

        weak_ptr<void const> emitter();

        template <class Signal>
        struct connected {
            //unspecified
        };

        namespace connect_flags {
            unsigned const connecting=1;
            unsigned const first_for_this_emitter=2;
            unsigned const last_for_this_emitter=4;
        }

    }

} }
```

The `meta::connected<Signal>` signal is also emitted when the returned object expires. Handlers of the meta signal take a reference to the `connection` object being created or destroyed, and a second `unsigned` argument, `flags`, which indicates the circumstances under which the meta signal is emitted:

- If the `connection` object is being created, the `connecting` bit is set, otherwise it is clear;
- If this is the first `Signal` connection being created for the emitter `e`, the `first_for_this_emitter` bit is set, otherwise it is clear;
- If this is the last `Signal` connection being destroyed for the emitter `e`, the `last_for_this_emitter` bit is set, otherwise it is clear.

- Because class `connection` is the protected base of class `pconnection`, handlers of `meta::connected<Signal>` take `connection &` regardless of which `connect` overload was used.
- The `meta::connected<Signal>` signal is thread-local; it will never be queued into other threads' `thread_local_queue` objects.

The passed `connection` object can be used to access the emitter and receiver objects passed to `connect`.

**Thread safety:**

Please see `emit` and `thread_local_queue`.

---

## release

*#include <boost/synapse/connect.hpp>*

```
namespace boost { namespace synapse {

    class connection;
    class pconnection;

    shared_ptr<connection const> release( weak_ptr<pconnection const> const & c );
    shared_ptr<connection> release( weak_ptr<pconnection> const & c );

} }
```

**Effects:**

Converts a weak `pconnection` reference to shared ownership `connection` reference. The lifetime of the connection is now explicitly managed by the returned `shared_ptr` object; see `connect`.

---

## translate

*#include <boost/synapse/translate.hpp>*

```
namespace boost { namespace synapse {

    template <
        class OriginalSignal,  class TranslatedSignal,
        class OriginalEmitter, class TranslatedEmitter>
    shared_ptr<connection> translate( OriginalEmitter * e1, TranslatedEmitter * e2 );

    template <
        class OriginalSignal,  class TranslatedSignal,
        class OriginalEmitter, class TranslatedEmitter>
    weak_ptr<pconnection> translate( <<OriginalEmitter>> e1, <<TranslatedEmitter>> e2
); ①

} }
```

① Multiple overloads are provided: `<<OriginalEmitter>>` is either `OriginalEmitter *`, `weak_ptr<OriginalEmitter> const &` or, equivalently, `shared_ptr<OriginalEmitter> const &`; `<<TranslatedEmitter>>` by analogy. At least one of `e1` and `e2` is *not* a raw pointer.

**Effects:**

The `translate` function template creates a connection which causes the emitter `e2` to emit `TranslatedSignal` each time the emitter `e1` emits `OriginalSignal` (the two signals must have compatible signatures). This behavior persists until:

- the returned `connection` object expires (this applies only to the `translate` overload that takes `e1` and `e2` as raw pointers);
- `e1` (passed as either `weak_ptr` or `shared_ptr` expires;
- `e2` (passed as either `weak_ptr` or `shared_ptr` expires.

The returned `connection` object does not assume ownership of `e1` or `e2`: passing `shared_ptr` is equivalent to passing `weak_ptr`.

If either `e1` or `e2`, passed as raw pointers, expire before the returned `connection` object has expired, the behavior is undefined.

## connection

## pconnection

*#include <boost/synapse/connection.hpp>*

```
namespace boost { namespace synapse {

    class connection { //abstract base

    protected:

        connection();
        ~connection();

    public:

        template <class T>
        void set_user_data( T const & );

        template <class T>
        T * get_user_data() const;

        template <class T>
        shared_ptr<T> emitter() const;

        template <class T>
        shared_ptr<T> receiver() const;
    };

    class pconnection: protected connection { //abstract base

    protected:

        pconnection();
        ~pconnection();

    public:

        using connection::set_user_data;
        using connection::get_user_data;
        using connection::emitter;
        using connection::receiver;

    };

} }
```

`set_user_data` | `get_user_data` | `emitter` | `receiver`

Overloads of `connect` and `translate` return either `shared_ptr<connection>` or `weak_ptr<pconnection>`, depending on whether or not the emitter and the receiver are passed as raw pointers. The former is used to control the lifetime of the connection explicitly, while the latter represents persistent connections, which expire implicitly with the expiration of the emitter or the receiver object.

Before being returned to the caller, `connection` objects are passed to handlers of the `meta::connected` signal, which can use the `emitter`/`receiver` member function templates to access the emitter/receiver object passed to `connect`. The `set_user_data`/`get_user_data` member function templates can be used to store and access auxiliary information.

Use `release` to convert a non-owning `weak_ptr<pconnection>` reference to an owning `shared_ptr<connection>` reference.

---

## set_user_data

```
template <class T>
void set_user_data( T const & data );
```

**Description:**

Stores a copy of `data` into `this`. Use `get_user_data` to access it.

---

## get_user_data

```
template <class T>
T * get_user_data() const;
```

**Returns:**

- If `this` contains object of type `T` previously copied by a call to `set_user_data`, returns a pointer to that data.
- If `set_user_data` has not been called for `this`, or if the type used to instantiate the `set_user_data` function template doesn't match the type used with `get_user_data`, returns `0`.

---

## emitter

```
template <class T>
shared_ptr<T> emitter() const;
```

**Returns:**

A `shared_ptr` that points the emitter that was passed to an overload of the `connect` (or `translate`) function template that returned the `connection` object.

**Notes:**

- An empty `shared_ptr` is returned if:
  - `T` does not match the static type of the emitter passed to `connect` (or `translate`), or

    - the emitter was passed to `connect` (or `translate`) as a `shared_ptr`/`weak_ptr` and it has expired.
- If the emitter was passed to `connect` (or `translate`) as a raw pointer, the returned `shared_ptr` points that emitter but does not (can not) keep it alive.

---

### `receiver`

```
template <class T>
shared_ptr<T> receiver() const;
```

**Returns:**

A `shared_ptr` that points the receiver that was passed to an overload of the `connect` (or `translate`) function template that returned the `connection` object.

**Notes:**

- An empty `shared_ptr` is returned if:
    - No receiver object was passed to `connect` (or `translate`), or
    - `T` does not match the static type of the receiver passed to `connect` (or `translate`), or
    - the receiver was passed to `connect` (or `translate`) as a `shared_ptr`/`weak_ptr` and it has expired.
- If the receiver was passed to `connect` (or `translate`) as a raw pointer, the returned `shared_ptr` points that receiver but does not (can not) keep it alive.

---

## `block`

*#include <boost/synapse/block.hpp>*

```
namespace boost { namespace synapse {

    class blocker;

    template <class Signal, class Emitter>
    shared_ptr<blocker> block( <<Emitter>> e ); ①

} }
```

① Multiple overloads are provided: `<<Emitter>>` is either `Emitter *`, `weak_ptr<Emitter>` or, equivalently, `shared_ptr<Emitter>`.

**Effects:**

1. Blocks the specified `Signal` from the emitter `e` until the returned `blocker` object expires. While the `Signal` is blocked, calls to `emit<Signal>` for `e` are ignored and return `0`. The returned `blocker` object does not own `e` even if `block` was passed a `shared_ptr`.

2. The `meta::emitter` emits the `meta::blocked<Signal>` signal:

```
namespace boost { namespace synapse { namespace meta {

    template <class Signal>
    struct blocked {
        //unspecified
    };

} } }
```

The `meta::blocked<Signal>` signal is also emitted when the returned `blocker` object expires. Handlers of the meta signal take a reference to the `blocker` object being created or destroyed, and a second `bool` argument, `is_blocked`, which is true if the signal is becoming blocked, false if it is becoming unblocked.

- Blocking affects existing as well as future connections.
- The `meta::blocked<Signal>` signal is thread-local; it will never be queued into other threads' `thread_local_queue` objects.

If `block` is passed a raw pointer, deleting the emitter before the returned `blocker` object has expired results in undefined behavior.

## blocker

*#include <boost/synapse/blocker.hpp>*

```
namespace boost { namespace synapse {

    class blocker { //abstract base

    protected:

        blocker();
        ~blocker();

    public:

        template <class T>
        shared_ptr<T> emitter() const;
    };

} }
```

`emitter`

The `block` function returns `shared_ptr<blocker>` that is used to control the time the signal remains blocked. As well, `blocker` objects are passed to handlers of the `meta::blocked` signal, which can use the `emitter` member function template to access the emitter object passed to `block`.

---

### emitter

```
template <class T>
shared_ptr<T> emitter() const;
```

**Returns:**

A `shared_ptr` that points the emitter that was passed to `block`.

**Notes:**

- An empty shared_ptr is returned if:
  - `T` does not match the static type of the emitter passed to `block`, or
  - the emitter was passed to `block` as a `shared`/`weak_ptr` and it has expired.
- If the emitter was passed to `block` as a raw pointer, the returned `shared_ptr` points that emitter but does not (can not) keep it alive.

---

## thread_local_queue

### create_thread_local_queue

*#include <boost/synapse/thread_local_queue.hpp>*

```
namespace boost { namespace synapse {

    struct thread_local_queue;
    shared_ptr<thread_local_queue> create_thread_local_queue();

} }
```

**Returns:**

A thread-local object that can be used to queue signals emitted asynchronously from other threads. Use `poll` to emit the queued signals synchronously into the calling thread. See [Interthread communication].

While any number of threads can use this function to create their own `thread_local_queue`, it is invalid to create more than one `thread_local_queue` object per thread.

---

## poll

*#include <boost/synapse/thread_local_queue.hpp>*

```
namespace boost { namespace synapse {

    int poll( thread_local_queue & q );

} }
```

**Effects:**

Synchronously emits all signals queued asynchronously into `q` by calls to `emit` from other threads. See [Interthread communication].

**Returns:**

The total number of signals emitted.

---

## wait

*#include <boost/synapse/thread_local_queue.hpp>*

```
namespace boost { namespace synapse {

    int wait( thread_local_queue & q );

} }
```

**Effects:**

The same as `poll(q)`, except that it blocks and does not return until at least one signal was delivered.

**Returns:**

The total number of signals emitted (always greater than 0).

---

## post

*#include <boost/synapse/thread_local_queue.hpp>*

```
namespace boost { namespace synapse {

    void post( thread_local_queue & q, function<void()> const & f );

} }
```

**Effects:**

Queues `f` to be called next time `q` is polled; that is, `f` will be executed synchronously in the thread that has created `q`.

While `poll` (or `wait`) must be called from the thread that created the `thread_local_queue` object, `post` may be called from any thread.

# Programming Techniques

## Monitoring of Dynamic Systems

It is often needed to monitor the operations of a complex dynamic system beyond the facilities available in its public API. One possible option to accomplish this is to use a logging library. Synapse provides another.

Consider a dynamic object environment in a video game, where various art assets may be loaded on the fly, cached, and eventually unloaded when they are no longer needed. Such events are typically not accessible through a formal interface because they are implementation details; yet there is still a need to analyze the efficiency of the caching algorithm.

Using Synapse, we can easily define a set of signal typedefs to represent such events:

```
typedef struct object_loaded_(*object_loaded)( char const * type, char const * name );
typedef struct object_unloaded_(*object_unloaded)( char const * type, char const *
name );
typedef struct cache_miss_(*cache_miss)( char const * type, char const * name );
typedef struct cache_hit_(*cache_hit)( char const * type, char const * name );
```

As part of the implementation of the `object_cache` class, we call `emit` to signal the corresponding events as they occur:

```
void object_cache::load_object( char const * type, char const * name ) {
    ....
    //load the object
    ....
    synapse::emit<object_loaded>(this,type,name);
}
```

`emit`

During development, users of the `object_cache` class may connect the Synapse signals in order to analyze its efficiency, yet there is no need to compile calls to `emit` out of release builds; typically it is sufficient to not connect the signals. Synapse is carefully designed to support this use case: programs that do not call `connect` do not need to link with Synapse.

On the other hand, because Synapse connections are dynamic, it is possible to connect the signals only when/if we need to monitor the `object_cache` operations. For example, they can be connected only while a diagnostic information window is active.

## Synapsifying C Callbacks

It is common for C APIs to use function pointers to implement callback systems. A typical example is the `SSL_set_info_callback` function from OpenSSL:

```
void SSL_set_info_callback(SSL *ssl,
       void (*cb) (const SSL *ssl, int type, int val));
```

Once the user calls `SSL_set_info_callback`, the C function pointed to by `cb` will be called with state information for `ssl` during connection setup and use.

One difficulty with such low level C APIs is that often the user needs to pass to the callback function program-specific data. Sometimes such callback setters can be given an additional `void * user_data` argument which they retain and pass verbatim when they invoke the callback function, together with its other arguments. While this solution is rather cumbersome, it's not even supported by `SSL_set_info_callback`.

Synapse can be used with this, as well as any other C-style callback API, to install C++ function objects — including lambda functions — as callbacks. This enables additional objects needed by the callback to be captured as usual.

To do this for the `SSL_set_info_callback` function, we first define a Synapse signal with a matching signature:

```
typedef struct SSL_info_callback_(*SSL_info_callback)( const SSL *ssl, int type, int
val );
```

Next, at global scope or during initialization, we install a handler for the `meta::connected<SSL_info_callback>` signal, which the `meta::emitter` emits every time the user connects or disconnects the `SSL_info_callback` signal we defined:

```
void emit_fwd( SSL const * ssl, int type, int val );

int main( int argc, char const * argv[ ] ) {

    connect<meta::connected<SSL_info_callback> >( meta::emitter(),
        [ ]( connection & c, unsigned flags ) { ①

            if( flags & meta::connect_flags::connecting ) { ②

                if( flags & meta::connect_flags::first_for_this_emitter ) ③
                    SSL_set_info_callback(c.emitter<SSL>().get(),&emit_fwd); ④

            } else { ⑤

                if( flags & meta::connect_flags::last_for_this_emitter ) ⑥
                    if( auto ssl = c.emitter<SSL>() ) ⑦
                        SSL_set_info_callback(ssl.get(),0); ⑧

            }

        } );

}

void emit_fwd( SSL const * ssl, int type, int val ) {
    emit<SSL_info_callback>(ssl,ssl,type,val);
}
```

connect | meta::connected | emit

① This lambda function is called every time the user connects or disconnects the SSL_info_callback Synapse signal.

② The SSL_info_callback signal is being *connected*.

③ This is the *first* time the SSL_info_callback signal is being connected for a particular SSL object (emitter).

④ Call connection::emitter to get the emitter as a shared_ptr<SSL> (we know the emitter is of type SSL *), and then use the OpenSSL API to install a C callback emit_fwd, which uses emit<SSL_info_callback> to call all connected Synapse functions.

⑤ The SSL_info_callback signal is being *disconnected*.

⑥ This is the *last* SSL_info_callback connection being destroyed for a particular SSL object (emitter).

⑦ Check if the SSL object is still accessible (it may have been destroyed already).

⑧ Uninstall the emit_fwd callback.

Once the above handler for the meta::connected<SSL_info_callback> signal is installed, we can simply use connect to install a C++ lambda handler for the SSL_info_callback signal we defined:

```
shared_ptr<SSL> ssl(SSL_new(ctx),&SSL_free);

connect<SSL_info_callback>( ssl,
    [ ]( const SSL *ssl, int type, int val ) noexcept {

    } );
```

connect

## Reporting Exceptions from `noexcept` Signal Handlers

Sometimes connected functions are not permitted to throw exceptions — this is usually the case when the callback originates in C code. With Synapse, such exceptions can be reported safely to a C++ context that can store them for later processing.

First, we define a Synapse signal we will use to report exceptions:

```
typedef struct exception_caught_(*exception_caught)();
```

If we take a handler of the `SSL_info_callback` (see above) as an example, we could modify it like this:

```
shared_ptr<SSL> ssl(SSL_new(ctx),&SSL_free);

connect<SSL_info_callback>( ssl,
    [ ]( const SSL *ssl, int type, int val ) noexcept {
        try {

            //code which may throw

        } catch(...) {

            int n=synapse::emit<exception_caught>(ssl); ①
            assert(n>0); ②

        }
    } );
```

connect | emit

① `emit` the `exception_caught` Synapse signal from the `ssl` object. Handlers of this signal must be able to deal with any exception, for example they can use `std::current_exception` to capture the exception and rethrow it once control has exited the critical `noexcept` path.

② `emit` returns the number of connected functions it called, so this `assert` ensures that the exception won't get ignored.

# Using Synapse with Qt to Avoid MOCing

The signal programming API that is used in Qt is intrusive: signals must be specified in the definition of each type. For this reason, it is not possible to add signals to existing Qt types. When this is needed, users are directed to define the new signals in their own class which derives from the Qt type they wanted to add signal(s) to.

There is a special example that illustrates this approach. Unfortunately, this requires the use of the proprietary Qt Meta Object Compiler which the author finds cumbersome. Below is the same example modified to use Synapse signals, which requires no MOCing (the changes made to the original program are marked with numbers):

```
#include <boost/synapse/connect.hpp>
#define QT_NO_EMIT //Suppress the #define emit from Qt since it clashes with
synapse::emit.
#include <QtWidgets/QApplication>
#include <QtWidgets/QPushButton>

namespace synapse=boost::synapse;

class Window : public QWidget
{
public:
    explicit Window(QWidget *parent = 0);
    signals: //Not needed with Synapse but okay
    typedef struct counterReached_(*counterReached)(); ①
private slots: //<-- Not needed with Synapse but okay
    void slotButtonClicked(bool checked);
private:
    int m_counter;
    QPushButton *m_button;
    shared_ptr<synapse::connection> c_; ②
};

Window::Window(QWidget *parent) :
    QWidget(parent)
{
     // Set size of the window
     setFixedSize(100, 50);

    // Create and position the button
    m_button = new QPushButton("Hello World", this);
    m_button->setGeometry(10, 10, 80, 30);
    m_button->setCheckable(true);

    // Set the counter to 0
    m_counter = 0;

    connect(m_button,&QPushButton::clicked,
        [this]( bool checked )
```

```
        {
            slotButtonClicked(checked);
        } ); ③

    c_=synapse::connect<counterReached>(this,&QApplication::quit); ④
}

void Window::slotButtonClicked(bool checked)
{
    if (checked)
        m_button->setText("Checked");
    else
        m_button->setText("Hello World");
    m_counter ++;
    if (m_counter == 10)
        synapse::emit<counterReached>(this); ⑤
}

int main(int argc, char **argv)
{
    QApplication app (argc, argv);

    Window window;
    window.show();

    return app.exec();
}
```

connection | connect | emit

① Was: `void counterReached();`

② Needed to keep the Synapse connection alive.

③ Was: `connect(m_button, SIGNAL (clicked(bool)), this, SLOT (slotButtonClicked(bool)));`

④ Was: `connect(this, SIGNAL (counterReached()), QApplication::instance(), SLOT (quit()));`

⑤ Was: `emit counterReached();`

# Case Study: Synapsifying GLFW

Synapse integrates well with some C event handling APIs. As an example, let's consider GLFW.

GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, **receiving input and events**.

Here is the function provided by GLFW for installing a key event handler for a window:

```
GLFWkeyfun glfwSetKeyCallback( GLFWwindow *window, GLFWkeyfun cbfun );
```

where `GLFWkeyfun` is declared as:

```
typedef void (*GLFWkeyfun)( GLFWwindow * window, int key, int scancode, int action,
int mods );
```

With Synapse, we can define signals to represent this as well as all other GLFW input and window state events:

*glfw_signals.hpp:*

```
extern "C" { typedef struct GLFWwindow GLFWwindow; }

namespace glfw_signals
{
    //User input callbacks
    typedef struct Key_(*Key)( GLFWwindow *, int key, int scancode, int action, int
mods );
    typedef struct Char_(*Char)( GLFWwindow *, unsigned int codepoint );
    typedef struct CharMods_(*CharMods)( GLFWwindow *, unsigned int codepoint, int
mods );
    typedef struct CursorPos_ (*CursorPos)( GLFWwindow *, double xpos, double ypos );
    typedef struct CursorEnter_(*CursorEnter)( GLFWwindow *, int entered );
    typedef struct MouseButton_(*MouseButton)( GLFWwindow *, int button, int action,
int mods );
    typedef struct Scroll_(*Scroll)( GLFWwindow *, double xoffset, double yoffset );
    typedef struct Drop_(*Drop)( GLFWwindow *, int count, char const * * paths );

    //Window state callbacks
    typedef struct WindowClose_(*WindowClose)( GLFWwindow * );
    typedef struct WindowSize_(*WindowSize)( GLFWwindow *, int width, int height );
    typedef struct FramebufferSize_(*FramebufferSize)( GLFWwindow *, int width, int
height );
    typedef struct WindowPos_(*WindowPos)( GLFWwindow *, int xpos, int ypos );
    typedef struct WindowIconify_(*WindowIconify)( GLFWwindow *, int iconified );
    typedef struct WindowFocus_(*WindowFocus)( GLFWwindow *, int focused );
    typedef struct WindowRefresh_(*WindowRefresh)( GLFWwindow * );

    //This is emitted from the GLFWwindow object to report exceptions from connected
signal handlers
    typedef struct exception_caught_(*exception_caught)( GLFWwindow * );
}
```

Next, in a different header we install `meta::connected` signal handlers for the signals above:

*glfw_synapsify.hpp:*

```
#include "glfw_signals.hpp"
#include <boost/synapse/connect.hpp>
#include <boost/synapse/connection.hpp>
#include "GLFW/glfw3.h"

template <class Signal>
class synapsifier;

template <class R, class... A>
class synapsifier<R(*)(GLFWwindow *,A...)>
{
    typedef R(*Signal)(GLFWwindow *,A...);
```

```
    typedef void (*GLFWfun)( GLFWwindow *,A... );

    static GLFWfun prev_;

    //This is the handler that GLFW calls. It emits the corresponding Synapse
    //signal and calls the previous GLFW handler for the same event, if any.
    static void handler( GLFWwindow * w, A... a )
    {
        using namespace boost::synapse;
        try
        {
            (void) emit<Signal>(w,w,a...);
        }
        catch(...)
        {
            //We can't let exceptions propagate up into C code, so the window
            //emits the exception_caught signal, which (if exceptions are
            //expected) should be connected to capture and handle the current
            //exception.
            bool handled = emit<glfw_signals::exception_caught>(w,w)>0;
            assert(handled);
        }
        if( prev_ )
            prev_(w,a...);
    }

    public:

    explicit synapsifier( GLFWfun (*setter)(GLFWwindow *,GLFWfun) )
    {
        using namespace boost::synapse;

        //Here we connect the Synapse meta::connected<Signal> signal. This
        //signal is emitted by the meta::emitter() when the Signal is being
        //connected (the user calls synapse::connect<Signal>) or disconnected
        //(when the connection expires). The emitter pointer passed to connect
        //(which in this case is of type GLFWwindow) is stored in the
        //synapse::connection object passed to the lambda below, and can be
        //accessed by the connection::emitter member function template.
        connect<meta::connected<Signal> >( meta::emitter(),
            [setter]( connection & c, unsigned flags )
            {
                if( flags&meta::connect_flags::connecting )
                {
                    //When the Signal is being connected for the first time,
                    //use the GLFW API to install our handler.
                    if( flags&meta::connect_flags::first_for_this_emitter )
                        prev_=setter(c.emitter<GLFWwindow>().get(),&handler);
                }
                else
                {
```

```
                    //When the last Signal connection expires, use the GLFW API
                    //to uninstall our handler and restore the previous handler.
                    if( flags&meta::connect_flags::last_for_this_emitter )
                    {
                        GLFWfun p=setter(c.emitter<GLFWwindow>().get(),prev_);
                        assert(p==&handler);
                    }
                }
            } );
    }
};

template <class R, class... A>
typename synapsifier<R(*)(GLFWwindow *,A...)>::GLFWfun synapsifier<R(*)(GLFWwindow *,
A...)>::prev_;

//Install all the synapse::meta::connected<....> handlers
synapsifier<glfw_signals::WindowClose> s1(&glfwSetWindowCloseCallback);
synapsifier<glfw_signals::WindowSize> s2(&glfwSetWindowSizeCallback);
synapsifier<glfw_signals::FramebufferSize> s3(&glfwSetFramebufferSizeCallback);
synapsifier<glfw_signals::WindowPos> s4(&glfwSetWindowPosCallback);
synapsifier<glfw_signals::WindowIconify> s5(&glfwSetWindowIconifyCallback);
synapsifier<glfw_signals::WindowFocus> s6(&glfwSetWindowFocusCallback);
synapsifier<glfw_signals::WindowRefresh> s7(&glfwSetWindowRefreshCallback);
synapsifier<glfw_signals::Key> s8(&glfwSetKeyCallback);
synapsifier<glfw_signals::Char> s9(&glfwSetCharCallback);
synapsifier<glfw_signals::CharMods> s10(&glfwSetCharModsCallback);
synapsifier<glfw_signals::CursorPos> s11(&glfwSetCursorPosCallback);
synapsifier<glfw_signals::CursorEnter> s12(&glfwSetCursorEnterCallback);
synapsifier<glfw_signals::MouseButton> s13(&glfwSetMouseButtonCallback);
synapsifier<glfw_signals::Scroll> s14(&glfwSetScrollCallback);
synapsifier<glfw_signals::Drop> s15(&glfwSetDropCallback);
```

emit | connect | meta::connected

The above glfw_synapsify.hpp should be included in exactly one compilation unit of a GLFW program, for example the main compilation unit. This will automatically install all meta::connected signal handlers.

With this, we simply use connect to hook up any GLFWwindow event. For example, if we have a GLFWwindow pointer w, we can install a key event handler like so:

```
auto c = synapse::connect<glfw_signals::key>(w,
    [ ]( GLFWwindow * w, int key, int scancode, int action, int mods )
    {
        ....
    } );
```

connect

Finally, this is the example from the GLFW Getting started page, modified to use the "synapsify" framework above (changes to the original example are marked with numbers):

```
//========================================================================
// Simple GLFW example
// Copyright (c) Camilla Löwy <elmindreda@glfw.org>
//
// This software is provided 'as-is', without any express or implied
// warranty. In no event will the authors be held liable for any damages
// arising from the use of this software.
//
// Permission is granted to anyone to use this software for any purpose,
// including commercial applications, and to alter it and redistribute it
// freely, subject to the following restrictions:
//
// 1. The origin of this software must not be misrepresented; you must not
//    claim that you wrote the original software. If you use this software
//    in a product, an acknowledgment in the product documentation would
//    be appreciated but is not required.
//
// 2. Altered source versions must be plainly marked as such, and must not
//    be misrepresented as being the original software.
//
// 3. This notice may not be removed or altered from any source
//    distribution.
//
//========================================================================

#include "glfw_synapsify.hpp" ①
namespace synapse = boost::synapse;

#include <glad/glad.h>
#include <GLFW/glfw3.h>

#include "linmath.h"

#include <stdlib.h>
#include <stdio.h>

static const struct
{
    float x, y;
    float r, g, b;
} vertices[3] =
{
    { -0.6f, -0.4f, 1.f, 0.f, 0.f },
    {  0.6f, -0.4f, 0.f, 1.f, 0.f },
    {   0.f,  0.6f, 0.f, 0.f, 1.f }
};
```

```
static const char* vertex_shader_text =
"#version 110\n"
"uniform mat4 MVP;\n"
"attribute vec3 vCol;\n"
"attribute vec2 vPos;\n"
"varying vec3 color;\n"
"void main()\n"
"{\n"
"    gl_Position = MVP * vec4(vPos, 0.0, 1.0);\n"
"    color = vCol;\n"
"}\n";

static const char* fragment_shader_text =
"#version 110\n"
"varying vec3 color;\n"
"void main()\n"
"{\n"
"    gl_FragColor = vec4(color, 1.0);\n"
"}\n";

static void error_callback(int error, const char* description)
{
    fprintf(stderr, "Error: %s\n", description);
}

/* ②
static void key_callback(GLFWwindow* window, int key, int scancode, int action, int
mods)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GLFW_TRUE);
}
*/

int main(void)
{
    GLFWwindow* window;
    GLuint vertex_buffer, vertex_shader, fragment_shader, program;
    GLint mvp_location, vpos_location, vcol_location;

    glfwSetErrorCallback(error_callback);

    if (!glfwInit())
        exit(EXIT_FAILURE);

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);

    window = glfwCreateWindow(640, 480, "Simple example", NULL, NULL);
    if (!window)
    {
```

```
        glfwTerminate();
        exit(EXIT_FAILURE);
    }

    //glfwSetKeyCallback(window, key_callback); ②

    auto connected = synapse::connect<glfw_signals::Key>(window, ③
        [ ]( GLFWwindow * window, int key, int /*scancode*/, int action, int /*mods*/
)
        {
            if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
                glfwSetWindowShouldClose(window, GLFW_TRUE);
        } );

    glfwMakeContextCurrent(window);
    gladLoadGLLoader((GLADloadproc) glfwGetProcAddress);
    glfwSwapInterval(1);

    // NOTE: OpenGL error checks have been omitted for brevity

    glGenBuffers(1, &vertex_buffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    vertex_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex_shader, 1, &vertex_shader_text, NULL);
    glCompileShader(vertex_shader);

    fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment_shader, 1, &fragment_shader_text, NULL);
    glCompileShader(fragment_shader);

    program = glCreateProgram();
    glAttachShader(program, vertex_shader);
    glAttachShader(program, fragment_shader);
    glLinkProgram(program);

    mvp_location = glGetUniformLocation(program, "MVP");
    vpos_location = glGetAttribLocation(program, "vPos");
    vcol_location = glGetAttribLocation(program, "vCol");

    glEnableVertexAttribArray(vpos_location);
    glVertexAttribPointer(vpos_location, 2, GL_FLOAT, GL_FALSE,
                          sizeof(vertices[0]), (void*) 0);
    glEnableVertexAttribArray(vcol_location);
    glVertexAttribPointer(vcol_location, 3, GL_FLOAT, GL_FALSE,
                          sizeof(vertices[0]), (void*) (sizeof(float) * 2));

    while (!glfwWindowShouldClose(window))
    {
        float ratio;
```

```
        int width, height;
        mat4x4 m, p, mvp;

        glfwGetFramebufferSize(window, &width, &height);
        ratio = width / (float) height;

        glViewport(0, 0, width, height);
        glClear(GL_COLOR_BUFFER_BIT);

        mat4x4_identity(m);
        mat4x4_rotate_Z(m, m, (float) glfwGetTime());
        mat4x4_ortho(p, -ratio, ratio, -1.f, 1.f, 1.f, -1.f);
        mat4x4_mul(mvp, p, m);

        glUseProgram(program);
        glUniformMatrix4fv(mvp_location, 1, GL_FALSE, (const GLfloat*) mvp);
        glDrawArrays(GL_TRIANGLES, 0, 3);

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

connect | connection

① Automatically install `meta::connected` signal handlers to synapsify GLFW.

② Commented-out parts of the original example, and...

③ ...the C++ lambda function connected to the `Key` Synapse signal from `glfw_signals.hpp` to handle the `Esc` key.

# Alternatives to Synapse

The unique design feature of Synapse is that it is non-intrusive with respect to the emitter object type. While other libraries provide users with types that can emit signals, Synapse is able to emit any signal from any object of any type whatsoever.

For a list of signal programming libraries, see this Wikipedia page.

## Comparison to Boost Signals2

Because Synapse is formatted for Boost review, people often ask what are the differences between Synapse and Boost Signals2.

*Table 1. Comparison between Signals2 and Synapse*

| | **Signals2** | **Synapse** |
|---|---|---|
| What is a signal? | An object of type `signal<T>`, which maintains a list of connected functions, and is callable like `std::function<T>` is. | A type, a C function pointer `typedef`. |
| How does emitting a signal work? | Invoking `s(⋯)`, where `s` is of type `signal<T>`, calls all connected functions. | Invoking `emit<S>(e,⋯)` calls all functions connecting the signal `S` for the object `e`. |
| What objects can emit signals? | Only instances of the `signal<T>` class template. | Any object of any type whatsoever: the `emit<S>(e)` function template takes `e` as a `void` pointer. |
| How does connecting a signal work? | Calling `s.connect(f)`, where `s` is of type `signal<T>`, connects the function `f`. | Calling `connect<S>(e,f)` connects the signal `S` from the emitter object `e` to the function `f`. |
| Support for meta signals? | No (not possible, a signal is an object). | Yes. Connecting a signal of type `S` emits the signal `meta::connected<S>` from the `meta::emitter`. |
| Integration with C-style callback APIs? | No (not possible). | Yes, through meta signals, see Synapsifying C Callbacks. |
| Multi-threading support? | Yes, the connection list maintained in each signal object is thread-safe. | Yes, the connection lists are thread-local, and signals are transported to other threads using `thread_local_queue` objects. |

| | Signals2 | Synapse |
|---|---|---|
| Can connected functions return values? | Yes, there is an elaborate system for dealing with multiple returns when a signal is invoked. | No, but of course it's possible to pass an argument by reference or a custom object to collect and/or accumulate the results if needed. |

# Macros and Configuration

## BOOST_SYNAPSE_ASSERT

All assertions in Synapse use this macro. If it is not defined, Synapse header files `#define` it either as `BOOST_ASSERT` or `assert`, depending on whether `BOOST_SYNAPSE_USE_BOOST` is defined.

# Distribution

Synapse is distributed under the Boost Software License, Version 1.0.

The source code is available on GitHub.

Synapse is not part of Boost.

# Support

The following support options are available:

- cpplang on Slack (use the `#boost` channel)
- Boost Users Mailing List
- Boost Developers Mailing List

# Portability

Synapse requires a C++ compiler that supports at least C++11.

# Building

There are two build scripts provided, one for Meson Build, the other for Boost Build.

To build the source code:

- Using Meson Build, from the Synapse root directory, execute e.g.:

```
meson bld/debug
cd bld/debug
ninja
```

- Using Boost Build, make sure Synapse is cloned under the `<boost-root>/libs` directory. From the Synapse root directory, execute:

```
../../b2 build
```

To run the unit tests:

- Using Meson Build, from the Synapse root directory, execute e.g.:

```
meson bld/debug
cd bld/debug
meson test
```

- Using Boost Build, make sure Synapse is cloned under the `<boost-root>/libs` directory. From the Synapse root directory, execute:

```
../../b2 build/test
```

`boost/synapse/emit.hpp` is a header-only component: if a program only emits signals, but does not connect signals, it needs not link with Synapse. This enables libraries to emit signals without forcing a link dependency on Synapse on programs that do not use it.

# Q&A

1. *Is there a way to stop the emit loop before all connected functions have been called?*

   No, except by throwing an exception.

2. *I am concerned about code size, does Synapse use a lot of templates?*

   Yes, there are templates instantiated for each signal type. This is done so that the dispatch by signal type occurs at compile-time, leaving only emitter dispatch at run-time. However, static types are erased as soon as possible, so template bloat is kept to a minimum.

# Acknowledgements

Special thanks to Peter Dimov for his valuable feedback on the Synapse design and for coming up with the perfect name for this library.

---



Documentation rendered by Asciidoctor with these customizations.