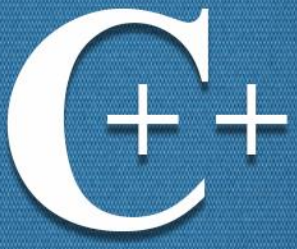

Signals & Slots

LIBRARY

vdksoft@gmail.com

Table of Contents

Introduction

Signals and slots is a mechanism used for communication between different parts of some application. The key concept is allowing objects of any kind to communicate with one another while keeping them as much decoupled as possible. Initially this concept was introduced in **Qt framework** and since then it has been proven to be very convenient, flexible and reliable mechanism for inter-object communication.

Signals are emitted by an object when its internal state has changed in some way that might be interesting to other objects. Slots are called when signals connected to them are emitted. Slots are executed one by one, in the order they have been connected to the signal. Execution of the code following the signal emission will happen once all slots have returned. Slots can be used to receive signals, but they are also normal C++ functions and can be called directly.

The signals and slots mechanism is type safe. The signal type must match the signature of the receiving slot and the compiler helps to detect type mismatches generating compile-time errors. Signals and slots can take any number of arguments of any types.

Each signal can be connected to any number of slots and each slot can be connected to any number of signals. It is even possible to connect a signal directly to another signal. In this case, emission of the first signal will emit the second signal, passing arguments from the first one to the second.

The signals and slots mechanism is especially convenient for Graphical User Interface programming. When one widget changes its state it is often expected that other widgets will be notified. If a user clicks some button, an appropriate function or even set of functions should be called. A GUI widget may send signals containing event information, which can be received by other widgets / controls using slots.

Signals and slots are loosely coupled. An object, which contains and emits a signal neither knows nor cares which slots receive the signal it emits and moreover whether anything receives the signal at all. On the other hand, an object, which has slots does not know if it has any signals connected to it. Such decoupling is good for information encapsulation and ensures that the object can be used as an independent software component. Signals and slots concept enables a powerful and flexible component programming mechanism that makes code easy to design and reuse.

Overview

`vdk-signals` is a one-header library, which implements a type-safe, thread-safe signals-slots system in C++. The library is implemented entirely in pure standard C++, and does not require any special tools (like Meta Object Compiler in **Qt**), third party libraries or non-standard extensions.

The core of this library is **`vdk::signal`** template class. This class exposes all public methods and encapsulates a set of internal classes. Each of these classes is responsible for some particular task such as memory management, type erasures, slot invoking, etc.

An instance of **`vdk::signal`** contains a list of connections (connected slots). When a particular event occurs the signal is emitted and all connected slots are executed one after another. The exact execution order is not specified, however in general it is the order in which the slots have been connected to the signal.

Slots are normal C++ functions, which are called in response to a particular signal emission. Any callable target can be connected to some signal as a slot. It may be a free (global) function, static member function of some class (static method), member function of some class (method) with any qualifiers (like **`const`**, **`volatile`**), function object (functor), lambda or even another signal.

To connect some slot to a signal, user passes function pointer | object pointer and method pointer | functor pointer to the signal's "**connect()**" method. To disconnect some slot from a signal, exactly the same should be passed to the signal's "**disconnect()**" method. Both methods return **true** or **false** to indicate whether the connection / disconnection was successful. Connection may not be successful if the slot is already connected to the signal. Disconnection may not be successful if the slot is not connected to the signal. All slots connected to a signal can be disconnected in one statement using "**disconnect_all()**" method.

A signal can be emitted through "**emit()**" method or "**operator()**". If slots connected to the signal expect an argument list, all arguments must be passed into "**emit()**" / "**operator()**" method. **vdk::signal** is optimized for signal emission. It is assumed that signal emission happens much more often than slot connection / disconnection, and hence should be as fast as possible. Everything inside **vdk::signal** works with respect to this assumption. One of the key goals of this library is making signal emission performance appropriate to be used in time-critical applications.

vdk-signals library supports auto-disconnection feature. A signal object can track connected slots through **std::shared_ptr** / **std::weak_ptr** and automatically disconnect destroyed slots, once such slots are detected during signal emission. **Important:** Only slots that require an object to be called on, can be trackable. Such a slot should be connected through **std::shared_ptr** instance, instead of raw object pointer. Internally, **vdk::signal** obtains **std::weak_ptr** from **std::shared_ptr**, and stores it. **Note:** it is important to mention that execution of trackable slots is slower than slots with explicit disconnection. This is the unavoidable overhead required to check whether the slot still exists. There is one special case in tracking objects – when object tracks itself. Several proposals will be given in Design Rationale section. **vdk-signals** also contains two helper functions for this purpose: **vdk::create_shared_object()** and **vdk::allocate_shared_object()**.

Another key goal of this library is memory efficiency. In its constructor **vdk::signal** expects to receive an optional argument – the capacity value that represents how many slots will be connected to the signal. It is used as a hint to the underlying memory manager. Using provided capacity, signal allocates memory sufficient to store entire list of connected slots as one contiguous memory block. It increases memory efficiency and reduces fragmentation. If provided capacity is not enough **vdk::signal** is still able to work, but in this case it will have to allocate additional memory block (with the size equal to capacity) during its lifetime, what may decrease efficiency of memory usage.

Signals can be blocked to temporarily prevent them from emission. If a signal instance is blocked it does not execute any connected slots. To block or unblock a signal method "**block()**" is used. It expects **true** or **false**, meaning that it should be blocked or unblocked. It is also possible to check if a signal is blocked using "**blocked()**" method, that returns **true** or **false**.

Sometimes it is useful to know if some particular slot is connected to the signal. For this purpose overloaded method "**connected()**" is used. It returns **true**, if the slot is connected to the given signal and **false** otherwise.

There are also some helper methods in the library. It is possible to check if something connected to a signal at all. It can be done with "**empty()**" method. It returns **true** if the list of connected slots inside **vdk::signal** instance is empty. Method "**size()**" returns how many slots are currently connected to the given signal.

In some cases, when user is absolutely sure that the library will be used in single-threaded environment only, one could activate special library mode to enable signals optimized for single-threaded usage. For that purpose, "**VDK_SIGNALS_LITE**" macro should be defined ***before*** including the library header. This macro switches the library mode making it single-threaded, so that user can avoid any unnecessary overhead introduced by thread-safety and benefit from keeping every bit of performance. This single-threaded mode is also optimized for memory usage efficiency. Don't pay for what you don't use!

Features

- Single-header implementation. Does not require any special compilation. Just put it in the project, #include and use;
- Signal class is solid and opaque. There is no mess with helper classes or functions. Signal can be treated as built-in type;
- Simple to use, with intuitive and clean syntax, but without any limitations of necessary functionality in the same time;
- Thread-safe signals with excellent concurrency capabilities. C++11 atomics ensure fast work even in high-load situations;
- Type-safe signals with compile-time checks. The compiler detects type mismatches generating compile-time errors;
- Efficient memory usage. Memory can be allocated up-front in single memory block. Eliminates memory fragmentation;
- Library relies on standard modern C++ only, without any non-standard extensions, meta-object compilers or other tools;
- Signals support auto-disconnection feature. Expired / destroyed slots are detected and disconnected automatically;
- Signals are able to connect any type of callable targets, including methods with qualifiers, lambdas and other signals;
- Special single-threaded mode is available. Applications that work in single thread does not lose any bit of performance;
- Exception safety. No thrown exception can leave a signal object in undetermined state. All exceptions are documented.

Tutorial

This tutorial is separated into different parts, each of those covers some particular concept.

Simple example: "Hello, World!"

This very simple example shows just the basic idea of signals-slots concept. Suppose, we have a function that writes "Hello, World!".

```
void hello_world()
{
	std::cout << "Hello, World!" << std::endl;
}
```

Let's create a signal compatible with our function that takes no arguments and returns **void**.

```
vdk::signal<void()> sig; // The same as vdk::signal<void(void)>
```

Now we connect created signal to the function using the **connect()** method.

```
sig.connect(&hello_world);
```

Finally, we emit the signal to call the slot.

```
sig.emit();
```

```
Output: Hello, World!
```

Connecting multiple slots

Multiple slots can be connected to the same signal. To show this, we need three functions with the same signature, to be able to connect them to the same signal of the appropriate type.

```
void first_function()
{
	std::cout << "First function" << std::endl;
}

void second_function()
{
	std::cout << "Second function" << std::endl;
}

void third_function()
```

```
{
	std::cout << "Third function" << std::endl;
}
```

Just like in the previous example, we can create a signal that takes no arguments and returns **void**. This time, we connect all functions as slots to the same signal. Then we emit the signal and all slots will be called.

```
vdk::signal<void(void)> sig;

sig.connect(&first_function);
sig.connect(&second_function);
sig.connect(&third_function);

sig.emit();
```

```
Output:  First function
         Second function
         Third function
```

The exact order of slot executions is not specified. However, in general they are executed in the same order they were connected to the signal.

Slot arguments

Signals propagate arguments of **emit()** method to each of the slots they call. A signal can have any number of arguments of any types. As an example, we will create three functions with different arguments and three signals of different types for them.

```
int function_1(int arg)
{
	std::cout << "function_1("<< arg << ")" << std::endl;

	return 0;
}

void function_2(int arg1, int arg2)
{
	std::cout << "function_2(" << arg1 << ", " << arg2 << ")" << std::endl;
}

double function_3(std::string str)
{
	std::cout << "function_3(" << str << ")" << std::endl;

	return 10.0;
}
```

The first function takes one argument of type **int** and returns **int**. So, the signal for this function should be like this:

```
vdk::signal<int(int)> sig1;
```

The second function takes two arguments of type **int** and returns **void**. Here is the second signal:

```
vdk::signal<void(int, int)> sig2;
```

The third function takes one argument of type **std::string** and returns **double**. This is the last signal:

```
vdk::signal<double(std::string)> sig3;
```

Now we can connect our signals to slots as we did before and emit them:

```
sig1.connect(&function_1);
sig2.connect(&function_2);
sig3.connect(&function_3);

sig1.emit(10);
sig2.emit(20, 30);
sig3.emit("string");
```

```
Output: function_1(10)
        function_2(20, 30)
        function_3(string)
```

As we can see all slots were called with expected arguments.

Connecting different kinds of slots

So far we saw how to connect standalone free functions to signals. Now let's have a look at how to connect methods of some class, function objects and lambdas. For the sake of simplicity, we are going to use slots with the same signature. Doing this we will be able to connect all those slots to one signal.

First of all, we create a signal that takes one argument of type **int** and returns **void**.

```
vdk::signal<void(int)> sig;
```

Next, we define some class with two methods one of which is static.

```
class SomeClass
{
public:

	static void static_method(int arg)
	{
		std::cout << "static_method(" << arg << ")" << std::endl;
	}

	void method(int arg)
	{
		std::cout << "method("<< arg << ")" << std::endl;
	}

};
```

Then we create an instance of this class and connect its methods to our signal.

```
SomeClass object;

sig.connect(&SomeClass::static_method);   // Connects static method
sig.connect(&object, &SomeClass::method); // Connects method
```

Note: static method does not need an object to be called on!

Now we define and create a function object.

```
struct SomeFunctor
{
	void operator()(int arg)
	{
		std::cout << "operator(" << arg << ")" << std::endl;
	}
};

SomeFunctor functor;
```

This functor can be connected to our signal as well.

```
sig.connect(&functor); // Connects function object
```

And the last thing we would like to connect to the signal – our friend lambda.

```
auto lambda = [](int arg)->void
{
	std::cout << "Hi, I am lambda! I have one argument: " << arg << std::endl;
};

sig.connect(&lambda); // Connects lambda
```

At this point we have the signal and four connected slots. We will call them all together by the signal emission:

```
sig.emit(10);
```

```
Output: static_method(10)
        method(10)
        operator(10)
        Hi, I am lambda! I have one argument: 10
```

Manual slot disconnection

Sometimes it is necessary to disconnect some slots from a signal. It can be done in exactly the same way as connecting them, but using **`disconnect()`** method. In the example below, we create the signal, connect it to the methods from previous example, and then disconnect them.

```
vdk::signal<void(int)> sig;

sig.connect(&SomeClass::static_method);
sig.connect(&object, &SomeClass::method);

sig.disconnect(&SomeClass::static_method);
sig.disconnect(&object, &SomeClass::method);
```

As can be seen in the example above, connecting and disconnecting slots are very similar: slots are disconnected from the signal in the same way as they were connected. If we try to emit the signal now – nothing happens.

It is also possible to disconnect all connected slots in one statement using **`disconnect_all()`** method.

```
sig.connect(&SomeClass::static_method);
sig.connect(&object, &SomeClass::method);

sig.disconnect_all();
```

Blocking signals

Signals can be blocked, meaning that the emission of such a signal has no effect and no connected slots are called. To block or unblock a signal – method **`block()`** is used. In the example below, we create the signal and connect it to slots from previous examples. Then we will emit this signal, block it, and emit again to see the effect of **`block()`** method.

```
vdk::signal<void(int)> sig;

sig.connect(&SomeClass::static_method);
sig.connect(&object, &SomeClass::method);

sig.emit(10); // Calls both slots
```

```
sig.block(); // Equivalent to sig.block(true);
sig.emit(10); // Has no effect
```

To unblock the signal we need to pass **false** to the **block()** method (by default this argument is **true**).

```
sig.block(false);

sig.emit(10); // Works again!
```

Automatic slot disconnection

vdk::signal can automatically track the lifetime of objects involved in signals-slots connections, including automatic disconnection of slots when objects involved in the slot call are destroyed. For example, what will happen if we do the following?

```
vdk::signal<void(int)> sig;

SomeClass * object = new SomeClass();

sig.connect(object, &SomeClass::method);

delete object;

sig.emit(10); // undefined behavior
```

This code results in undefined behavior, since we destroy the object just before signal emission, and slot-method is called on destructed object. To prevent this undesirable effect we could use trackable slot with **std::shared_ptr**. Instead of passing a raw object pointer to the **connect()** method, we provide **std::shared_ptr** that manages the object. This allows automatically disconnect the slot with destroyed object during signal emission. It is also guaranteed by the library that no trackable object expires while the slot it is associated with, is in the middle of execution.

Now, let's create an instance of **SomeClass** managed by **std::shared_ptr** and connect it to the signal.

```
vdk::signal<void(int)> sig;

std::shared_ptr<SomeClass> object_ptr(new SomeClass);

sig.connect(object_ptr, &SomeClass::method);

sig.emit(10);
```

Everything works fine as expected. But what will happen, if we do this:

```
object_ptr.reset();

sig.emit(10);
```

Nothing bad. Destroyed object does no cause undefined behavior any more. Inside signal instance it is tracked by **std::weak_ptr** and once the object destroyed, the associated slot is disconnected automatically.

Note: So far, to specify a slot for signal's methods (like connect / disconnect) we used raw object pointers. However, once we connected some trackable object, this object must be specified in the same way (through **std::shared_ptr** instance) in all signal's methods.

For example, if we connected the slot using raw pointer, like this:

```
SomeClass object;

sig.connect(&object, &SomeClass::method);
```

in the future we will have to specify this slot in the same way, for example:

```
sig.disconnect(&object, &SomeClass::method);
```

However, if we connected the trackable slot using **std::shared_ptr**, like this:

```
std::shared_ptr<SomeClass> object_ptr(new SomeClass);

sig.connect(object_ptr, &SomeClass::method);
```

in the future we will have to specify this slot in the same way, through **std::shared_ptr** instance.

For example:

```
sig.disconnect(object_ptr, &SomeClass::method);
```

Note: we must pass **std::shared_ptr** instance by value, not by pointer to **std::shared_ptr**.

Some useful methods

Here we briefly go through some useful methods. They are not as important as methods described earlier, but sometimes can be very useful.

How can we check if the particular slot is connected to our signal? Easily! Just pass the slot into **connected()** method. This method returns **true** if the slot is currently connected and **false** otherwise.

```
vdk::signal<void(int)> sig;

std::shared_ptr<SomeFunctor> functor_ptr(new SomeFunctor());


sig.connect(functor_ptr);


std::cout << sig.connected(functor_ptr) << std::endl; // Prints true
std::cout << sig.connected(&hello_world) << std::endl; // Prints false
```

We can find out how many slots are currently connected to the signal with **size()** method. We can also check if the signal is connected to something at all using **empty()** method.

```
vdk::signal<void(int)> sig;

std::cout << sig.size() << std::endl; // Prints 0
std::cout << sig.empty() << std::endl; // Prints true

sig.connect(&lambda);

std::cout << sig.size() << std::endl; // Prints 1
std::cout << sig.empty() << std::endl; // Prints false
```

How to specify the particular slot in the signal's methods?

It may be obvious by now that if we need to do something with some particular slot (connect, disconnect, etc.) we must specify this slot in the signal's methods, and every time it is done in exactly the same way for each type of slot.

To summarize the idea, look at the table below. It shows how to specify different types of callable entities (slots) in the signal's methods.

Slot Specifiers	
Standalone (free) function	`&function`
Static member function (static method)	`&SomeClass::static_method`
Member function (method)	`&object, &SomeClass::method`
Function object (functor)	`&functor`
Lambda expression	`&lambda`
Signal	`&signal`

Note: for trackable slots this table works exactly the same, except that instead of raw object pointers, **`std::shared_ptr`** instances must be passed (not pointers to shared pointers!).

Connecting signal directly to another signal

In previous examples we connected all sorts of slots to signals and they worked fine. However, what will happen, if we connect one signal directly to another? It will work just fine as well! In this case the first signal (the one that connects) will play a role of actual signal, and the other one (that is being connected) will play a role of a slot. Let's try this:

```
vdk::signal<void(int)> sig1;

sig1.connect(&SomeClass::static_method);
sig1.connect(&object, &SomeClass::method);
```

Now, we've created the signal and connected two slots to it, so we can see what happens when we connect this signal to the second one, and emit it.

```
vdk::signal<void(int)> sig2;

sig2.connect(&sig1);

sig2.emit(10); // Emits slots connected to the first signal
```

Of cause, we could connect the second signal as trackable using **`std::shared_ptr`**, and the result would be exactly the same. **Note:** in order to connect one signal to another, they both must have the same type, such as in this example they both take **`int`** and return **`void`**.

Self-tracking

Sometimes we may want to obtain **`std::shared_ptr`** in a class constructor to connect some signals with trackable slots. To do this, we need to add an additional first parameter in our class constructor, that takes **`std::shared_ptr<our_class>`**. So, let's do this:

```
class Class
{
	int some_data;
	vdk::signal<void(int)> sig;

public:
	Class(std::shared_ptr<Class> s_ptr, int arg)
	: some_data(arg)
	{
		sig.connect(s_ptr, &Class::method);
	}
```

```
        void emit_signal()
        {
                sig.emit(some_data);
        }

        void method(int arg)
        {
                std::cout << "method("<< arg << ")" << std::endl;
        }

};
```

The example may look weird and useless, however sometimes this functionality can be quite helpful. Now we can construct this object with special function **vdk::create_shared_object()** and emit the signal.

```
std::shared_ptr<Class> ptr = vdk::create_shared_object<Class>(10);

ptr->emit_signal();
```

As this example shows, now we have an object that tracks itself through **std::shared_ptr**.

Note: all arguments required in the class constructor must be passed into **vdk::create_shared_object()** function in the order they would be normally passed into constructor.

Important: If you wish to keep this **std::shared_ptr** inside your class – keep it as **std::weak_ptr**. Otherwise it may end up with memory leak, since instance of **std::shared_ptr** stored inside your class object will prevent this object from destruction.

Signal's constructor

So far we omitted the argument of signal's constructor. However, to ensure effective memory usage it would be a good idea to specify how much memory we will be using for the signal object. We can do this by passing the number of slots we are going to connect to the signal. It allows to allocate sufficient and contiguous memory block and use it during signal's lifetime.

So, for example, if we assume that only two slots are going to be connected to the signal, we can do this:

```
vdk::signal<void(int)> sig(2);
```

However, if our guess was not right and we connect three slots (or more), signal will have to allocate additional memory block (equal to capacity) that will be freed only during signal destruction. This may reduce the efficiency of memory usage, degrade performance and increase memory fragmentation. However, if you are not interested in extra efficiency, you can leave this argument and use default value, that is currently 5.

Design Rationale

This section intends to explain design decisions adopted in this library. It may help other software developers, who are going to implement something similar, pointing out the problems and small challenges they more likely will come across with. Users, who are not interested in implementation details may easily skip this entire section.

A few words from author…

You may be wondering why one more signals & slots library? There are a lot of them out there and they all are different in complexity and feature sets, but the idea is always the same. The main reason for me to write this library was the fact that all other implementations did not meet my requirements in many ways. I was working on a project that required some communication mechanism that had to be used under very high pressure. So, first of all I needed some thread-safe signals-slots system that could work fast enough – and it

was my main requirement. It is absolutely necessary to realize that signal emission must be as fast as possible. All other actions like connection / disconnection can take much more time since it happens great deal rarely. If it happens very often – signals-slots concept is probably not the right choice. In short – communication is the main job of signals-slots system – not connection / disconnection.

Several libraries were tested and the first one was **boost**. I must admit that **boost::signals2** library is very good, reliable, feature-full and can be the first choice in many cases, but I was completely disappointed with signal emission performance. I also could not use **Qt**'s signals since they work inside **Qt-framework** only.

Another very important thing – memory management. People who write complicated applications that require maximum performance (like games) know how important memory management can be. And it is not only about slowness of default **new** and **delete** operators, which make calls to the operating system. It is also about memory fragmentation. If signal instance creates its internal objects in separated places anywhere on heap – it is OK, until your program is required to work for a long period of time. And during this time your application may allocate and deallocate memory for some other objects according to its own inner logic. However, if you have a lot of signals and slots, and even if you do not connect / disconnect them too often, you may end up with very fragmented / shredded memory that slows everything down. So, at least a communication system of your application should not be a bottleneck and must not cause any problems with memory.

Summarizing, the general performance of this library should be good enough to use it everywhere: in single-threaded and multi-threaded environments, in games, in GUI development, in engineering programs, in audio-video applications – but especially in software systems that require the maximum speed and work under very high load.

Data structure

One of the most important things that needed to be considered while implementing this library was data structure. Choosing a wrong data structure could spoil the entire design and introduce a lot of undesirable effects. Each data structure has its own advantages and disadvantages, and as it is usually the case in engineering science, there is no absolutely perfect one that could be ideal for all scenarios. To choose the best suited structure it must be absolutely clear how signal works internally.

The signal object is just a container for a set of callbacks. During signal emission all slots must be executed one after another. There is no any special order of slot execution, but all slots, which are connected to the signal must be executed. So, there is no need to keep all connections in some particular sorted order. The data structure must have the ability to grow, so that any number of slots could be connected to any signal.

It is always a good idea to keep all related data close to each other, ideally in one memory block. This could be achieved using **std::vector**, but this data structure has a big disadvantage for signal implementation. The problem is in the nature of the data structure itself. If some slot is disconnected during signal emission (some element is deleted), **std::vector** moves all the following elements closer to the beginning of the data structure invalidating iterators. Even if slots are accessed by indexes some of them may be skipped in signal emission, if disconnection took place. This behavior is unacceptable. Another big problem is concurrency. If vector's internal array needs to be reallocated in memory during concurrent signal emission (for example as a result of connecting a new slot) all reading threads may end up with dangling pointers.

Taking this into account, it is obvious that the best choice for internal signal structure is linked list, since insertion / deletion of nodes does not invalidate iterators and all those nodes lay in memory independently from each other, linked only by pointers. However, the linked list for this library should be a special one. It should keep all its nodes in one contiguous memory block to improve memory usage efficiency and for this purpose it uses memory manager.

Memory management

Effective memory management was one of the main goals for **vdk-signals**. Since it is not possible for the library implementer to know in advance how many slots will be connected to the signal, using dynamic memory allocation is unavoidable. Moreover, a huge amount of connected slots would make an internal list too big to keep it on stack anyway.

On the other hand, if signal object creates nodes for the list of connected slots anywhere on heap using default **new** and **delete** operators every time it needs to allocate memory for a new node, this allocation scheme leads to significant memory fragmentation. It is a good design choice to keep all data related to the signal object in a single contiguous memory block of appropriate capacity and provide this capacity as a tool for a library user to make memory allocation as efficient as possible for each particular application.

So, according to these conclusions **vdk::signal** object allocates a raw memory up-front that is equal to provided (or default) capacity value multiplied by the size of connection object and uses it as a memory storage, or buffer, just like it would be done by fixed size memory pool. It is extremely fast operation to allocate / deallocate memory from such a pool. In general, it is equal to two pointer modifications. If some slot is disconnected (node is deleted) the linked list just tells the underlying memory manager (memory pool) that this node's space can be recycled. If some slot is connected (node is inserted), underlying memory manager just acquires new memory block from its internal buffer. If internal buffer is exhausted, memory manager allocates new memory block from the operating system with the same capacity and connects it to the previous one, resulting in something like a linked list of big memory blocks (pages, regions) each of which contains nodes for signal's linked list. Ideally, if user provides an appropriate capacity there will be only one such memory block allocated for all connection objects of a particular signal.

To make this allocation scheme possible, **vdk::signal** constructor expects an optional argument – capacity. It should be treated as a hint that reflects approximate (better maximum) number of connected slots. If this capacity value is too small, memory manager will have to allocate new memory blocks (each of which of the same capacity as the first one) that will be freed only during signal object destruction. It degrades performance and memory usage efficiency. If the capacity value is too big, it will lead to unnecessary memory wastes. However, memory management may be not that critical for a lot of applications, which do not require high efficiency and performance, especially considering the fact that the absolute majority of existing signals-slots libraries do not even consider any special allocation schemes.

Thread-Safety

The primary motivation for **vdk-signals** is to provide a reliable and fast library, which can be used safely in a multi-threaded environment under very high pressure (with multiple concurrent emissions). This is achieved primarily through C++11 Standard atomic variables and specialized synchronization mechanism. Atomic variables ensure maximum performance for signal emission in multi-threaded application, and also help to make this emission non-blocking. In order to better understand design decisions regarding thread-safety of this library, some examined approaches should be discussed first.

In general, a callback should not hold any lock during its execution, since library implementer has no way to know in advance what this callback is supposed to do. Violation of this general rule may lead to undesirable consequences, such as deadlocks. Also, acquiring a lock on some synchronization primitive (like **std::mutex**) usually takes significant amount of time, but signal emission, and consequently callback execution, must be as fast as possible. Moreover, holding a lock during signal emission prevents other threads from concurrent emissions and introduces lock convoy that again violates the primary goal of this library. Even a reader-writer lock is not an option here, since some slot may need to connect or disconnect another slot during its execution, and it needs to acquire write-lock for that. But how can it acquire write-lock, if it already holds

read-lock (acquired in the beginning of signal emission)? As a result, this slot would be waiting until all readers (including itself) release read-lock to perform disconnection (acquire a write-lock). But it will never happen because this slot is in the middle of execution and is waiting the time when it will finish it. Recursive reader-writer lock is not an option either, because of above reasons and because connection / disconnection can be called as a result of slot execution from different threads.

The main problem with lock-free data structures is synchronization and element deletion. Without locks it is very hard to find out if someone is holding a reference to a linked list node and as a result if it is safe to delete the node. There are some techniques for writing lock-free data structures, which were considered for implementation in the library and carefully examined. However, after detailed research it was obvious that they were not appropriate for this library for one reason or another (the majority of such techniques are not portable). Moreover, it was not required to make all operations on the data structure lock-free, since underlying memory manager that allocates memory needs synchronization anyway. Connection and disconnection methods could still hold a lock, since according to the assumption above and general signals-slots concept they are called rarely and it would not cause significant performance penalty.

Several mechanisms that allow lock-free reads of a data structure were tested. The first one that worth mentioning is read-copy-update. The basic idea is quite simple. An object that represents a data structure (linked list) has an atomic pointer that points to the first node and a mutex for serialization of modifications. Reading threads do not acquire the mutex, they just load the atomic pointer and traverse the data structure it points to. Any thread, which modifies the linked list, locks the mutex, copies all nodes to the new place in memory and makes all required changes (updates), while other threads can't see it. After that, this thread modifies the atomic pointer to point to the first node of the new updated data structure and deletes the old version. So, all new coming threads can see the linked list in already updated state.

This mechanism suffers from several problems. First of all, it is a huge performance penalty if the entire data structure has to be copied for every single modification. Second, what happens, if some slot disconnects another slot? The first slot locks the mutex, copies and modifies the entire linked list and… waits forever until it will be able to delete the old version of the data structure, from which it is currently being called. This scheme can be improved with reference counting, so each thread can check if it holds the last reference to the old version of the data structure, and if so, safely delete it. However, this introduces another serious problem. What if we modified the linked list, let's say 5 times (connected 5 slots), and then we need to delete just one node (disconnect one slot)? The entire data structure is copied and updated 5 times, and every copy of the data structure contains an exemplar of the node we are going to delete. So, after this node deletion it is still pretty much alive in all copies of the data structure and can be called. What if we disconnected several slot? They all will be called from those per-thread caches. This problem can also be solved with one more level of indirection, but it will introduce additional overhead and the entire design is turning into a mess.

Another approach that needs to be discussed here in order to better understand design decisions is deferred synchronization. This approach is even simpler and more efficient than the first one. Again, the data structure has an atomic pointer and an atomic counter. But this time all changes to the linked list are made in place and no coping is required. Each node has an atomic pointer to the next node, and this chain forms so called "active linked list". In addition, each node has a pointer that is used when this node is logically removed from the active list and connected to the linked list of removed nodes. So, every reading thread increments atomic counter at the beginning of traversing and decrements it at the end. If some node needs to be added (slot is connected), this node is created and atomically inserted into the active list just by updating the atomic pointer of the previous node. If some node needs to be deleted (slot is disconnected), this node is logically removed from the active list and added to the list of removed nodes. **Note:** no physical reallocation or copying is required here, just pointer updates. Each modifying thread checks the atomic counter to see

whether the data structure is in so called quiescent state. Quiescent state is a point in time when no readers are active. Even if this period of time is just a time slice (extremely small), it is enough to be sure that all logically removed nodes cannot be accessed from active list by new coming threads and hence can be safely deleted.

This approach, however, has a noticeable disadvantage. It works well in low-load situations, when we can be sure that there are points in time when no readers are active. In high-load situations there may never be a quiescent state. So, it leads to unnecessary memory consumption and very undetermined memory reclamation. Not only that, but if there are a lot of connected slots (list has a lot of nodes) and some of those slots spend much time (more or less) for execution, the same effect appears – no slots can be physically deleted, because some threads start signal emission before all other threads finished it.

Now, it is obvious that all approaches discussed above do not work well for signal-slot system. So, I implemented another mechanism for this library. The general idea was taken from deferred synchronization, but it was improved to make it possible to delete nodes even if the linked list is incessantly traversed by readers.

The signal object consists of two atomic counters, one atomic pointer to the first node (first connected slot), two plain (non-atomic) pointers for logically removed nodes, and a write lock. **Note:** this lock is needed to serialize write accesses, and does not affect reading threads. Each node (connection object) consists of an atomic pointer to the next node (it forms so called active list) and a plain pointer for logical removal.

The key concept of this mechanism is using two synchronization stages for reads of the data structure. Once a thread starts signal emission (traversing the list of connected slots) it acquires a read access incrementing the read counter related to the current stage. Once the thread has finished signal emission, it releases read access decrementing the counter that was incremented at the beginning of emission.

When some slot needs to be connected (node is inserted) – everything is quite simple. Since it is modifying operation the write lock is acquired, new node is created and atomically inserted into the active list. When some slot needs to be disconnected (node is deleted) it is logically removed from the active list, just like it is done in deferred synchronization mechanism. However, this logically removed node is connected to the list of removed nodes related to the current stage.

Stages are switched by modifying threads only (while write access is acquired), and only after physical deletion of removed nodes from the opposite stage. It is important that physical deletion must take place only in the list of removed nodes, that is related to the stage opposite to the current stage.

It is safe to delete the entire list of logically removed nodes when its stage counter reaches zero. And it will eventually reach zero since all new coming threads are incrementing only one particular stage counter and all threads that are in the middle of signal execution will finally decrement the stage counter they incremented in the beginning of emission. It is not allowed to change the stage until previous stage was cleaned up and all logically removed nodes were deleted physically.

During every modifying operation such as slot connection / disconnection the thread checks if counter of the opposite stage is zero. If it is this thread deletes all logically removed nodes, which may be there, and switches the stage. If opposite stage counter is not zero, the thread just keeps doing what it intended to do.

This mechanism makes signal emission lock-free and actually wait-free, since no thread interferes with another one in making progress. The only case when signal emission will be locked – if it detects expired slot. In this case I acquires a lock to disconnect this expired slot and saves all other threads from checking this expired slot many times. It is important to mention, that this lock cannot lead to deadlock against any other locks in user's code, since it is acquired and released locally inside signal object and during this time the thread does not execute any user's code.

So, the primary goal is achieved. This algorithm is able to physically delete elements even if there is absolutely no period in time when all readers are inactive. Another main advantage of this algorithm is that there is no need in coping entire data structure or keeping several versions of the same data structure, etc. All modifications are made in place.

Return values

Some libraries allow to receive return values from slot executions. It may seem as a trivial task, but in practice in turns into a real problem. Receiving return values is usually accomplished in one of two ways:

1) Signal emission returns the value obtained from the last slot execution.
2) Signal has a special combiner / aggregator / collector class that receives all values and handles them.

In this library implementation none of them have been chosen. The first option looks extremely weird. Is it really possible scenario that only the last value is needed? What about the rest of them? Why exactly the last one and not some from the middle? Or the first one? What possible useful information could be taken from this last value?

The second option looks much more reasonable, but it has its own drawbacks. In multithreaded environment this combiner object must be thread-safe, otherwise the signal object needs to care about some kind of mutual exclusion while working with this combiner. Another problem is the fact that this "feature" enormously degrades emission performance and is needed so rarely (I still have not been in such a situation), that there was no point to implement it in this library. However, it may change in the future and if I find some way to provide this functionality without significant performance cost – I may implement it as well. But for now, I don't waste performance for rarely used functionality. It is C++ after all! You don't pay, if you don't use!

Tracking mechanism

Auto-disconnection is absolutely necessary feature for any signals-slots implementation. However, it is also worth mentioning here that tracking mechanism must not be intrusive. A user should have a choice whether he / she wants to use auto-disconnection of expired slots or not. It must be in this way, because any type of tracking mechanism always has its overhead, but according to C++ philosophy, a user must have a choice to decide, whether this overhead is acceptable for the particular case or not. If not, it should be possible to avoid it. There are three common ways to implement tracking mechanism. Let's look at them one by one.

The first option is to make the class that needs to be tracked inherit from some kind of base class, like **`trackable`**, **`has_slots`**, etc. This base class keeps a container with pointers to the signals it is connected to (or pointers to connection objects). Once the tracked class object is destroyed, its base class object is also destroyed and in its destructor it traverses the container of connected signals (or connections) disconnecting itself from each and every one of them. This mechanism may seem straightforward, simple and reliable. However, it has some serious disadvantages.

The first one – it is not reliable in multi-threaded environment, since the destructor of the base class is called after the destructor of derived (tracked) class. In the point in time when derived class is already destroyed and the base class has not disconnected itself yet, any slot on partially destroyed object can be called resulting in undefined behavior.

Another problem of this approach is inheritance. Tracked class needs to inherit from some base class that introduces unnecessary relations (I would call it noise) in inheritance chain, and all this just to disconnect the object from signals at the end of its lifetime. Moreover, what if we wanted to connect methods of some class that belongs to a third-party library? Obviously, this library was not designed to work with such a signals-slots system and does not derive from our base class. The last point may not seem as a big problem, since

we could just make it as a library requirement (restriction) – class derives from our base class if it wants to be automatically disconnected, otherwise it can be disconnected manually.

The second option to set up tracking for an object is very similar to the previous one. However, instead of inhering from some base class, we intrude special object (releaser) inside the tracked class. It can be called **`releaser`**, **`disconnector`**, etc. In this case there is no any unnecessary inheritance, since this releaser object is just a member variable – field. However, this approach is not any better than the previous one. It is still required to modify the tracked class, and it is still not reliable in multithreaded environment. If the releaser object is somewhere in the middle of the tracked class structure, then there is a possibility that releaser's destructor will be called after tracked class object is already partially destroyed, so there is a window for concurrent thread to call some methods on partially destroyed object.

Of cause, we could make a restriction to put the releaser object at the very bottom of the tracked class. So, when destruction starts, all member variables are destroyed in reverse order, and destructor of the last one will be executed first, disconnecting all signals. However, it is not a very good solution. We could make almost the same restriction in the previous approach: user is required to disconnect all slots in the destructor of tracked class, probably by some special method. In both cases it is easy to forget about these restrictions since nothing will remind about it. Moreover, some new maintainer of existing code may change something in tracked object structure without even knowing about all these restrictions, and once for a while slots will be called on the not completely alive object.

The third option is tracking by **`std::shared_ptr`** / **`std::weak_ptr`**. This approach was taken by **`boost::signals2`**, and it is the most flexible, powerful and reliable solution. It does not introduce any unnecessary inheritance, may be used even with classes from third-party libraries, which were implemented earlier, without any signals-slots system in mind. It does not involve any changes in the structure of the tracked class itself, that may not even know that it is participating in some signals-slots communication. The last point enables the real decoupling, since the tracked object has absolutely no idea that it is tracked by something. This approach makes it possible to build completely independent software components. In addition, it is absolutely thread-safe, since **`std::shared_ptr`** control block uses atomic reference counters.

The idea of this tracking is simple. An object that needs to be tracked is manager by **`std::shared_ptr`**. When it is being connected to some signal, this **`std::shared_ptr`** is converted to **`std::weak_ptr`** in **`connect()`** method and stored inside connection object in signal's data structure. Once some slot of the tracked object needs to be executed, **`std::weak_ptr`** is converted back to **`std::shared_ptr`**. If this new temporary **`std::shared_ptr`** is empty – object was destroyed and, as a result, slot is expired and cannot be called. In this case, the connection is disconnected (deleted from the connection list). If **`std::shared_ptr`** is not empty, object still alive and can be called. After the slot execution, this temporary **`std::shared_ptr`** is destroyed, allowing tracked object to expire in the usual way. This scheme also guarantees that the slot cannot expire during its execution, because of that temporary **`std::shared_ptr`** that have taken ownership of the tracked object for the time of execution.

Self-tracking

`vdk-signals` uses **`std::shared_ptr`** / **`std::weak_ptr`** to track the lifetime of connected objects, when necessary. However, there is one special case that may be interesting. What if some object needs to set up tracking of itself? There are several options to deal with it:

1) Connect signals and slots in the class's constructor and explicitly disconnect them in destructor, without tracking. If this is possible in the particular situation, it would be the most preferable way since the lifetime of signals and slots strictly scoped between the object's constructor and destructor and it is obvious when they need to be disconnected. It is also worth mentioning that slots without tracking are faster to execute than trackable ones.

2) There are two helper standalone functions in the library specifically designed for this purpose: "**vdk::create_shared_object()**" and "**vdk::allocate_shared_object()**". These functions are identical in everything except memory allocation. The first one uses usual "**::operator new**" and "**::operator delete**", the second one uses provided allocator. Both functions create an object passing the instance of **std::shared_ptr** (that manages this object) in its constructor as the first argument, and return one more **std::shared_ptr** for external usage. **Note:** to use these functions your class must take **std::shared_ptr<your_class>** as the first argument in its constructor. **Important:** if you need to store this **std::shared_ptr** obtained from the constructor inside your class for later usage, you should keep it as **std::weak_ptr**. Otherwise, if you keep it as **std::shared_ptr** it will prevent the entire object from destruction even though there is no other references to it from outer world. If constructor of your class expects some other arguments (see tutorial), they must be passed into "**create_shared_object()**" or "**allocate_shared_object()**" in the order they would be passed directly into constructor.
3) Another solution could be using **std::enable_shared_from_this**, but this class does not allow retrieving **std::shared_ptr** from within constructor. So it can be considered as partial solution.

Of cause, there are also some other approaches. For example, **boost::signals2** uses a special concept – post-constructors and pre-destructors. However, detailed explanation of all possible solutions to set up self-tracking is barely related to signals and slots themselves. Here I introduced just the most convenient solutions.

The order of slot executions

In some libraries it is possible to specify the order of slot executions. It may seem as a good thing, but in general, the application that uses signals-slots concept should not rely on any particular order of slot executions. They may be called in arbitrary order and not necessarily in connection order. Moreover, it is very hard to reason about any particular order in multithreaded environment, where everything happens concurrently. A slot, that might be expected to execute next, can be disconnected by some other thread that just has had a chance (and time from thread scheduler) to get to this slot first.

It is also worth mentioning, that signals-slots concept is a variant of Observer Pattern, and it is considered to be a good practice for this design pattern, that all observers observing the same subject should not have any ordering dependencies relative to each other. The same principle applied in this library.

Single-threaded mode

Sometimes signals-slots mechanism may be used in single-threaded environment and it would be a good thing to avoid any unnecessary overhead introduced by thread-safety. For example, it can be useful in GUI programming, since the majority of graphical user interfaces work in one thread, specially dedicated for this purpose. For such scenarios, **vdk-signals** introduces an additional library mode, optimized for single-threaded usage.

In this mode, the library does not use any atomic variables, or any locking mechanisms. It is also worth mentioning here, that in this mode the library has a slightly different memory allocation scheme. It always keeps all connection objects in single memory block, even if provided capacity value was not enough. The signal object itself also takes less memory, than its multi-threaded friend.

To activate this mode, the special macro must be defined ***before*** the library header inclusion. This macro is: **VDK_SIGNALS_LITE**. After this macro is defined the library works in single-threaded mode and **vdk::signal** object should not be accessed concurrently from multiple threads. This mode was introduced as a special tool for extremely time-critical cases, when no overhead caused by thread-safety is acceptable.

Performance

vdk-signals library was mainly tested on the following platforms:

1) Windows 10 with Visual Studio 2015 Community Edition. C/C++ Compiler Version 19.00.23506.
2) Linux openSuse Leap 42.1. GCC Version 4.8.5.
3) Linux Ubuntu 14.04. GCC Version 4.8.4.

Important: In order to use this library, the compiler must be compatible with at least C++11 Standard.

To provide the performance comparison and to show the difference between this library and some other one - **boost::signals2** was chosen, since it is one of the most famous multi-threaded signals-slots libraries. Additional comparison with **Qt**'s signals is also provided, but for single thread only, since this framework uses its own model for cross thread signal-slot communications.

All tests were performed on Visual Studio 2015 in release builds. The tables below contain data representing time spent on signal emissions by each library (less is better). **vdk-signals** and **boost::signals2** were tested with exactly the same conditions:

1) Slots are not trackable;
2) Number of connected slots: from one to ten;
3) Number of spawned threads: from one to four.

Note: these tests are not intended to show this library under high load or to provide exact time intervals spent on signal emissions (it may vary from machine to machine), but aims to give an impression and analysis of its ***comparative*** (relative) performance against another multi-threaded library, tested on the same operating system, the same compiler, and the same hardware.

vdk-signals			
	1 slot	5 slots	10 slots
1 thread	38	41	42
2 threads	121	137	163
4 threads	768	774	786

boost::signals2			
	1 slot	5 slots	10 slots
1 thread	314	725	1237
2 threads	961	2375	3923
4 threads	6872	20014	35453

vdk-signals (lite mode)			
	1 slot	5 slots	10 slots
1 thread	11	12	14

Qt			
	1 slot	5 slots	10 slots
1 thread	134	408	758

To be completely accurate, it is necessary to point out that **Qt** slots are always tracked by signals. So, in the table above it should be considered while comparing **Qt**'s performance against all others.

Public API

vdk::signal<ResT(ArgTs...)>

```
explicit signal(size_type capacity = 5);
```

Constructs a new signal object with provided / default capacity. Allocates memory for internal data structure according to the **capacity**. May throw **std::bad_alloc** or derived exception of memory allocation fails. It is recommended to set the capacity value equal to expected number of connected slots.

```
~signal() noexcept;
```

Destructs the signal object. Deallocates all previously allocated memory.

```
1)   bool connect(Function * function);
2)   bool connect(Class * object, Method * method);
3)   bool connect(std::shared_ptr<Class> object, Method * method);
4)   bool connect(Class * functor);
5)   bool connect(std::shared_ptr<Class> functor);
```

Connects the signal to the slot. Overloaded for the following types of slots:

1) Function, static method
2) Method
3) Method on tracked object
4) Functor
5) Tracked functor

Returns **true** if connection is successful and **false** if the slot is already connected to the signal. If capacity is exceeded, allocates additional memory block (equal to capacity) from the operating system. Throws **std::bad_alloc** or derived exception if this allocation fails. If the exception is thrown, this method has no effect.

```
1)   bool disconnect(Function * function) noexcept;
2)   bool disconnect(Class * object, Method * method) noexcept;
3)   bool disconnect(std::shared_ptr<Class> object, Method * method) noexcept;
4)   bool disconnect(Class * functor) noexcept;
5)   bool disconnect(std::shared_ptr<Class> functor) noexcept;
```

Disconnects the signal from the slot. Overloaded for the following types of slots:

1) Function, static method
2) Method
3) Method on tracked object
4) Functor
5) Tracked functor

Returns **true** if disconnection is successful and **false** if the slot is not connected to the signal.

```
void disconnect_all() noexcept;
```

Disconnects the signal from all connected slots.

```
1)   bool connected(Function * function) const noexcept;
2)   bool connected(Class * object, Method * method) const noexcept;
3)   bool connected(std::shared_ptr<Class> object, Method * method) const noexcept;
4)   bool connected(Class * functor) const noexcept;
5)   bool connected(std::shared_ptr<Class> functor) const noexcept;
```

Checks whether the slot is connected to the signal. Overloaded for the following types of slots:

1) Function, static method
2) Method
3) Method on tracked object
4) Functor
5) Tracked functor

Returns **true** if the slot is currently connected and **false** otherwise.

```
void block(bool blocked = true) noexcept;
```

Blocks the signal if argument **blocked** is **true**. Otherwise unblocks it. Blocked signal does not invoke any connected slots during emission.

```
bool blocked() const noexcept;
```

Checks whether the signal is blocked. Returns **true** if the signal is currently blocked and **false** otherwise.

```
void emit(ArgTs … args);
void operator(ArgTs … args);
```

Emits the signal. Invokes all connected slots one by one with provided arguments. If one of slots throws an exception, this exception propagates to the signal invoker and emission stops. If tracked slot is expired (object destroyed), this slot is detected and disconnected during the signal emission.

```
size_type size() const noexcept;
```

Returns the number of currently connected slots.

```
bool empty() const noexcept;
```

Returns **true** if the signal has no connected slots and **false** otherwise.

Non-member functions

```
template<typename T, typename … ArgTs>
std::shared_ptr<T> create_shared_object(ArgTs … args);
```

Constructs an object of type **T**, passing **std::shared_ptr<T>** and **args** as arguments to the constructor of **T**. **std::shared_ptr<T>** must be the first argument in **T**'s constructor. Allocates memory using **::operator new**, deallocates using **::operator delete**. May throw **std::bad_alloc** or derived exception of memory allocation fails. **T**'s constructor must not throw.

```
template<typename T, typename Alloc, typename … ArgTs>
std::shared_ptr<T> allocate_shared_object(const Alloc & alloc, ArgTs … args);
```

Constructs an object of type **T**, passing **std::shared_ptr<T>** and **args** as arguments to the constructor of **T**. **std::shared_ptr<T>** must be the first argument in **T**'s constructor. Allocates memory using provided allocator **alloc**, deallocates using the same allocator. May throw **std::bad_alloc** or derived exception of memory allocation fails. **T**'s constructor must not throw. **alloc** must satisfy C++ *Allocator* concept requirements.

I hope this library will be useful to you. If you have any comments, questions, ideas, suggestions, or propositions, please feel free to contact me at: **vdksoft@gmail.com**. All feedbacks will be much appreciated.