YSignalSlot

YSignalSlot is a header only C++ library. It is an implementation of signal-slot mechanism via C++. It is similar to C++ signal, Boost::signal and SigSlot. It intensively uses C++11 features.

Advantages:

- Almost everything is compile-time. So it is fast.
- It can be used with single slot function or unlimited slot functions.
- "emit" function's return type is pointer of your actual slot function's return type and it can be any type.
- It supports "void" return type.
- Slot(s) can be set to enable or disable. If slot(s) is/are disabled, signal can't emit this/these slot(s).
- If slot which emitted is disabled or its return type is void, "emit" function returns "NULL" pointer.
- It works with member slot functions.
- Slot functions' signatures are converted and stored as string.
- Slot functions' argument counts and receivers are stored.
- It uses meta programming methods for "static for loop" and "static if".
- Signals can be used nested.
- Receivers can be "this" pointer.
- It is free and LGPL licensed.
- It uses C++11 features.

Disadvantages:

- It works only member functions.
- Executable size may be large. Because codes generated at compile-time, not run-time.
- It doesn't works with "const" member functions. (like "int function1(int) const;")
- Users must add "-std=c++0x" flag to compiler. This is required for using C++11.
- It generates a struct inside the class where you define signal. So it uses a little more code size.

Usage And Mechanism:

1) Including and Compiling:

```
#include "YSignalSlot.h"
```

When you are compiling your code you must add "-std=c++0x" flag to compiler. That's all.

2) Creating A Slot Function:

A slot function is created with "**YSLOT()**" macro. This macro creates a **new** object from the "**YSignalSlot_SlotFunction**" class. This class acts like a container of a single member function's properties and it uses like receiver. These properties are return type of a function, class name where its defined, signature (function pointer), string version of signature (with return type, class name, function name and arguments), argument counts and pointer of a receiver object which is function's class type.

"**YSLOT()**" macro's arguments are return type, class name, function name and argument types of function. Return type and arguments can be **any** type, like user defined class. Slot function must be defined before the macro.

a) Example:

```
class ExampleClass
{
    public:
        char slotFunction (int);
};

int main()
{
    YSLOT(char, ExampleClass, slotFunction, int)
    return 0;
}
```

After compiler expands the macro, code turns into like below.

```
class ExampleClass
{
    public:
        char slotFunction(int);
};
```

```
int main()
{
     (new YSignalSlot_SlotFunction
      <ExampleClass, char (ExampleClass::*)(int), char>
      (&ExampleClass::slotFunction,
       YSIGNALSLOT_STRINGIFY_FUNCTION_SIGN(char,
                                           ExampleClass,
                                           slotFunction,
                                           int)
     ))
     return 0;
}
```

3) Creating A Single Signal:

A single signal is created with "**YSIGNAL()**" macro. This macro generates a whole struct's code. The struct name is determined with first parameter of "**YSIGNAL()**" macro, "**signalName**". This macro generates some typedefs (like return type, receiver type), constructor of struct, "**emit**" function with pointer of slot function's return type, helper functions (getSlotSign, getArgumentCount, getReceiver, setEnable, setDisable, isEnabled) and an object pointer from generated struct with "**signalName**" variable name. After generated signal struct code, it has these functions and typedefs:

- **returnType** (**typedef**)
- **receiver** (**typedef**)
- constructor(receiver* rcvPtr)
- returnType* **emit**(…args) (**template**)
- std::string& **getSlotSign**()
- unsigned int **getArgumentCount**()
- receiver * **getReceiver**()
- void **setEnable**()
- void **setDisable**()
- bool **isEnabled**()

"**YSIGNAL ()**" macro's arguments are "**signalName**" and pointer of object which instantiated from "**YSignalSlot_SlotFunction**" class. The second argument should provide with "**YSLOT()**" macro. Signals name has to be unique in defined class scope. "**YSIGNAL ()**" macro must use in the class scope.

After defining signal struct with "**YSIGNAL ()**" macro, the pointer of signal struct must be instantiate. This can do with "**YSIGNAL_INIT()**" macro. This macro takes two arguments, "**signalName**" and object pointer of receiver class which defined in "**YSLOT()**" macro's second argument (class name).

a) Warnings:

Users must be careful at these points:

- "**YSIGNAL ()**" macro must be used in the class scope.
- "**YSIGNAL ()**" macro must be used after the slot function definition.
- "**YSLOT ()**" macro should be used at the second parameter of "**YSIGNAL ()**".
- "**YSIGNAL_INIT ()**" macro must be used in the constructor of class.
- Slot function must exist really. If any difference of real slot function (like return type, arguments etc.), you will get the compiler error.
- The object pointer of receiver class which used in "**YSIGNAL_INIT ()**" macro, must be instantiated before "**YSIGNAL_INIT ()**". This doesn't give an error but your code won't run correctly and terminated at this point. Type of this error is "SIGFAULT".
- If you use a slot function which defined in another class, this slot function must be public. Because you can't access the private members of another class(es).
- Signals can be public, private or protected. It depends on to where you call the "**YSIGNAL ()**" macro. Public signals can emit from outside of the class scope. This can be cause of undesirable situations happen.
- More examples about the using of single signal (like slot functions at another class, nested signals etc.) are located at "examples" directory.

b) Example:

```
class ExampleClass
{
    public:
        ExampleClass();
        char slotFunction(int);
        void otherFunction();
        YSIGNAL(mySignal, YSLOT(char,ExampleClass,slotFunction,int))
};

ExampleClass::ExampleClass()
{
    YSIGNAL_INIT( mySignal, this )
}


char ExampleClass::slotFunction(int x)
{
    // do something in here like below
    return (char)x;
}
```

```
void ExampleClass::otherFunction()
{
    std::cout << "dereferenced return value of slot is: "
              << *(mySignal->emit(89));
}

int main()
{
    ExampleClass test1;
    test1.otherFunction();
    // Output: dereferenced return value of slot is: Y
    return 0;
}
```

After compiler expands the "**YSIGNAL(** mySignal, **slot** **)**" macro, class code turns into like below. "**slot**" means **YSLOT(**char,ExampleClass,slotFunction,int**)** and this macro will expand also, but this is not shown below.

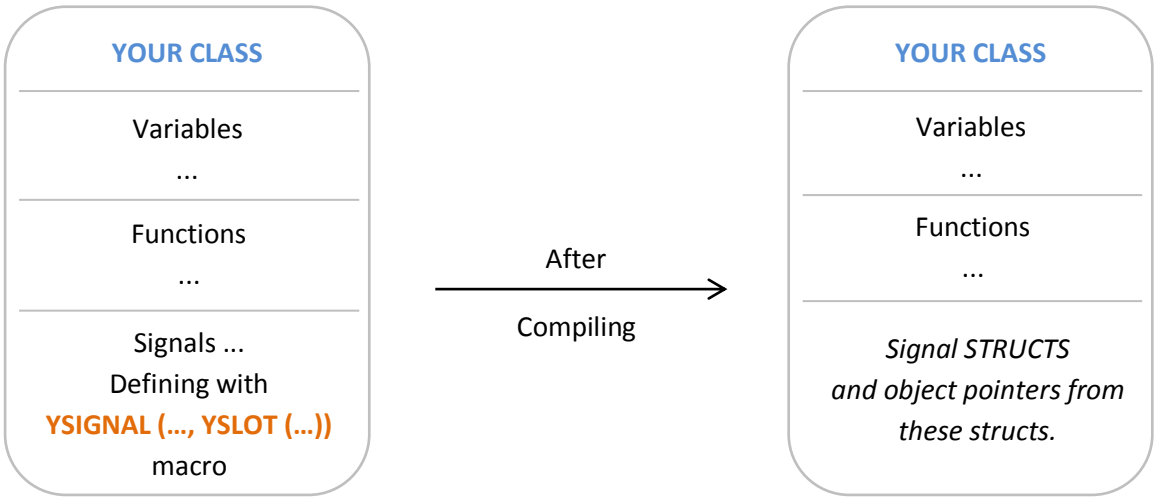
```
class ExampleClass
{
  public:
    ExampleClass();
    char slotFunction(int);
    void otherFunction();



    struct SIGNAL_STRUCT_mySignal
    {
      public:
        typedef YSIGNALSLOT_TYPEOFPOINTER(slot)::returnType returnType;
        typedef YSIGNALSLOT_TYPEOFPOINTER(slot)::receiver   receiver;

      private:
        decltype(slot)  __slot;
        receiver       *__receiver;
        bool            __enableState;

      public:
        SIGNAL_STRUCT_ mySignal(receiver *rcvr)
        {
          __slot        = slot;
          __receiver    = rcvr;
          __enableState = true;
          __slot->setReceiver(rcvr);
        }
```

```
        template<typename... Args>
        auto emit(Args ...args)
        -> decltype( (__receiver->*(__slot->getSign()))(args...) )*
        {
          if( __enableState )
          {
            /* Emitting of single enabled slot with static if */
            /* Static if is used for deducting of slot function return
               type is void or not
            */
            return YSignalSlot_EmitFuncHelper<
                                    std::is_void<returnType>::value,
                                    decltype(__slot), Args...
                                             >
                                             ()(__slot, args...);
          }
          else
          {
            returnType *ret = NULL;
            return ret;
          }
        }

        std::string& getSlotSign()
        {
          return __slot->getStringifiedSign();
        }




        unsigned int getArgumentCount()
        {
          return __slot->getArgumentCount();
        }

        auto getReceiver() -> decltype( __slot->getReceiver())
        {
          return __slot->getReceiver();
        }

        void setEnable()
        {
          __enableState = true;
        }

        void setDisable()
        {
          __enableState = false;
        }

        bool isEnabled()
        {
          return __enableState;
        }
    };

    SIGNAL_STRUCT_mySignal *mySignal;
};
```

And the compiler expands also **YSIGNAL_INIT(** mySignal, this **)** like below.

```
ExampleClass::ExampleClass()
{
    mySignal = new SIGNAL_STRUCT_mySignal(this);
}
```

c) Diagram:



4) Creating A Multi Signal:

A multi signal is created with "**YSIGNAL_MULTI ()**" macro. This macro generates a whole struct's code. The struct name is determined with first parameter of "**YSIGNAL_MULTI()**" macro, "**signalName**". This macro generates constructor of struct, "**emit**" function with pointer of selected slot function's return type, "**emitAll**" function which calls all slots, helper functions (getSlotSign, getSlotCount, getArgumentCount, getReceiver, setEnable, setDisable, isEnabled) and an object pointer from generated struct with "**signalName**" variable name. After generated signal struct code, it has these functions:

- constructor(receiver* rcvPtr)
- [various]* **emit**(...args) (**template**)
- void **emitAll**(...args) (**template**)
- std::string& **getSlotSign**() (**template**)
- unsigned int **getSlotCount**()
- unsigned int **getArgumentCount**() (**template**)
- [various]* **getReceiver**() (**template**)
- void **setEnable**()
- void **setDisable**()
- bool **isEnabled**()

"**YSIGNAL_MULTI ()**" macro's arguments are "**signalName**" and pointer(s) of object(s) which instantiated from "**YSignalSlot_SlotFunction**" class. The arguments except first argument should provide with "**YSLOT()**" macro. Signals name has to be unique in defined class scope. "**YSIGNAL_MULTI ()**" macro must use in the class scope.

After defining signal struct with "**YSIGNAL_MULTI ()**" macro, the pointer of signal struct object must be instantiate. This can do with "**YSIGNAL_MULTI_INIT ()**" macro. This macro can take two or more arguments. These are "**signalName**" and object pointer(s) of receiver class(es) which defined in "**YSLOT()**" macros' second argument (class name). Users can send more than one slot function to "**YSIGNAL_MULTI ()**" macro.

a) Warnings:

Users must be careful at these points:

- "**YSIGNAL_MULTI ()**" macro must be used in the class scope.
- "**YSIGNAL_MULTI ()**" macro must be used after the slot function(s) definition.
- "**YSLOT()**" macro should be used at the second and after parameters of "**YSIGNAL_MULTI ()**".
- "**YSIGNAL_MULTI_INIT ()**" macro must be used in the constructor of class.
- Numbers of object pointer arguments of "**YSIGNAL_MULTI_INIT ()**" macro have to be same size with the numbers of slot function(s).
- Slot function(s) must exist really. If any difference of real slot function(s) like return type, arguments etc., you will get the compiler error.
- The object pointers of receiver class(es) which used in "**YSIGNAL_MULTI_INIT ()**" macro, must be instantiated before "**YSIGNAL_MULTI_INIT ()**". This doesn't give an error but your code won't run correctly and terminated at this point. Type of this error is "SIGFAULT".
- If you use slot function(s) which defined in another class(es), this/these slot function(s) must be public. Because you can't access the private members of another class(es).
- Signals can be public, private or protected. It depends on to where you call the "**YSIGNAL_MULTI ()**" macro. Public signals can emit from outside of the class scope. This can be cause of undesirable situations happen.
- More examples about the using of multi signal (like slot functions at another class, nested signals etc.) are located at "examples" directory.
- The syntax of calling **template** functions in signal struct except "**emitAll**" function must be like this: `signalName->functionName<slotIndex>(…);`

b) Example:

```
class ExampleClass
{
    public:
        ExampleClass();
        char slotFunction(int);
        int  slotFunction2(int);
        void otherFunction();
        YSIGNAL_MULTI( mySignal,
                       YSLOT(char,ExampleClass,slotFunction,int),
                       YSLOT(int,ExampleClass,slotFunction2,int)
                     )
};

ExampleClass::ExampleClass()
{
    YSIGNAL_MULTI_INIT( mySignal, this, this )
}

char ExampleClass::slotFunction(int x)
{
    // do something in here like below
    return (char)x;
}

int ExampleClass::slotFunction2(int x)
{
    // do something in here like below
    return x;
}


void ExampleClass:: otherFunction()
{
    std::cout << "dereferenced return value of first slot is: "
              << *(mySignal->emit<0>(89))
              << std::endl
              << "dereferenced return value of second slot is: "
              << *(mySignal->emit<1>(89));

}

int main()
{
    ExampleClass test1;
    test1.otherFunction();
    // Output:
    // dereferenced return value of first slot is: Y
    // dereferenced return value of second slot is: 89

    return 0;
}
```

After compiler expands the "**YSIGNAL_MULTI (mySignal, slot1, slot2)**" macro, class code turns into like below. "**slot1**" means **YSLOT(**char,ExampleClass,slotFunction,int**)** and "**slot2**" means **YSLOT(**int,ExampleClass,slotFunction2,int**)** and these macros will expand also, but this is not shown below.

```
class ExampleClass
{
  public:
    ExampleClass();
    char slotFunction(int);
    int  slotFunction2(int);
    void otherFunction();


    struct SIGNAL_STRUCT_mySignal
    {
      private:
        static const unsigned int slotsCount =
std::tuple_size<decltype(std::make_tuple(slot1, slot2))>::value;

        decltype(std::make_tuple(slot1, slot2))  slotsTupple;
        std::vector<bool>                        *enableState;


      public:

        template<typename... Args>
        SIGNAL_STRUCT_mySignal(Args ...args)
        {
          slotsTupple    = std::make_tuple(slot1, slot2);
          auto receivers = std::make_tuple(args...);
          enableState    = new std::vector<bool>(slotsCount, true);

          /* Assignement of multi slot's receiver with static for loop */
          YSignalSlot_SetReceiversHelper<
                                          decltype(slotsTupple),
                                          decltype(receivers),
                                          0,
                                          slotsCount
                                         >
                                         ()(slotsTupple, receivers);
        }

        template<int ind>
        std::string& getSlotSign()
        {
          return std::get<ind>(slotsTupple)->getStringifiedSign();
        }

        unsigned int getSlotCount()
        {
          return slotsCount;
        }
```

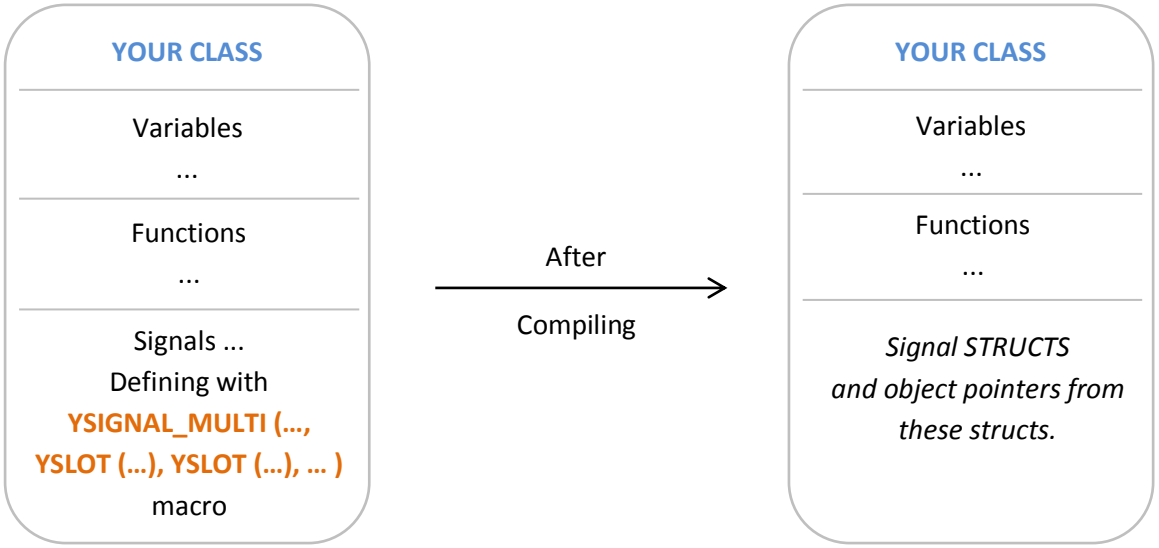
```
template<int ind>
unsigned int getArgumentCount()
{
  return std::get<ind>(slotsTupple)->getArgumentCount();
}


template<int ind>
auto getReceiver()
-> decltype( std::get<ind>(slotsTupple)->getReceiver())
{
  return std::get<ind>(slotsTupple)->getReceiver();
}


bool setEnable(unsigned int ind)
{
  if( ind > slotsCount-1 ) { return false; }

  enableState->at(ind) = true;
  return true;
}


bool setDisable(unsigned int ind)
{
  if( ind > slotsCount-1 ) { return false; }

  enableState->at(ind) = false;
  return true;
}


bool isEnabled(unsigned int ind)
{
  if( ind > slotsCount-1 ) { return false; }
  return enableState->at(ind);
}


template<typename... Args>
void emitAll(Args ...args)
{

  /* Emitting of all enabled slots with static for loop */
  YSignalSlot_EmitAllHelper<
                            decltype(slotsTupple),
                            0,
                            slotsCount,
                            Args...
                           >
                           ()(slotsTupple, *enableState, args...);
}
```

```
        template<int ind, typename... Args>
        auto emit(Args ...args)
        -> decltype( (std::get<ind>(slotsTupple)->getReceiver()-
>*(std::get<ind>(slotsTupple)->getSign()))(args...))*
        {
          if( enableState->at(ind) )
          {

            /* Emitting of single enabled slot with static if */
            /* Static if is used for deducting of slot function
               return type is void or not */
            return YSignalSlot_EmitFuncHelper
                                <
                                 std::is_void<typename
YSIGNALSLOT_TYPEOFPOINTER(std::get<ind>(slotsTupple))::returnType>::value,
                                 decltype(std::get<ind>(slotsTupple)),
                                 Args...
                                >
                                ()(std::get<ind>(slotsTupple), args...);
          }
          else
          {
            typename
YSIGNALSLOT_TYPEOFPOINTER(std::get<ind>(slotsTupple))::returnType *ret =
NULL
            return ret;
          }
        }

    };

    SIGNAL_STRUCT_mySignal *mySignal;

};
```

And the compiler expands also **YSIGNAL_MULTI_INIT (** mySignal, this, this **)** like below.

```
ExampleClass::ExampleClass()
{
    mySignal = new SIGNAL_STRUCT_mySignal(this, this);
}
```

c) Diagram:


YOUR CLASS
Variables
...
Functions
...
Signals ...
Defining with
YSIGNAL_MULTI (…,
YSLOT (…), YSLOT (…), …)
macro
After
Compiling
YOUR CLASS
Variables
...
Functions
...
Signal STRUCTS
and object pointers from
these structs.

Header File Information:

Filename:	YSignalSlot.h
File size:	30.227 byte
Code size:	460 lines
Classes:	• YSignalSlot_SlotFunction
Structs:	• YSignalSlot_SetReceiversHelper • YSignalSlot_EmitAllHelper • YSignalSlot_EmitFuncHelper
Macros:	• YSIGNALSLOT_TYPEOFPOINTER • YSIGNALSLOT_STRINGIFY_FUNCTION_SIGN_HELPER • YSIGNALSLOT_STRINGIFY_FUNCTION_SIGN • YSLOT • YSIGNAL_MULTI • YSIGNAL_MULTI_INIT • YSIGNAL • YSIGNAL_INIT