@odinthenerd

– not the god

Hana Dusíková

compile time regex

Mixins


ARMZ
STM32L053
R8T6
GH24E VG
CHN GH 528

Bare metal drivers

-
-
-
-
-
-

Bare metal drivers

- In assembler
-
-
-
-
-

Bare metal drivers

- In assembler
-
-
-
-
-

Bare metal drivers

- In assembler
- In C with Macros
-
-
-
-

Bare metal drivers

- In assembler
- In C with Macros
-
-
-
-

Bare metal drivers

- In assembler
- In C with Macros
- By hand in C++
-
-
-

Bare metal drivers

- In assembler
- In C with Macros
- By hand in C++
-
-
-

Bare metal drivers

- In assembler
- In C with Macros
- By hand in C++
- Aggregation
-
-

Bare metal drivers

- In assembler
- In C with Macros
- By hand in C++
- Aggregation
-
-

Bare metal drivers

- In assembler
- In C with Macros
- By hand in C++
- Aggregation
- Inheritance + virtual functions
-

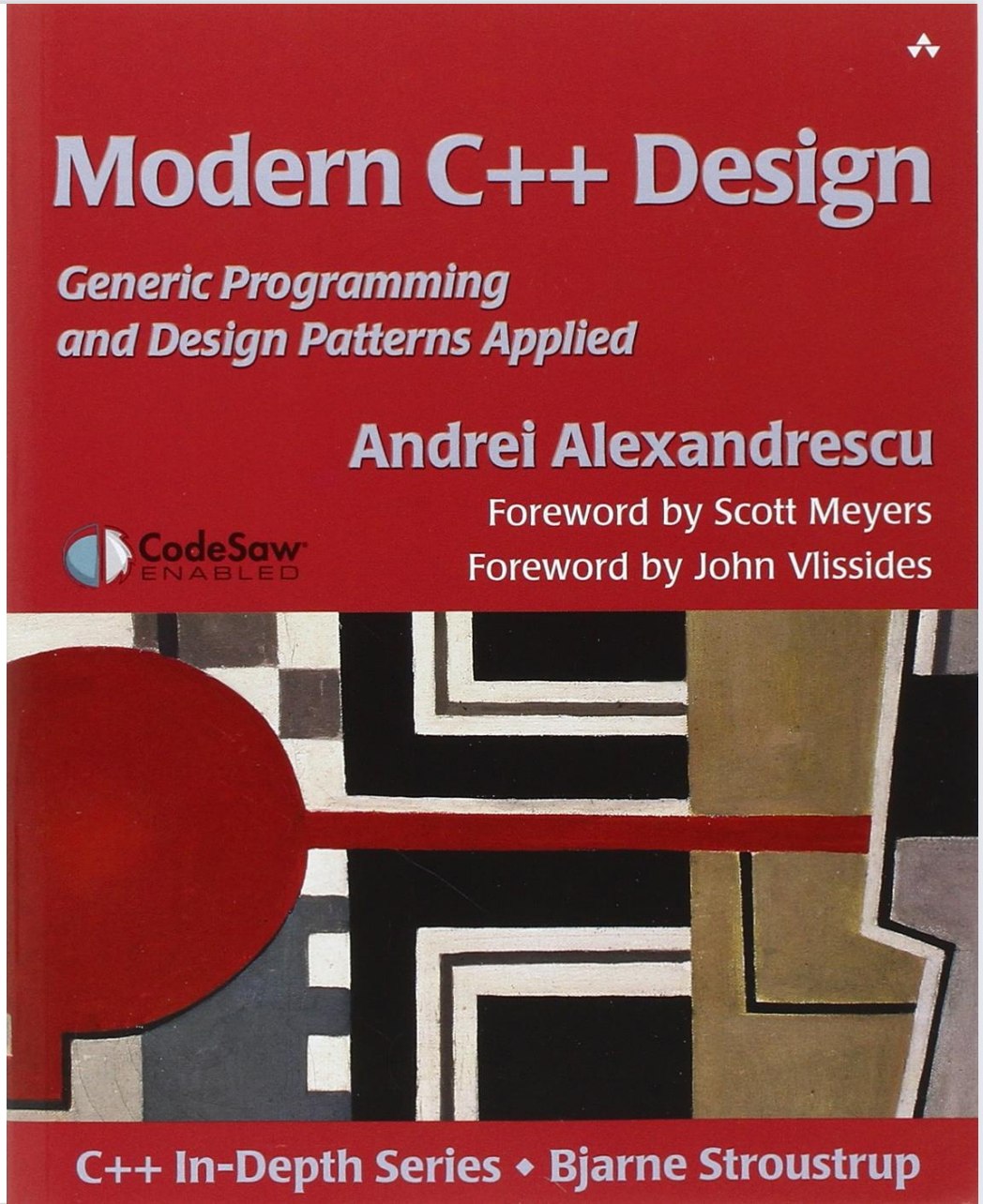
Bare metal drivers

- In assembler
- In C with Macros
- By hand in C++
- Aggregation
- Inheritance + virtual functions
-

Bare metal drivers

- In assembler
- In C with Macros
- By hand in C++
- Aggregation
- Inheritance + virtual functions
- Inheritance + CRTP

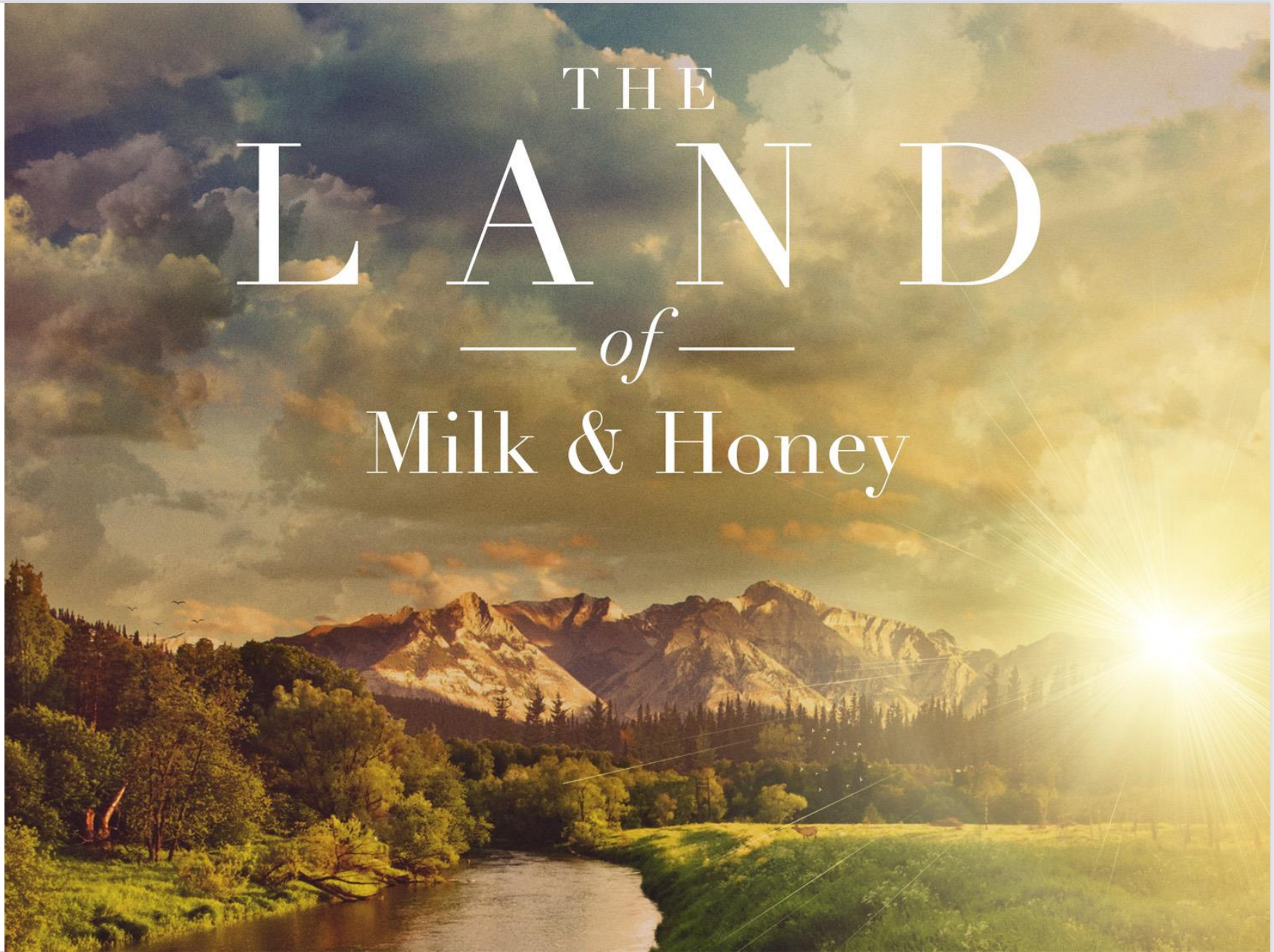
Inspiration in old book form



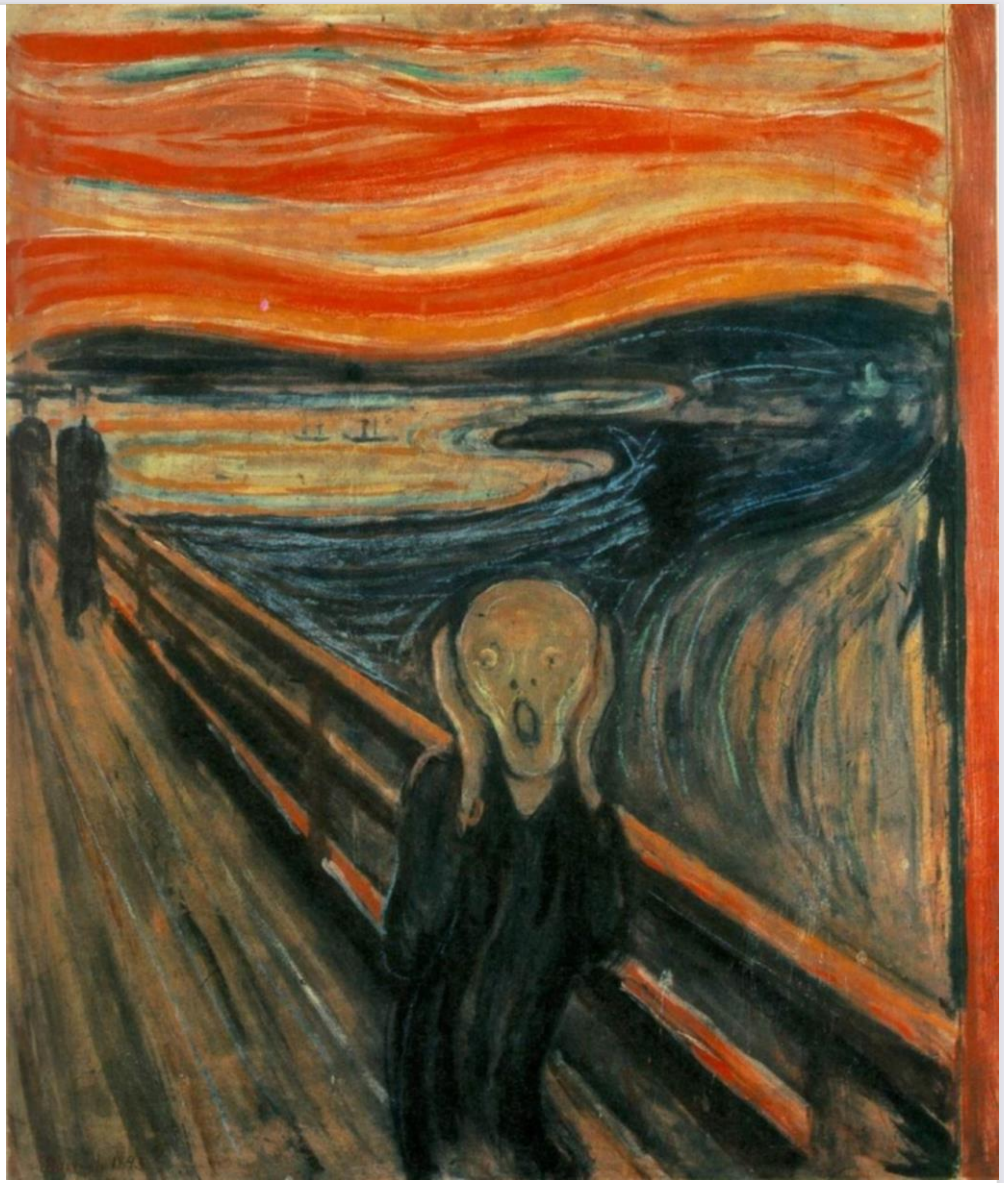
Bare metal drivers

- In assembler
- In C with Macros
- By hand in C++
- Aggregation
- Inheritance + virtual functions
- Inheritance + CRTP

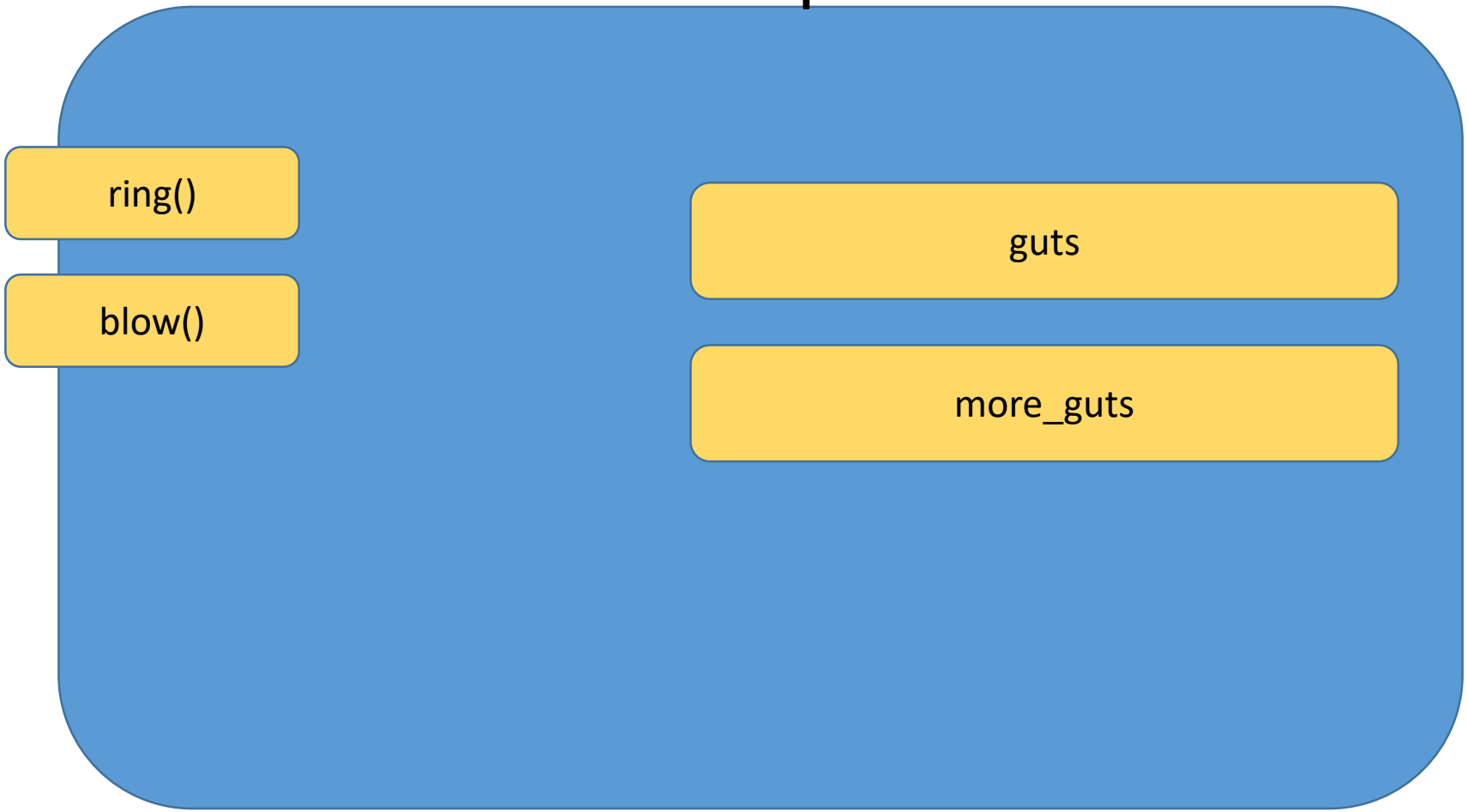
Even your cat will like you!



Implementation details



Mixin composition



Code example

```
auto thing = mixin::compose(
    mixin::interface<bells, whistles>,
    guts,
    more_guts);
```

Code example

```
auto thing = mixin::compose(
    mixin::interface<bells, whistles>,
    guts,
    more_guts);
```

Code example

```
auto thing = mixin::compose(
    mixin::interface<bells, whistles>,
    guts,
    more_guts);
```

Code example

```
auto thing = mixin::compose(
    mixin::interface<bells, whistles>,
    guts{haggis},
    more_guts);
```

Code example

```
auto thing = mixin::compose(
    mixin::interface<bells, whistles>,
    guts{haggis},
    more_guts);

thing.ring();
```

Code example

```
auto thing = mixin::compose(
    mixin::interface<bells, whistles>,
    guts{haggis},
    more_guts,
    my_allocator{arena});
```

Definition of terms

- Composition

Mixin composition

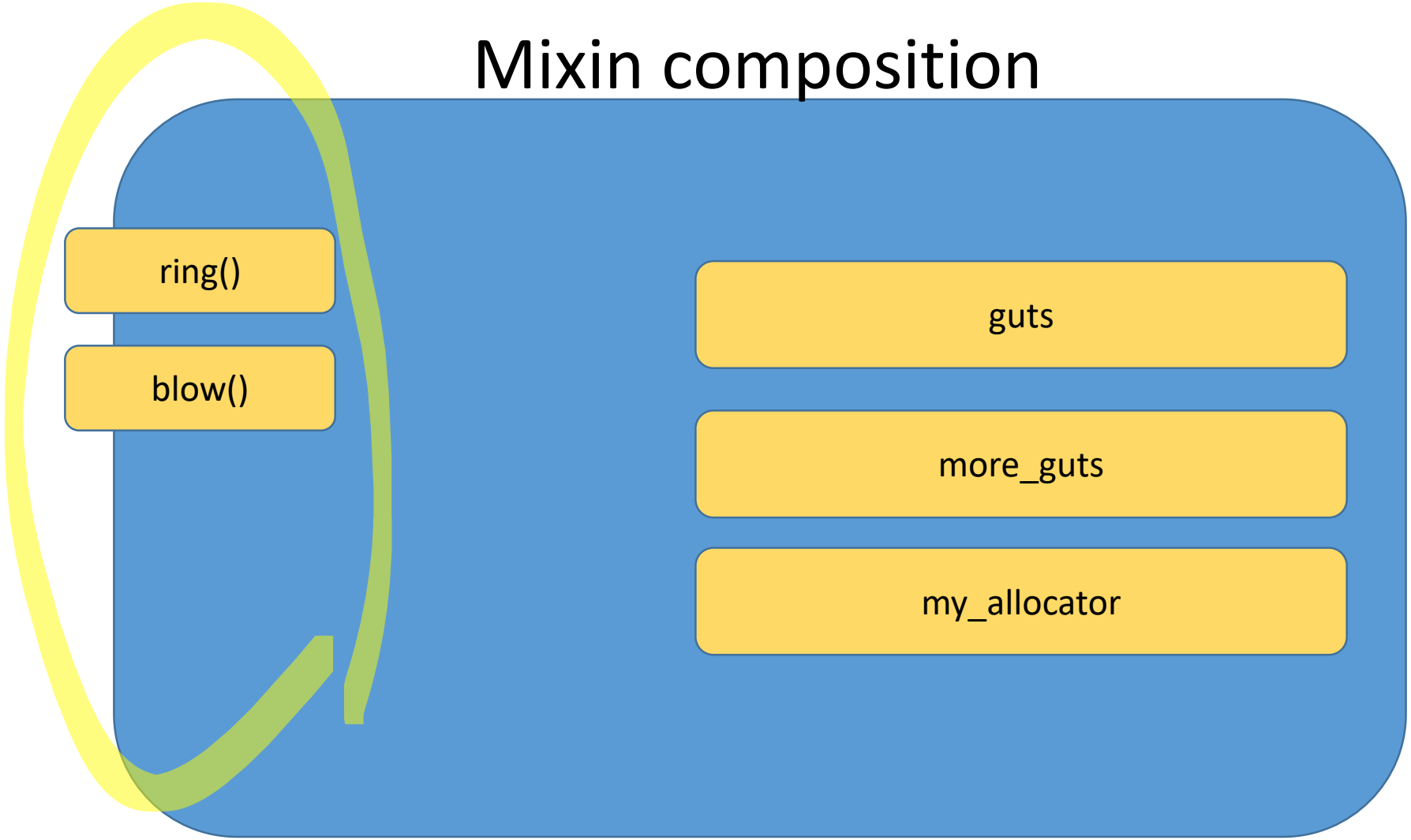
Composition

```
auto thing = mixin::compose(
    mixin::interface<bells, whistles>,
    guts{haggis},
    more_guts,
    my_allocator{arena});
```

Definition of terms

- Composition
- Interface / Implementation

Mixin composition



Interface

```
template<typename T>
struct bells : T {
    void ring();
};
```

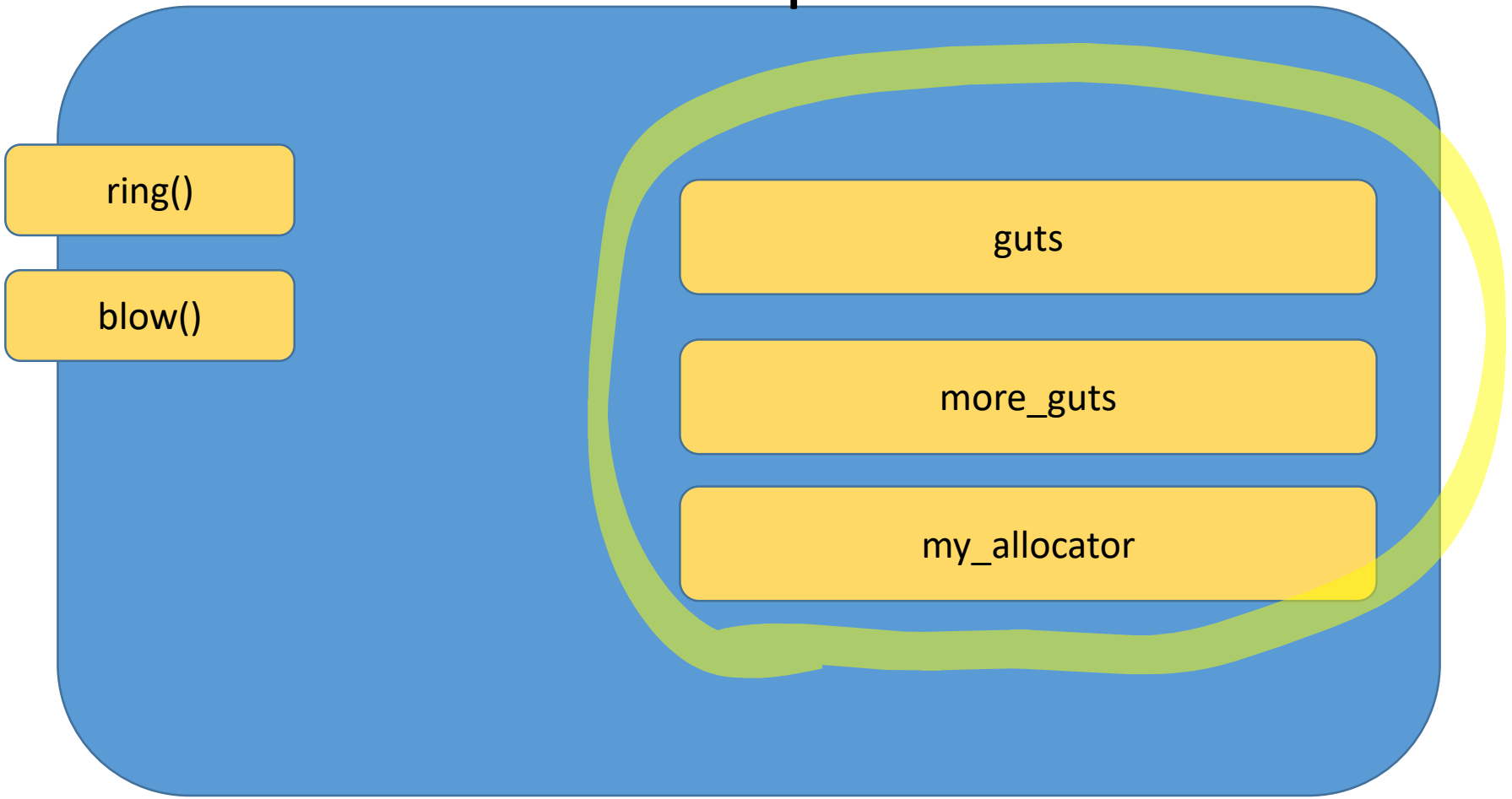
Interface – concept

```
template<typename T>
struct bells : T {
    void ring();
};
```

Interface – adds to the composition objects interface

```
template<typename T>
struct bells : T {
    void ring();
};
```

Mixin composition



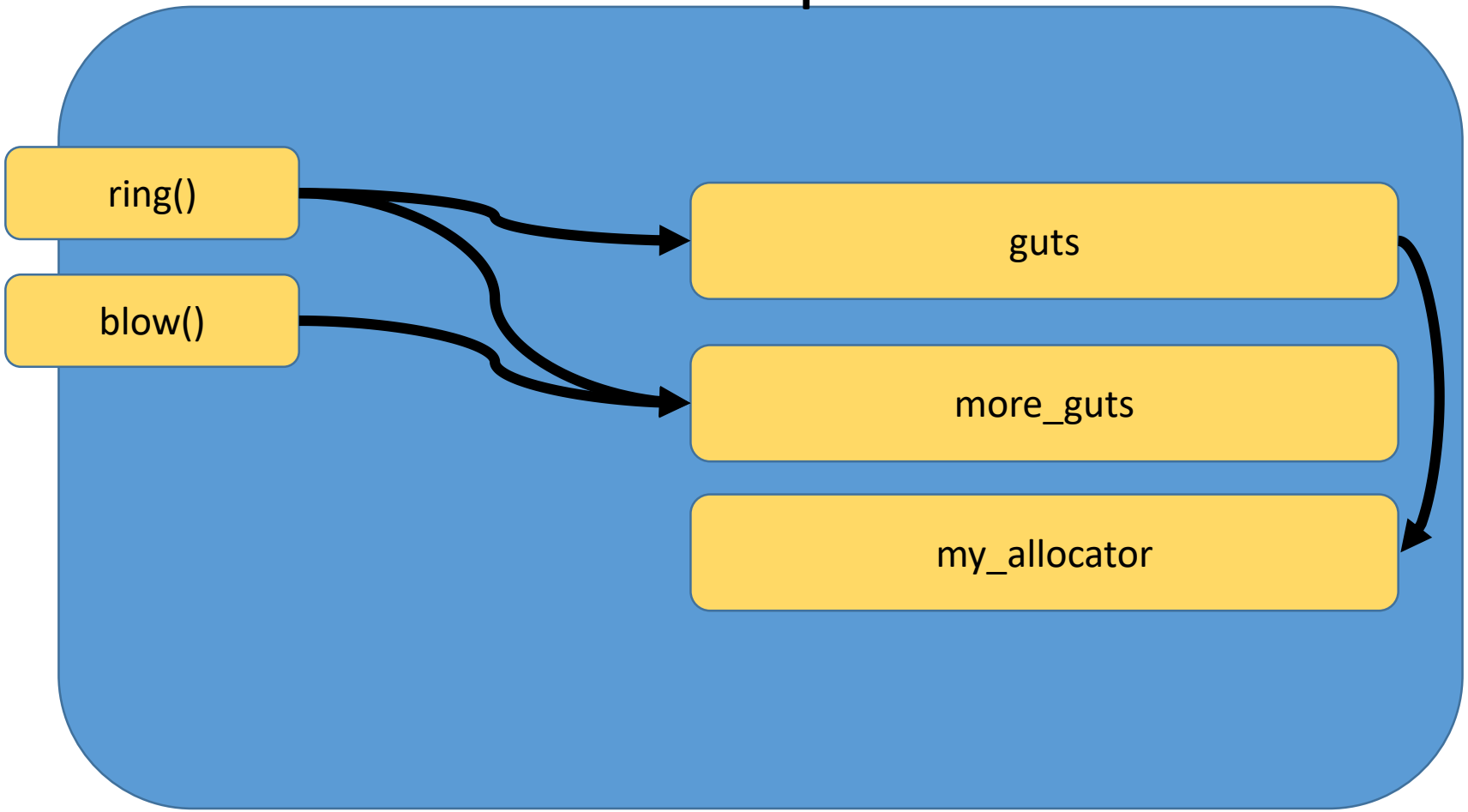
Implementation

```
struct guts {

};
```

Definition of terms

- Composition
- Interface / Implementation
- Abilities

Mixin composition



Abilities

```
struct ringable{};
```

Find mixins by ability

```
template<typename T>
struct bells : T{
    void ring(){
        for_each(this, ability<ringable>, [](auto& a){
            a.ring();
        });
    }
};
```

Associating abilities with a mixin

```
using guts = make_mixin<
    guts_impl,
    ringable,
    magic_frog_power,
    allocator_use_capable>;
```

Definition of terms

- Composition
- Interface / Implementation
- Abilities
- Requirements

Find mixins by ability 0-n

```
template<typename T>
struct bells : T{
    void ring(){
        for_each(this, ability<ringable>, [](auto& a){
            a.ring();
        });
    }
};
```

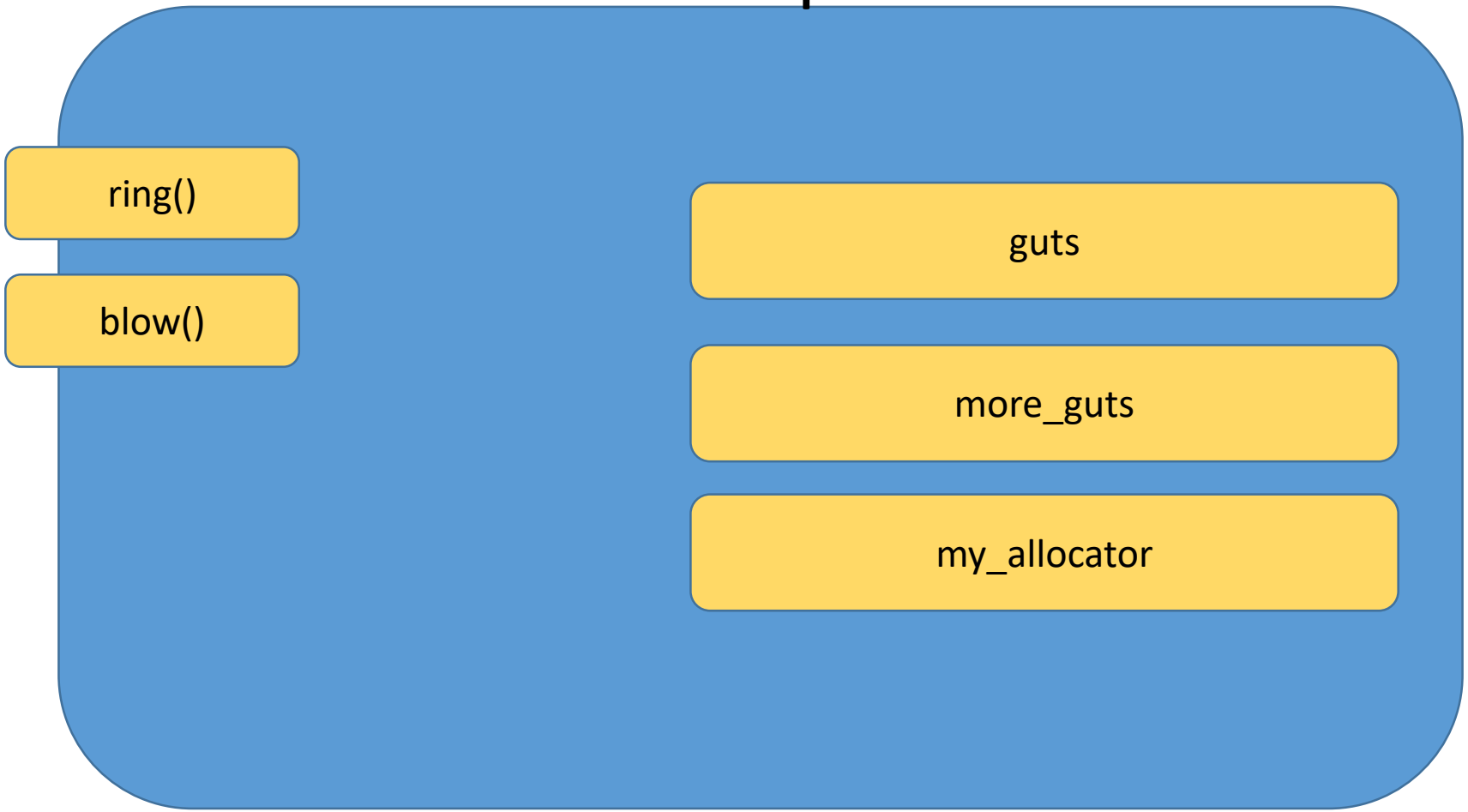
Find mixins by ability exactly 1

```
template<typename T>
struct bells : T{
    void ring(){
        execute(this, ability<ringable>, [](auto& a){
            a.ring();
        });
    }
};
```

Find mixins that match an arbitrary predicate

```
template<typename T>
struct bells : T{
    void ring(){
        call_on(this, predicate<my_selector>, [](auto& a){
            a.ring();
        });
    }
};
```

Mixin composition



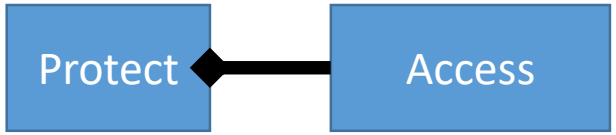
Return type of compose()

```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<Ts...> data;
  //…

};
```

Return type of compose()

```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<Ts...> data;
  //…

};
```

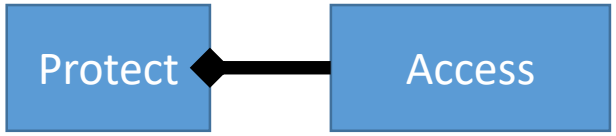
```
protect<access<composition<Ts…>>>
```



Protect

```
template<typename T>
struct protect : protected T {};
```

```
protect<access<composition<Ts…>>>
```



Return type of compose()

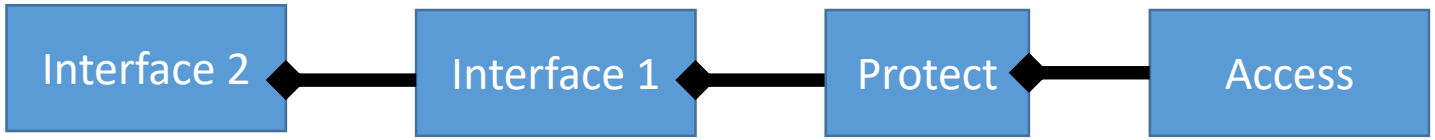
```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<Ts...> data;
  //…

};
```

```
interface1<protect<access<composition<Ts…>>>>
```

Return type of compose()

```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<Ts...> data;
  //…

};
```

```
interface2<interface1<protect<access<composition<Ts…>>>>>
```



Return type of compose()

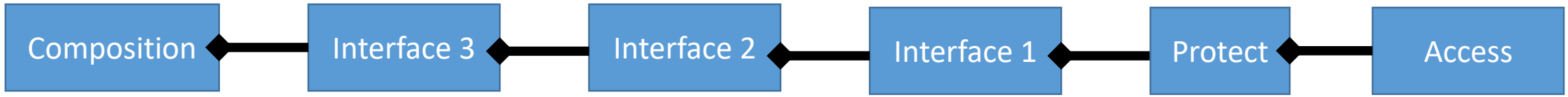
```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<Ts...> data;
  //…

};
```

```
interface3<interface2<interface1<protect<access<composition<Ts…>>>>>>
```

Return type of compose()

```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<Ts...> data;
  //…

};
```

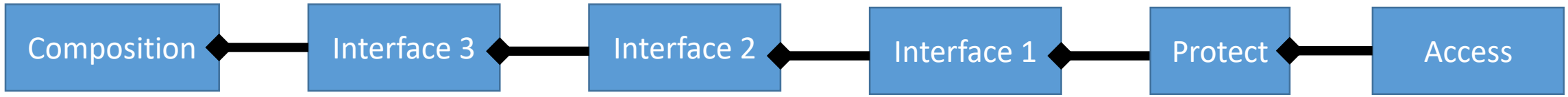
```
interface3<interface2<interface1<protect<access<composition<Ts…>>>>>>
```



Completing the circle

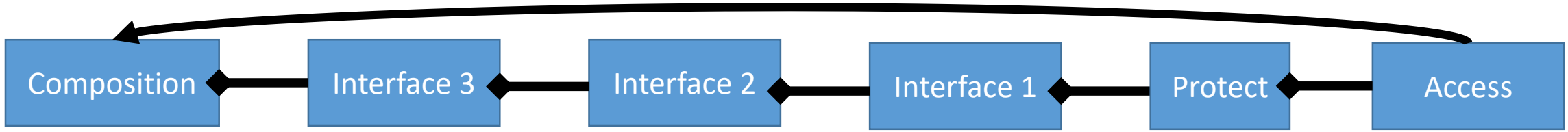
```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<Ts...> data;
  //…

};
```

```
interface3<interface2<interface1<protect<access<composition<Ts…>>>>>>
```



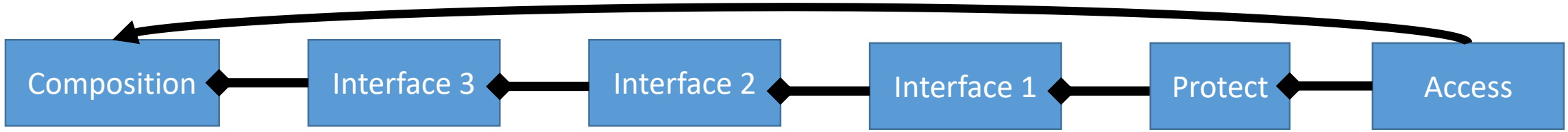
Access

```
template<typename T>
struct access {
    auto& get_data(){
        return static_cast<T*>(this)->data;
    }
};
```



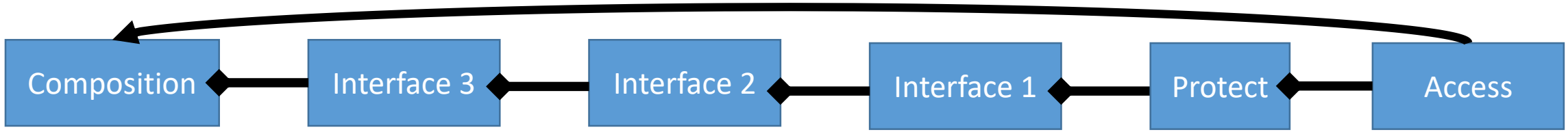
Encapsulation

```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<Ts...> data;
  friend access<composition<Ts...>>;
  //…

};
```



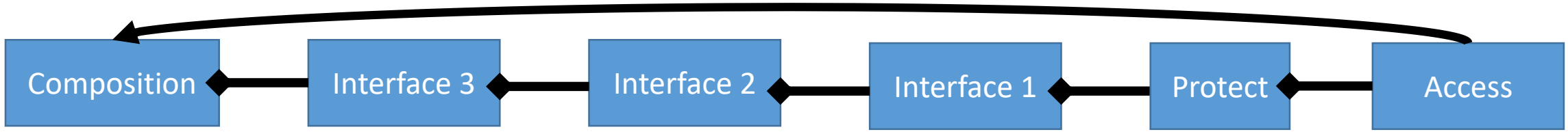
Public interface call

```
template<typename T>
struct bells : T{
    void ring(){
        for_each(this, ability<ringable>, [](auto& m){
            m.ring();
        });
    }
};
```

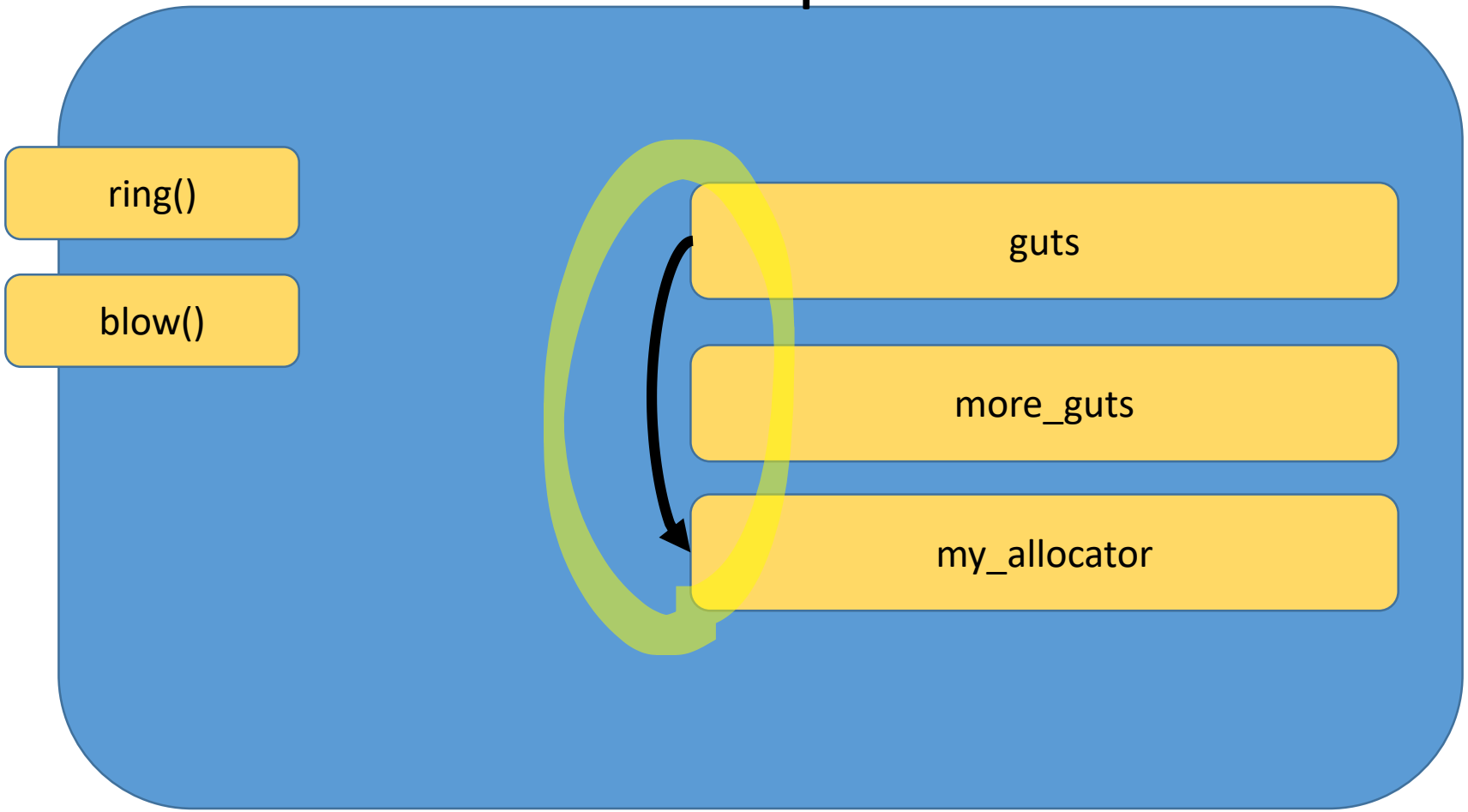


for_each

```
template<typename T, typename A, typename L>
void for_each(access<T>* p, A, L l){
    auto& data = p->get_data();
    //magic here
}
```



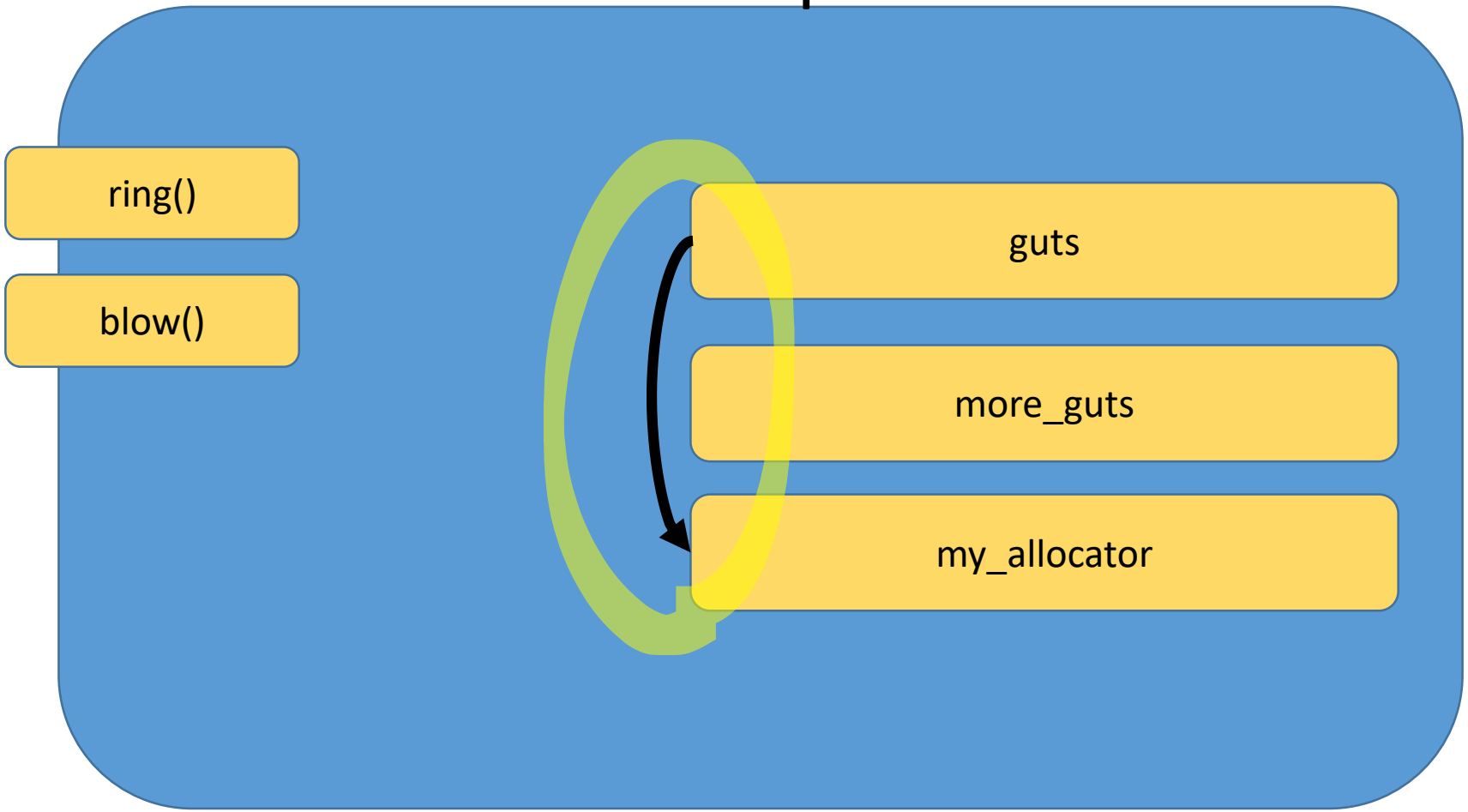
Mixin composition



Inter implementation mixin access

```
template<typename T>
struct bells : T{
    void ring(){
        for_each(this, ability<ringable>,
            [a = access_to(this)](auto& m){
                m.ring(a);
            }
        );
    }
};
```

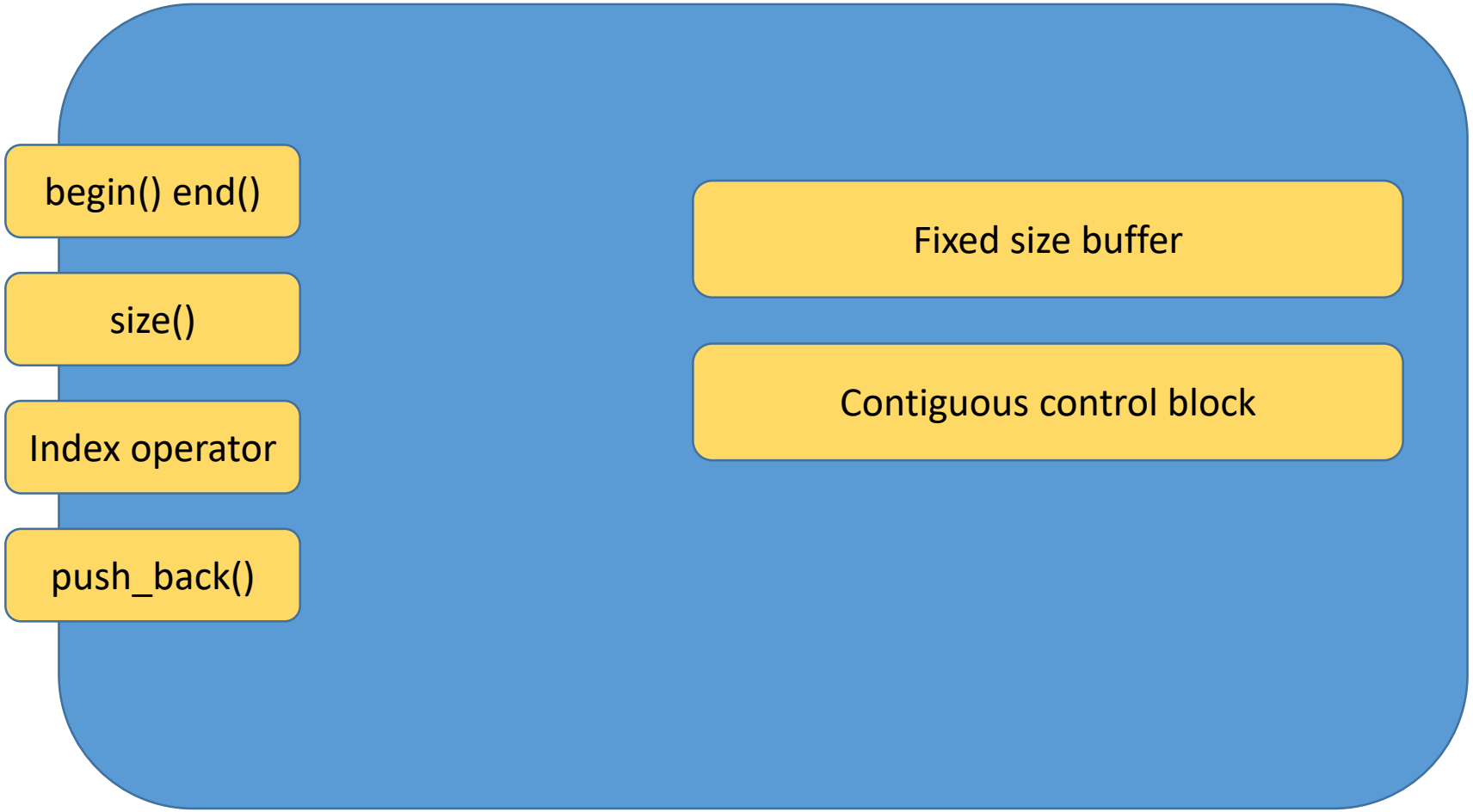
Mixin composition



Init and Destruct

```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  //…
public:
  composition(std::tuple<Ts...> &&d) : data{std::move(d)} {
    for_each(this,ability<requires_init_and_destruct>,
        detail::call_init(this));
  }
  ~composition() {
    for_each(this,ability<requires_init_and_destruct>,
        detail::call_destruct(this));
  }
};
```

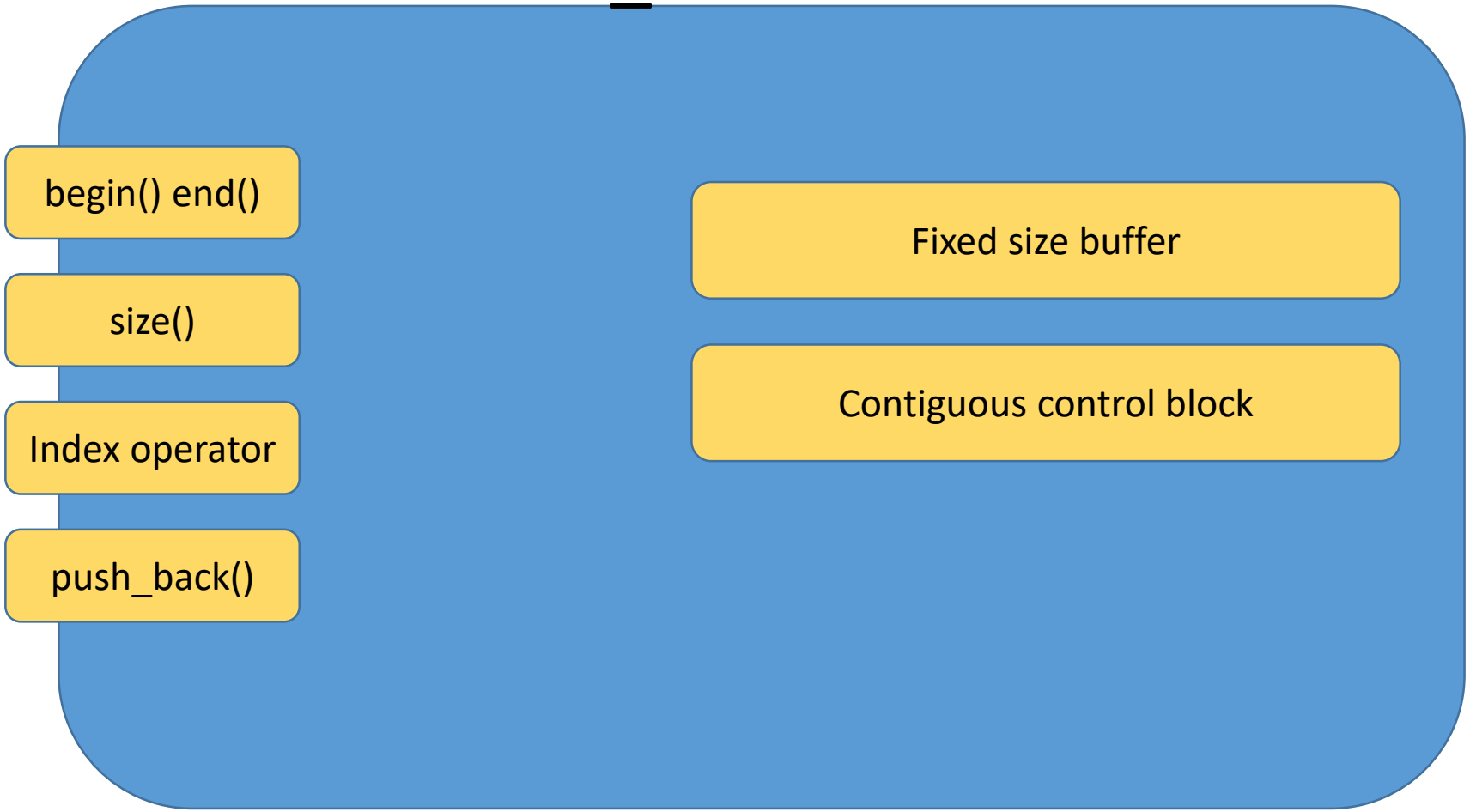
Can we build a fixed_vector?



Can we build fixed_vector?

- Data footprint dependent on other mixins
-
-
-
-

fixed_vector



Dynamic mixins

```
using guts = make_dynamic_mixin<
    fixed_buffer_factory,
    allocator>;
```

Return type of compose()

```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<Ts...> data;
  //…

};
```

Return type of compose()

```
template<typename... Ts>
class composition:public call_<detail::make_base<composition<Ts...>>, Ts...>
{
  std::tuple<call_<Ts, Ts...>...> data;
  //…

};
```

Can we build fixed_vector

- Data footprint dependent on other mixins
- Debug builds are bloated
-
-
-

Can we build fixed_vector

- Data footprint dependent on other mixins
- Debug builds are bloated
- Iterator validity contracts are hard
-
-

Can we build fixed_vector

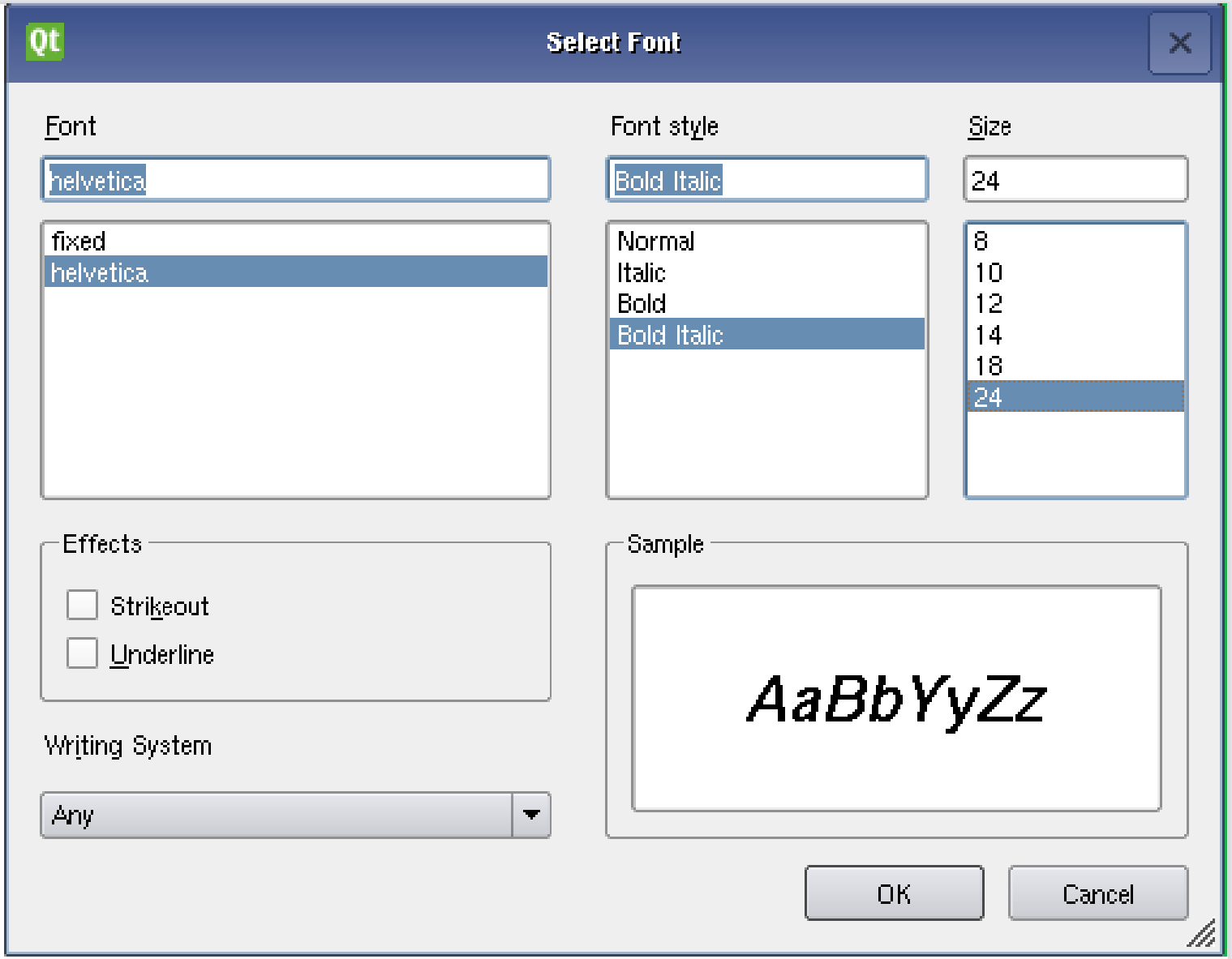
- Data footprint dependent on other mixins
- Debug builds are bloated
- Iterator validity contracts are hard
- Testing is hard
-

Can we build fixed_vector

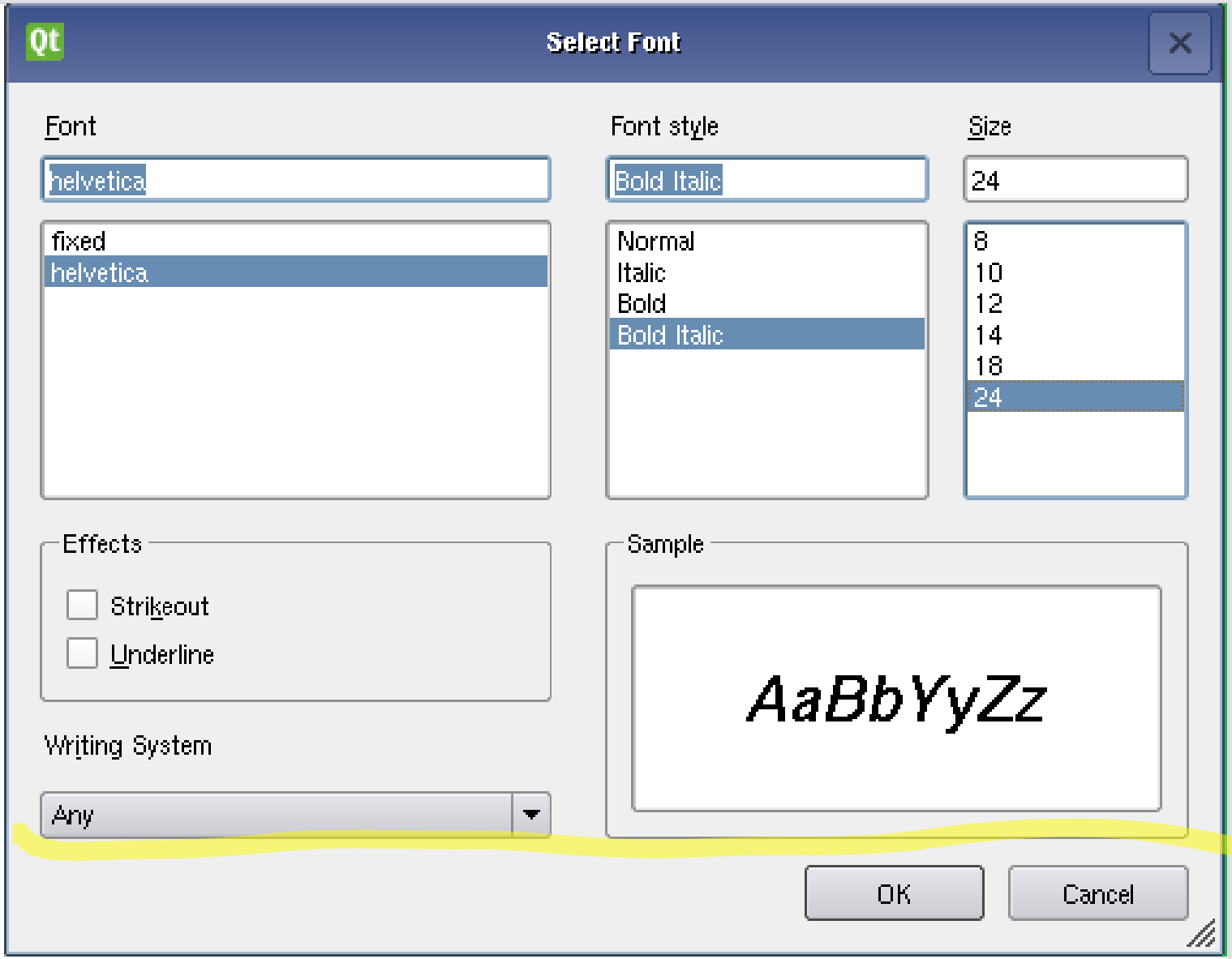
- Data footprint dependent on other mixins
- Debug builds are bloated
- Iterator validity contracts are hard
- Testing is hard
- Constructors are hard

Can we build industrial strength GUIs?

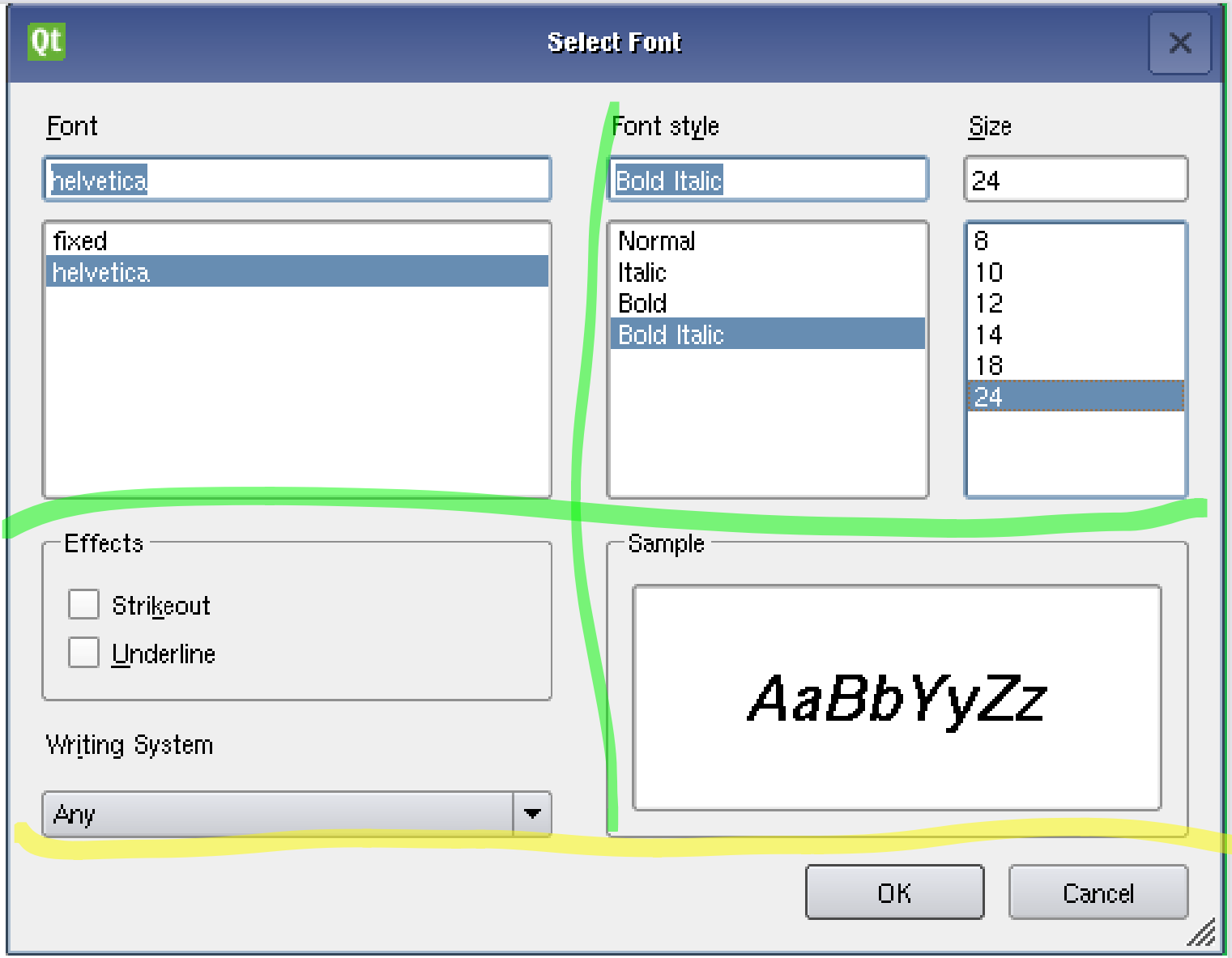
Qt


Select Font
Font
helvetica
fixed
helvetica
Font style
Bold Italic
Normal
Italic
Bold
Bold Italic
Size
24
8
10
12
14
18
24
Effects
Strikeout
Underline
Sample
AaBbYyZz
Writing System
Any
OK
Cancel

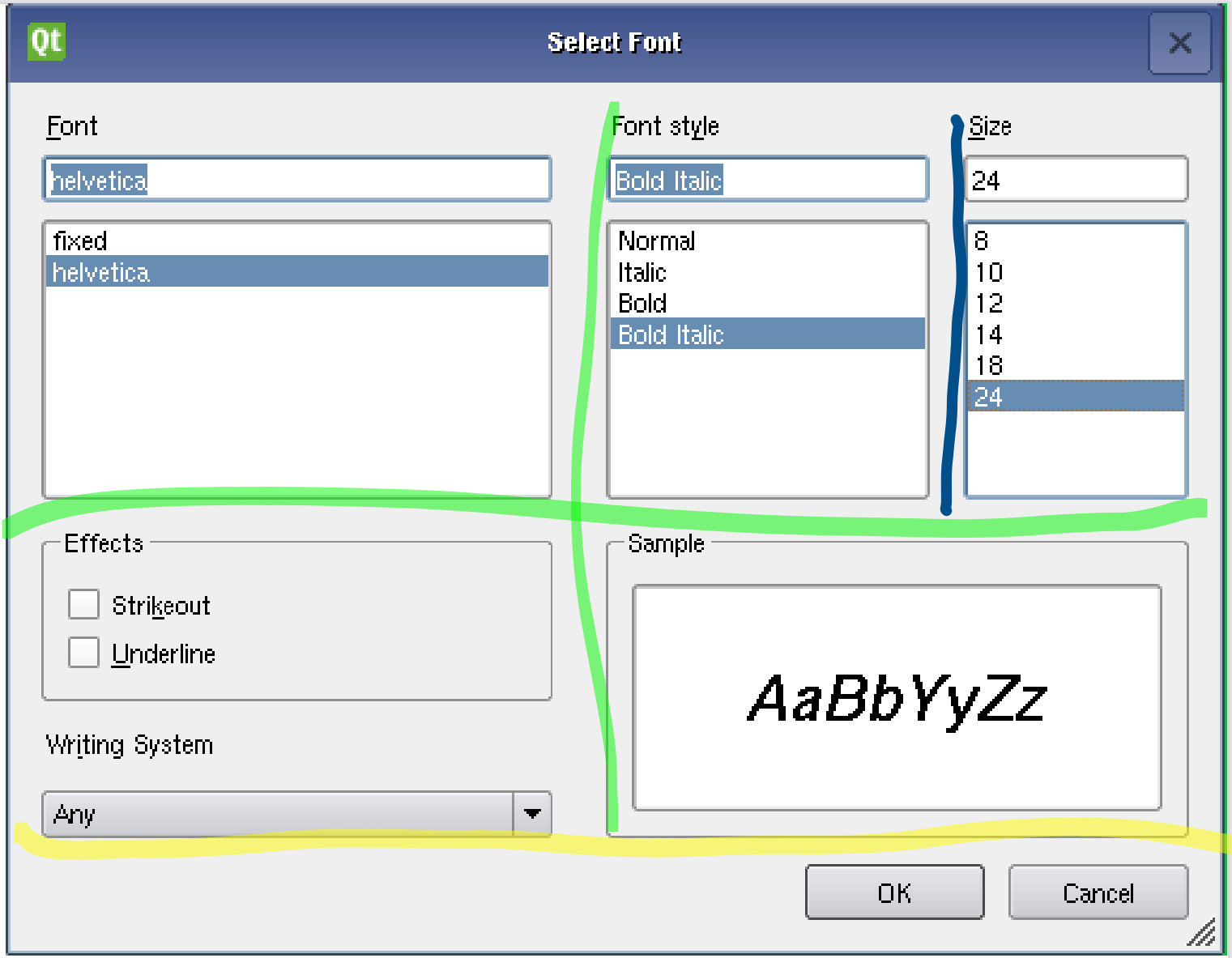
Qt


Select Font
Font
helvetica
fixed
helvetica
Font style
Bold Italic
Normal
Italic
Bold
Bold Italic
Size
24
8
10
12
14
18
24
Effects
Strikeout
Underline
Sample
AaBbYyZz
Writing System
Any
OK
Cancel

Qt


Select Font
Font
helvetica
fixed
helvetica
Font style
Bold Italic
Normal
Italic
Bold
Bold Italic
Size
24
8
10
12
14
18
24
Effects
Strikeout
Underline
Sample
AaBbYyZz
Writing System
Any
OK
Cancel

Qt


Select Font
Font
helvetica
fixed
helvetica
Font style
Bold Italic
Normal
Italic
Bold
Bold Italic
Size
24
8
10
12
14
18
24
Effects
Strikeout
Underline
Sample
AaBbYyZz
Writing System
Any
OK
Cancel

Statically linking Widgets

```
auto font_dialog = dialog(
    v_box_layout(
        grid_layout(
            dimentions(2_c, 2_c),
                        //bunch of widgets
        ),
        h_box_layout(
            h_stretch,
            push_button("Ok",ok_action),
            push_button("Cancel",cancel_action)
        )
    )
);
```

Statically linking Widgets

```
auto font_dialog = dialog(
    v_box_layout(
        grid_layout(
            dimentions(2_c, 2_c),
                        //bunch of widgets
        ),
        h_box_layout(
            h_stretch,
            push_button("Ok",ok_action),
            push_button("Cancel",cancel_action)
        )
    )
);
```

Statically linking Widgets

```
auto font_dialog = dialog(
    v_box_layout(
        grid_layout(
            dimentions(2_c, 2_c),
                        //bunch of widgets
        ),
        h_box_layout(
            h_stretch,
            push_button("Ok",ok_action),
            push_button("Cancel",cancel_action)
        )
    )
);
```

Statically linking Widgets

```
template<typename...Ts>
auto v_box_layout(Ts...args){
    return compose(ability<widget_event_subscribe>,
                   interface<widget_interface>,
                   widget_event_forward_to_children{},
                   drawable_v_box{},
                   args...
        );
}
```

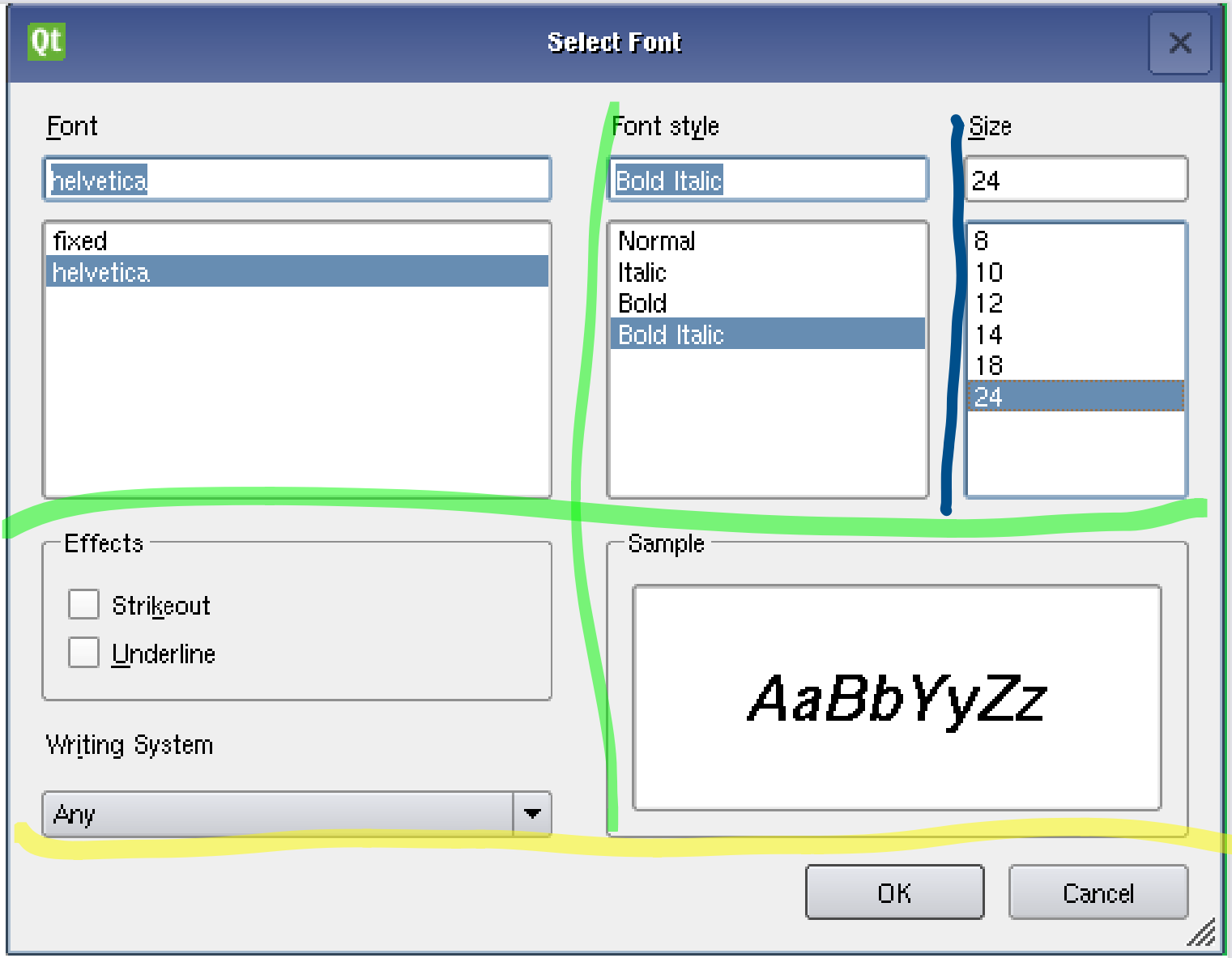
Statically linking Widgets

```
template<typename...Ts>
auto v_box_layout(Ts...args){
    return compose(ability<widget_event_subscribe>,
                   interface<widget_interface>,
                   widget_event_forward_to_children{},
                   drawable_v_box{},
                   args...
        );
}
```

Statically linking Widgets

```
template<typename...Ts>
auto v_box_layout(Ts...args){
    return compose(ability<widget_event_subscribe>,
                   interface<widget_interface>,
                   widget_event_forward_to_children{},
                   drawable_v_box{},
                   args...
           );
}
```

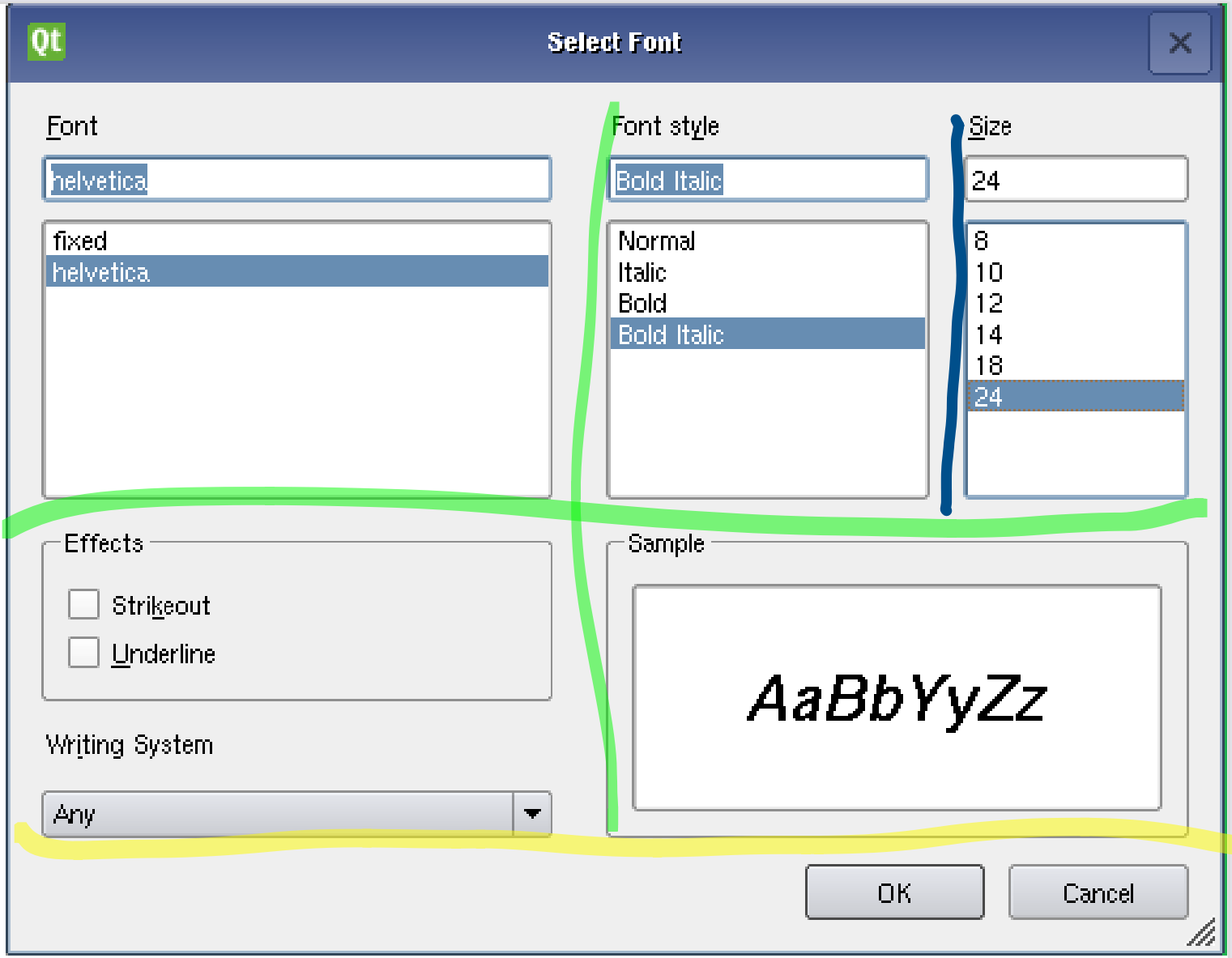
Qt


Select Font
Font
helvetica
fixed
helvetica
Font style
Bold Italic
Normal
Italic
Bold
Bold Italic
Size
24
8
10
12
14
18
24
Effects
Strikeout
Underline
Sample
AaBbYyZz
Writing System
Any
OK
Cancel

Event dispatch

```
template<typename B>
struct widget_interface : B{
  template<typename E>
  auto dispatch_event(E& e){
    return for_each(this,ability<widget_event_subscribe>, gather<E>,
      [a = access_to(this),&](auto& m){m.dispatch_event(e,a);});
  }
  template<typename E, typename A>
  auto dispatch_event(E& e, A a){
    return for_each(this,ability<widget_event_subscribe>, gather<E>,
      [&](auto& m){m.dispatch_event(e,a);});
  }
};
```

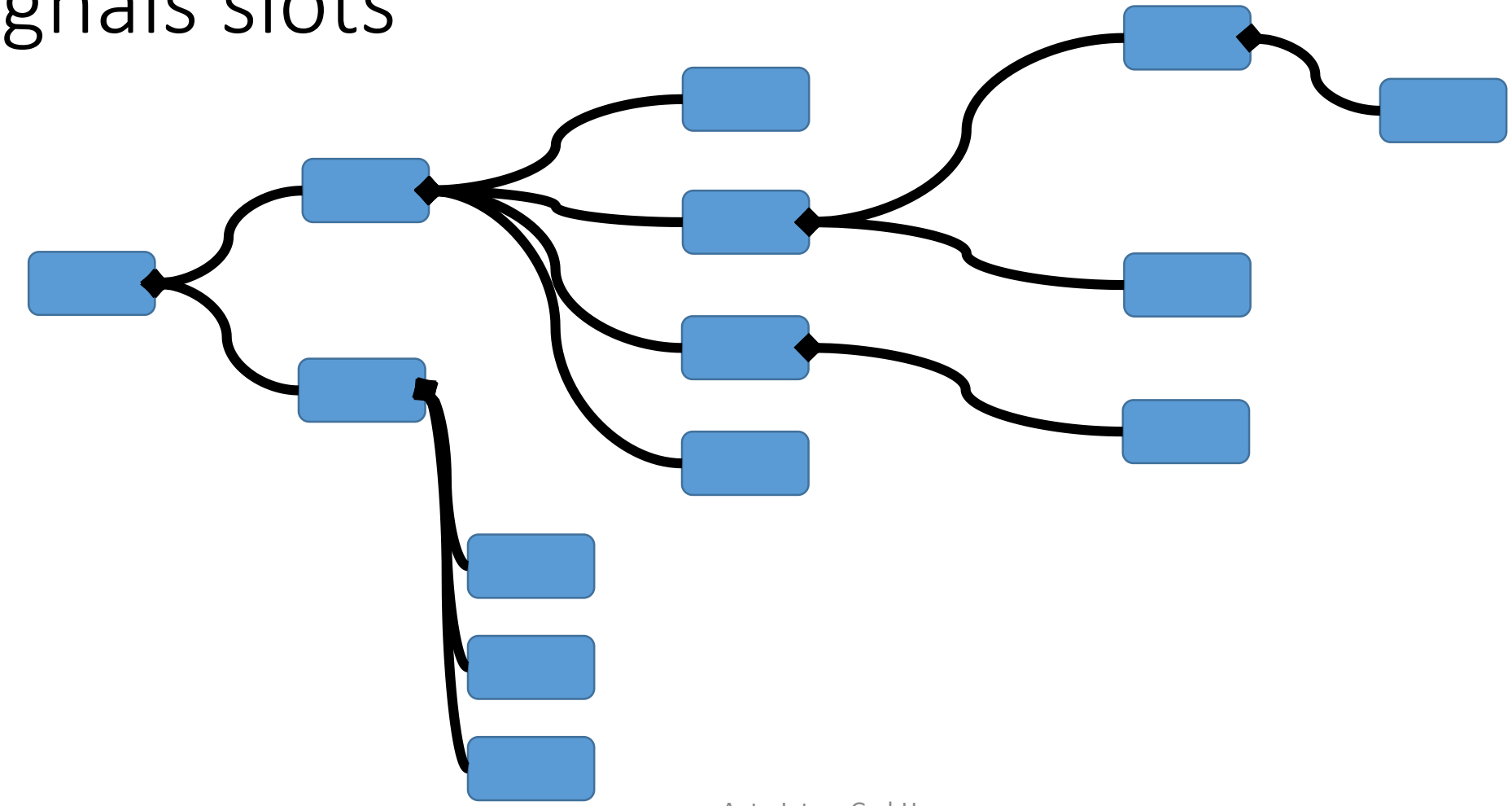
Qt


Select Font
Font
helvetica
fixed
helvetica
Font style
Bold Italic
Normal
Italic
Bold
Bold Italic
Size
24
8
10
12
14
18
24
Effects
Strikeout
Underline
Sample
AaBbYyZz
Writing System
Any
OK
Cancel

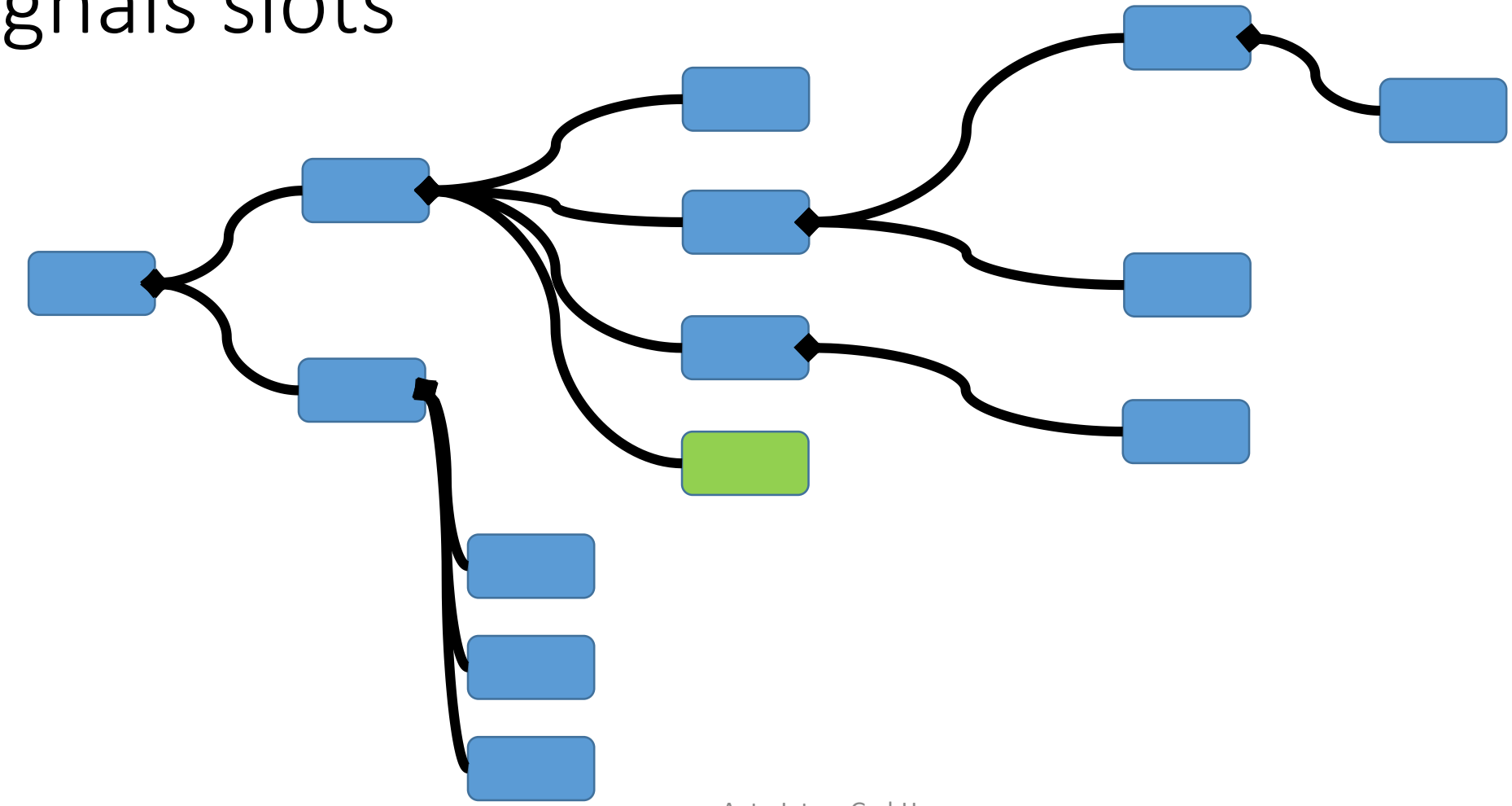
Signals slots

```
template<typename T>
struct on_keypress{
  template<typename A, typename R>
  auto consume_event(keypress_event e, A a, R root){
    //...
    root.dispatch_event(
       signal_event(signal_name<T>,a));
  }
};
```

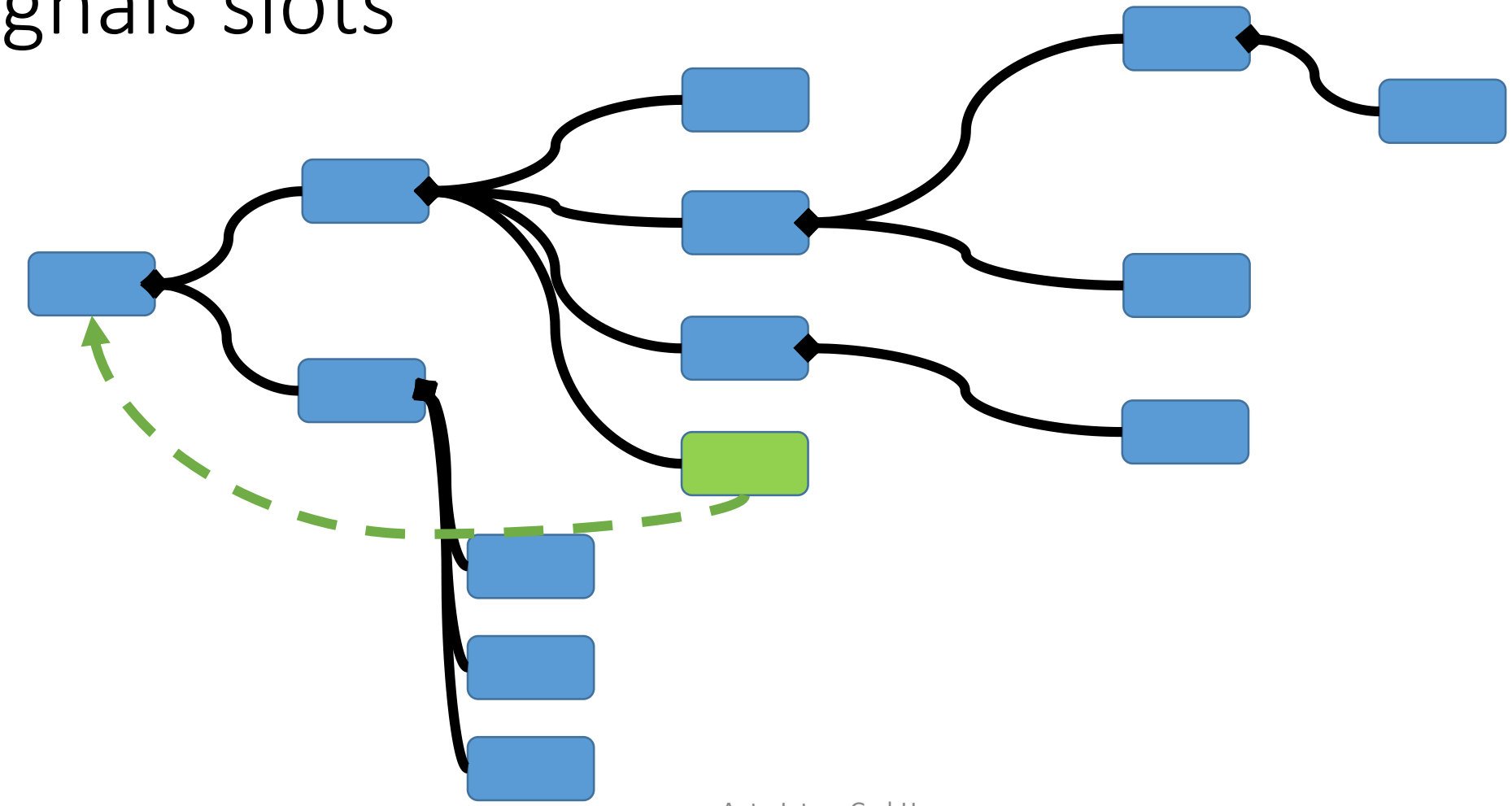
Signals slots



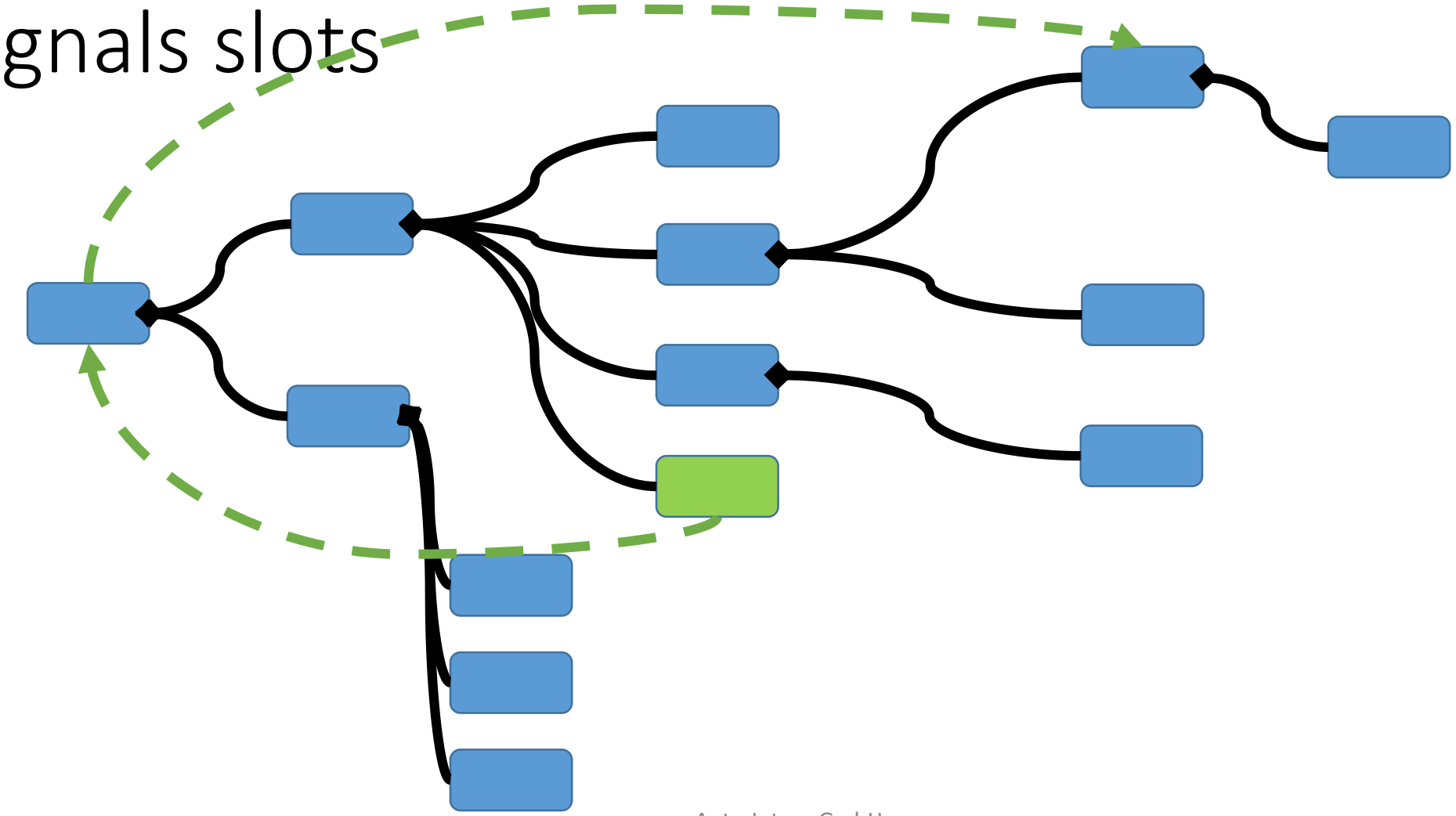
Signals slots



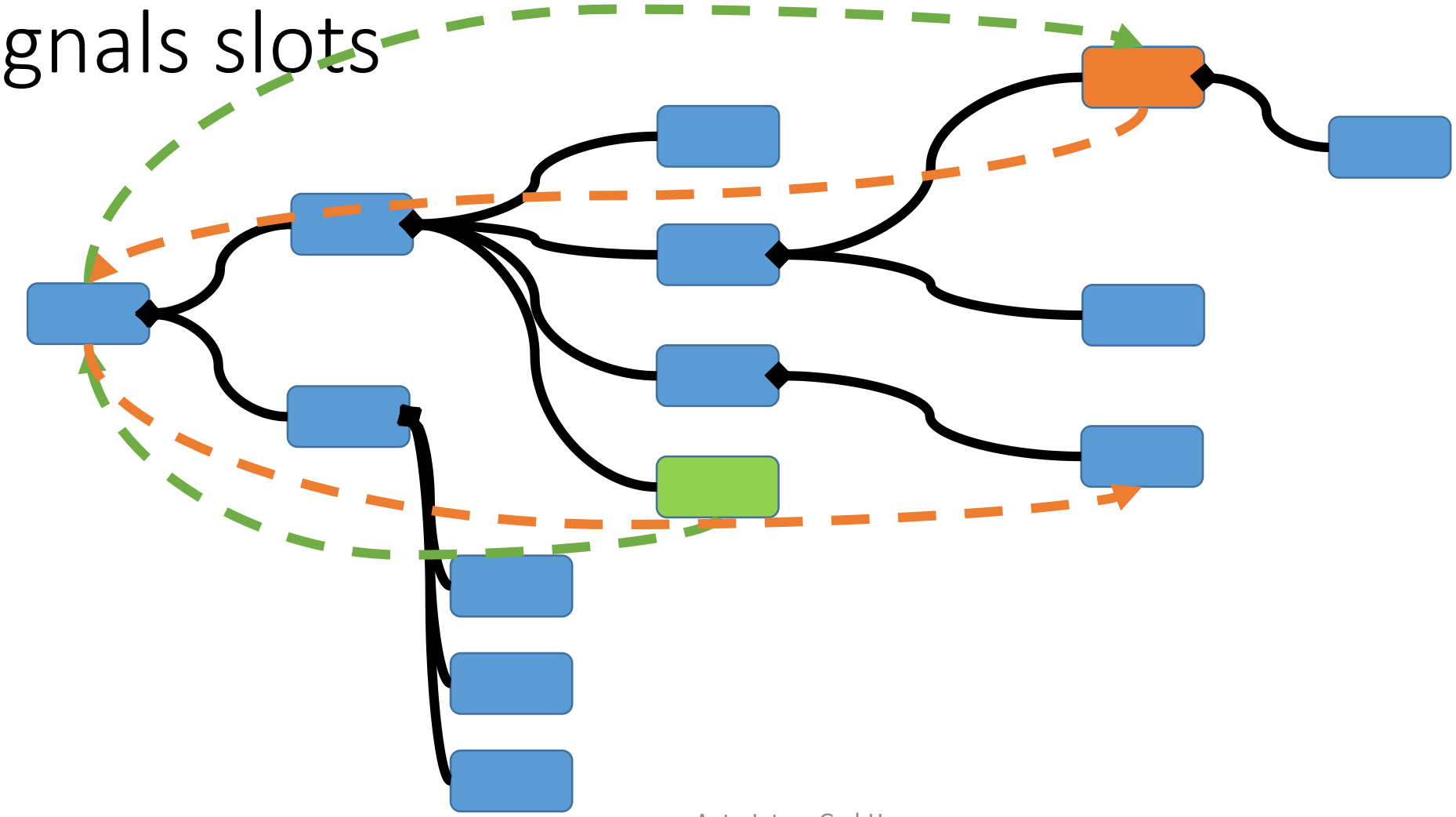
Signals slots



Signals slots



Signals slots



Step 7 of 17 step hello world tutorial

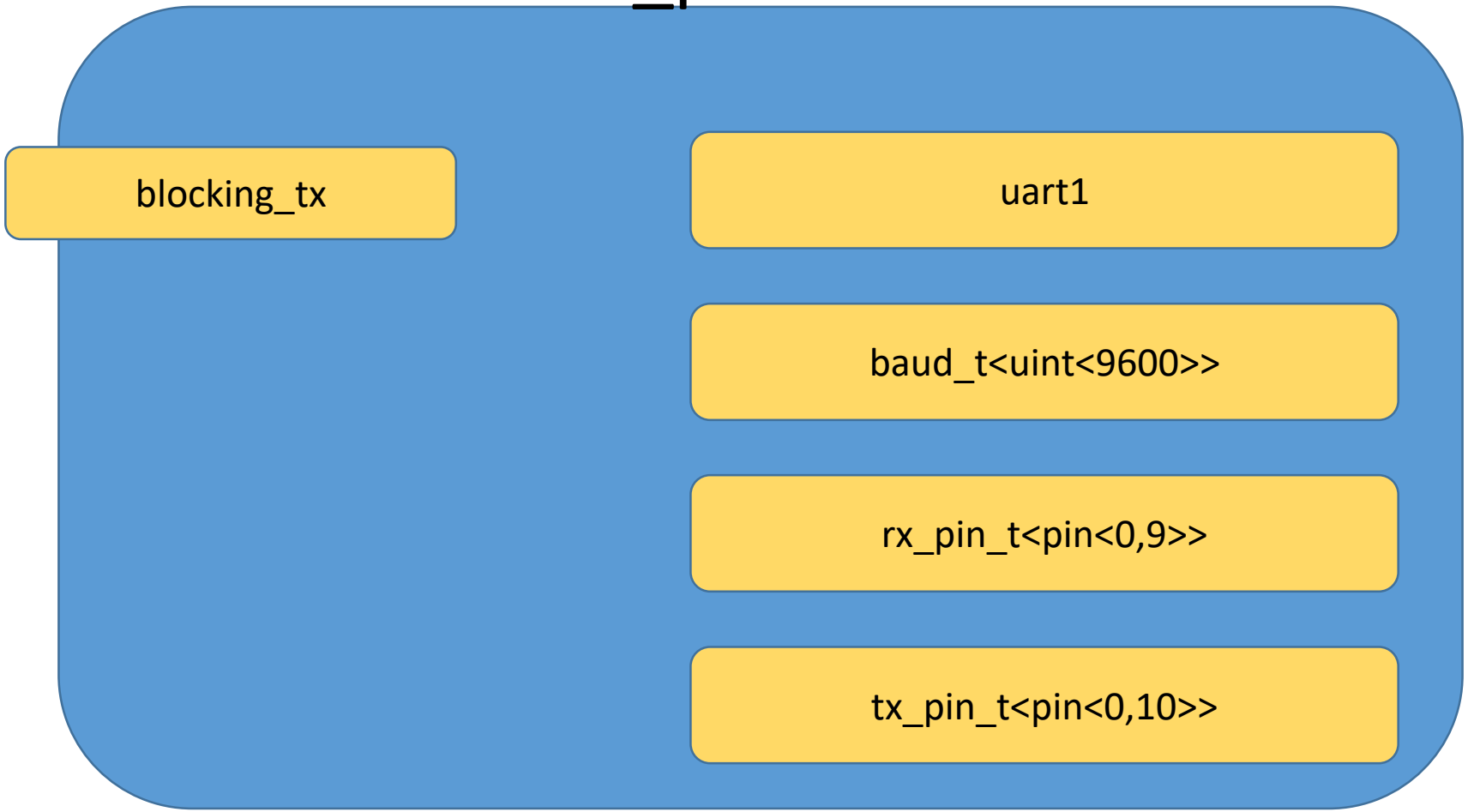
7. Now we will add the UART support. Add the following code before your main() function:

```
#include <stm32f10x_usart.h>

void InitializeUSART()
{
    USART_InitTypeDef usartConfig;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO, ENA
    USART_Cmd(USART1, ENABLE);

    usartConfig.USART_BaudRate = 9600;
    usartConfig.USART_WordLength = USART_WordLength_8b;
    usartConfig.USART_StopBits = USART_StopBits_1;
    usartConfig.USART_Parity = USART_Parity_No;
    usartConfig.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
    usartConfig.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_Init(USART1, &usartConfig);

    GPIO_InitTypeDef gpioConfig;

    //PA9 = USART1.TX => Alternative Function Output
    gpioConfig.GPIO_Mode = GPIO_Mode_AF_PP;
    gpioConfig.GPIO_Pin = GPIO_Pin_9;
    gpioConfig.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_Init(GPIOA, &gpioConfig);

    //PA10 = USART1.RX => Input
    gpioConfig.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    gpioConfig.GPIO_Pin = GPIO_Pin_10;
    GPIO_Init(GPIOA, &gpioConfig);
}

unsignedchar USART_ReadByteSync(USART_TypeDef *USARTx)
{
    while ((USARTx->SR & USART_SR_RXNE) == 0)
    {
    }

    return (unsigned char)USART_ReceiveData(USARTx);
}
```

drivers

```
auto myUart = make_uart(
        interface<blocking_tx>,
        uart1,
        9600_baud,
        rx = 0.9_pin,
        tx = 0.10_pin
);

myuart.blocking_send("hello world");
```

Serial_port


blocking_tx
uart1
baud_t<uint<9600>>
rx_pin_t<pin<0,9>>
tx_pin_t<pin<0,10>>

Liberasure style composable type erasure

```
auto thing = erase(interface<foo,bar>);

thing = compose(interface<foo,bar,baz>,
                some_guts{},
                other_guts{});

thing = compose(interface<foo,bar,ding,dong>,
                james_bond{},
                martini{},
                blond{},
                redhead{});
```

@odinthenerd

- Github.com
- Twitter.com
- Gmail.com
- Blogspot.com
- LinkedIn.com
- Embo.io