#NANTES TECH

Capnovae

Embedded Development As A Service

Modern C++ in Embedded Software from real life

CUST-REQ-0042 : Acquisition rate : 50 Hz

```
auto handle = xTimerCreateStatic(
    "acquisition", // timer name
    20, // period
    true, // is_auto_reload
    pv_parameter, // callback parameter
    acquire, // callback
    &timer_context // FreeRTOS callback context
);
```

Is implementation correct ?

20 ? 20 s ? 200 ms ? 20 ticks ?

Use units !!

```
auto handle = xTimerCreateStatic(
    "acquisition", // timer name
    20_ticks, // period
    true, // is_auto_reload
    pv_parameter, // callback
    parameter
    callback, // callback
    &timer_context // FreeRTOS
    callback context
);
```

User-defined literals enable adding a suffix to it (C++11) :

```
auto period = 100_ticks ;
```

integer literals (decimal)

```
auto color = 0x643216_RGB ;
```

also with hexa, octal, binary

```
auto weigth = 2.17_kg ;
```

floating literals

```
auto sol = ‘G’ _ennote;
```

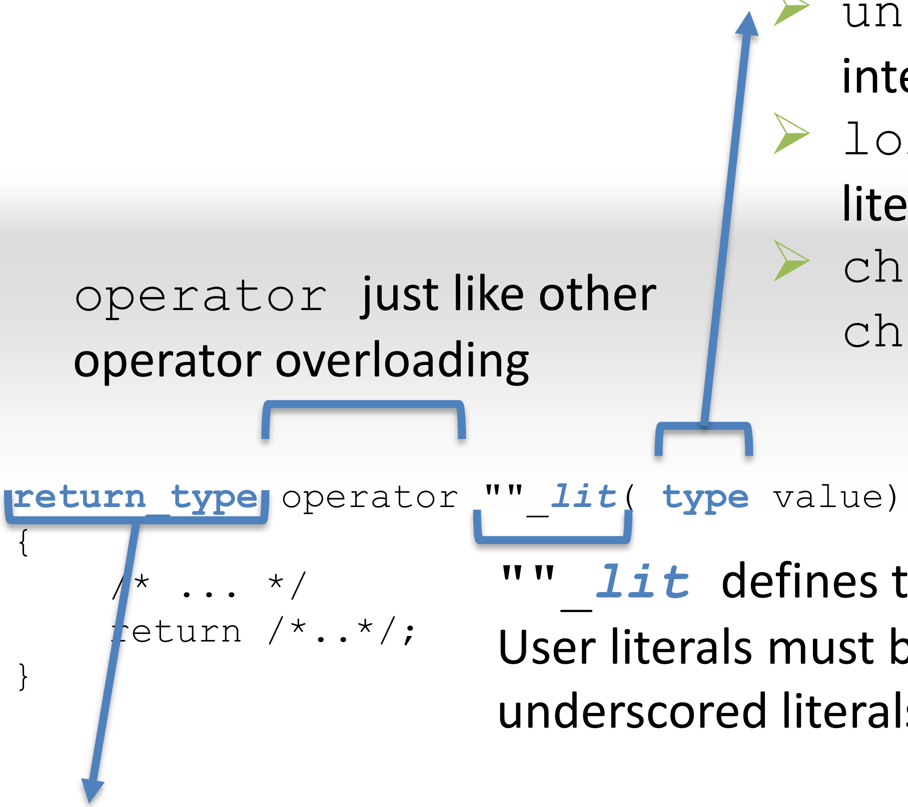
character literals

```
auto full_date = “2016-10-03T18:22:39+1”_UTC ;
```

string literals

… making literals clearly readable

User literals are defined through the definition of a new operator :



type can be :

- `unsigned long long` for integral literals ;
- `long double` for floating point literals ;
- `char, wchar_t, char16_t, char32_t` for character literals.

`operator` just like other operator overloading

```
return_type operator ""_lit( type value)
{
      /* ... */
      return /*..*/;
}
```

`""_lit` defines the literal. String must be empty. User literals must begin with underscore. None underscored literals are reserved for future uses

`return_type` is what you want

Capnovae

Embedded Development As A Service

Operator for user defined string literals :

```
return_type operator ""_lit( type value, std::size_t str_size)
{
    /* ... */
    return /*..*/;
}
```

type for strings can be :

- `const char *`
- `const wchar_t *`
- `const char16_t *`
- `const char32_t *`

Examples

```
auto color = 0x643216_qtRGB ;
```

```
QRgb operator ""_qtRGB(unsigned long
long rgb)
{
    return QRgb(rgb);
}
```

```
auto weigth = 2.17_kg ;
```

```
double operator ""_kg(long double w)
{
    return w;
}
```

```
auto sol = ‘G’ _ennote;


        enum class notes {
            ut,
            re,
            mi,
            fa,
            sol,
            la,
            si
        };
```

```
notes operator "" _ennote(char c)
{
    switch(c)
    {
        case 'A': return notes::la;
        case 'B': return notes::si;
        case 'C': return notes::ut;
        case 'D': return notes::re;
        case 'E': return notes::mi;
        case 'F': return notes::fa;
        case 'G': return notes::sol;
        default: throw;
        break;
    }
}
```

Examples

```
auto full_date = “2016-10-03T18:22:39+1”_UTC ;
```

```
#include <ctime>
#include <sstream>
#include <iomanip>
tm operator ""_UTC(const char* str_, std::size_t str_len_)
{
    std::istringstream iss(str_);
    tm ret={};
    iss>>std::get_time(&ret,"%Y-%m-%dT%H:%M:%S");
    return ret;
}
```

Literals defined by the norm (>=C++14) :

- `h` : hours
- `min` : minutes
- `s` : seconds
- `ms` : milliseconds
- `us` : microseconds
- `ns` : nanoseconds

```
#include <chrono>

// one of :
//using namespace std;
//using namespace std::chrono;
//using namespace std::literals;
//using namespace std::chrono_literals;
//using std::literals::chrono_literals::operator "" h;
using namespace std::literals::chrono_literals;

int main()
{
auto one_day = 24h;
auto half_hour = 30min;
auto usain_bolt = 9.9666s;
// …
    return 0;
}
```

Literals defined by the norm (>=C++14) :

- `i` : pure imaginary number (returns `complex<double>`)
- `if` : pure imaginary number (returns `complex<float>`)
- `il` : pure imaginary number (returns `complex<long double>`)

```
#include <complex>

// one of :
//using namespace std;
//using namespace std::literals;
//using namespace std::complex_literals;
//using std::literals::complex_literals::operator ""i;
using namespace std::literals::complex_literals;

int main()
{
auto id = 1i; // 0+1i
auto ifl = 1if;
auto ild = 1il;

// …
    return 0;
}
```

Literals defined by the norm (>=C++14) :

- s : converts string literals into std::basic_string

```
#include <string>

// one of :
//using namespace std;
//using namespace std::literals;
//using namespace std::string_literals;
//using std::literals::string_literals::operator ""s;
using namespace std::literals::string_literals;

int main()
{
auto hello = "hello"s; // std::string
auto world = u8"world"s; // std::string

auto how = u"how"s; // std::u16string
auto are = U"are"s; // std::u32string
auto you = L"you"s; //  std::wstring

// …
    return 0;
}
```

More on numeric literals (integer and double)

```
auto v_i = 12_lit;
auto v_d = 12e-7_lit;

/* … */ operator ""_lit(unsigned long long ull);
/* … */ operator ""_lit(long double ld);
     auto v_i = operator ""_lit(12ULL);
     auto v_d = operator ""_lit(12e-7L);
```

if not found :

```
/* … */ operator ""_lit(const char* c);
     auto v_i = operator ""_lit("12");
     auto v_d = operator ""_lit("12e-7");
```

mutually exclusive
(only one defined)

(x)or :

```
template<char... T> /* … */ operator ""_lit();
     auto v_i = operator ""_lit<'1','2'>();
     auto v_d = operator ""_lit<'1','2','e','-','7'>();
```

CUST-REQ-0042 : Acquisition rate : 50 Hz

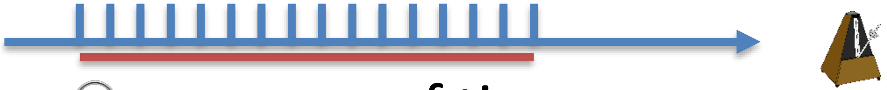
```
auto handle = xTimerCreateStatic(
    "acquisition", // timer name
    20_ticks, // period
    true, // is_auto_reload
    pv_parameter, // callback
    parameter
    callback, // callback
    &timer_context // FreeRTOS
    callback context
);
```

Is implementation correct ?

Use <chrono> for handling times quantities !!!

<chrono> (>=C++11)

duration

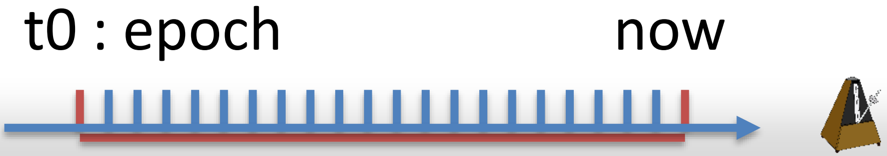


= span of time

= number of ticks and a tick period

= difference between two time_point

clock

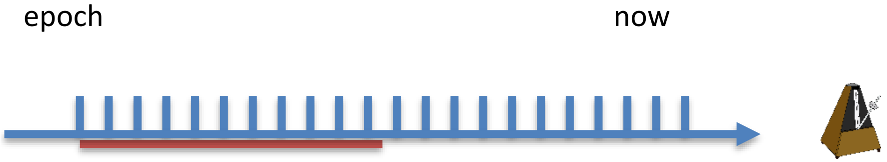


= time point + duration

= starting point + resolution (tick rate)

+ now

time_point

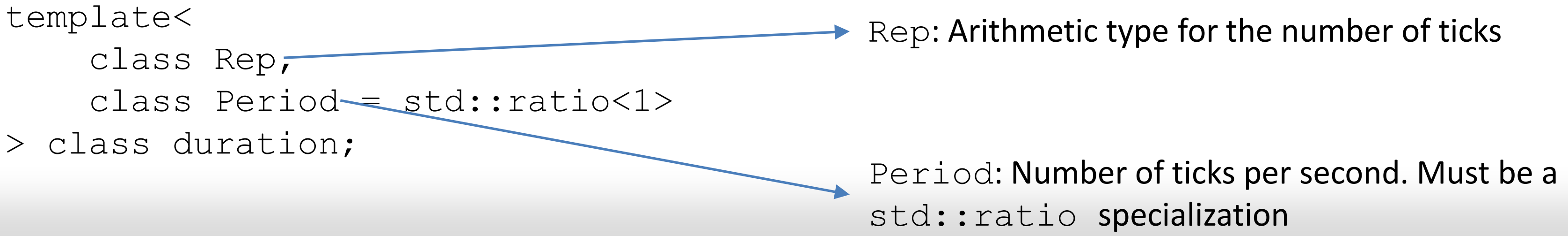


= point in time

= clock + duration

= date

duration



```
template<
  class Rep,
  class Period = std::ratio<1>
> class duration;
```

`Rep`: Arithmetic type for the number of ticks

`Period`: Number of ticks per second. Must be a `std::ratio` specialization

A `duration` object contains a number of ticks : `count()`

```
std::chrono::duration<int> v1(5);
std::cout<<v1.count()<<" ticks are "<<v1.count()<<" seconds"<<"\n";
std::chrono::duration<int, std::ratio<20,1>> v2(5);
std::cout<<v2.count()<<" ticks are "<<v2.count()*20<<" seconds\n";
```

<ratio> Representing compile-time rational constant(>=C++11)

```
template<intmax_t N, intmax_t D = 1>
class ratio
{
public:
   static_assert(D != 0, "denominator cannot be zero");
   static constexpr intmax_t num = /* sign(N) * sign(D) * abs(N) / gcd */;
   static constexpr intmax_t den = /* abs(D)/ gcd */;
   typedef ratio<num, den> type;
};
```

```
template <class R1, class R2> using ratio_add = /* ratio<U,V> R1+R2 */;
ratio_subtract, ratio_multiply, ratio_divide

template <class R1, class R2> struct ratio_equal /* :
integral_constant<bool, R1==R2> {}*/;
ratio_not_equal, ratio_less, ratio_less_equal, ratio_greater,
ratio_greater_equal
```

std::ratio<1,	10^{-1}	10^{-2}	10^{-3}	10^{-6}	10^{-9}	10^{-12}	10^{-15}	10^{-18}	10^{-21}	10^{-24}
	deci	centi	milli	micro	nano	pico	femto	atto	zepto	yocto

std::ratio<,1	10^{1}	10^{2}	10^{3}	10^{6}	10^{9}	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}
	deca	hecto	kilo	mega	giga	tera	peta	exa	zetta	yotta

Defined time units

`std::chrono::nanoseconds`	`duration</* impl */, std::nano>`
`std::chrono::microseconds`	`duration</* impl */, std::micro>`
`std::chrono::milliseconds`	`duration</* impl */, std::milli>`
`std::chrono::seconds`	`duration</* impl */>`
`std::chrono::minutes`	`duration</* impl */, std::ratio<60>>`
`std::chrono::hours`	`duration</*impl */, std::ratio<3600>>`

`impl`: signed integer type wide enough to represent around 292 years

Literals (>=C++14)

`std::chrono::nanoseconds`	`ns`
`std::chrono::microseconds`	`us`
`std::chrono::milliseconds`	`ms`
`std::chrono::seconds`	`s`
`std::chrono::minutes`	`min`
`std::chrono::hours`	`h`

Arithmetic :

```
#include <chrono>
#include <iostream>

using namespace std::literals::chrono_literals;

int main()
{
    std::chrono::milliseconds t1 = 10min;
    auto t2 = -t1;
    std::cout<<std::boolalpha<<(t1==-t2)<<"\n";

    --t1;++t2;
    std::cout<<(t1==-t2)<<"\n";

    t1++;t2--;
    std::cout<<(t1==-t2)<<"\n";

    t1 += 60s;
    std::cout<<(t1==11min)<<"\n";
    t2 = t1 - 60s;
    std::cout<<(t2==10min)<<"\n";

    t1 = t2 * 2;
    std::cout<<(t1==20min)<<"\n";

    auto t3 = 1h + 12min + 24s;

    std::cout<<((t3%1min)==24s)<<"\n";
    std::cout<<((t3%1h)==(12min +  24s))<<"\n";
    return 0;
}
```

Implicit conversions :

For integral representations, durations shall be proportional

```
std::chrono::minutes t1 = 10min;
std::chrono::seconds t2;
t2 = t1; // OK
t1 = t2; // Error
```

Always for floating representations

```
std::chrono::duration<double,std::ratio<60,1>> t1(10.76);
std::chrono::duration<float,std::kilo> t2;
t2 = t1; // OK
t1 = t2; // OK
```

Explicit conversions with `duration_cast`

Makes the conversion with potential loss of precision if destination tick rate is less frequent than source tick rate

```
std::chrono::microseconds v(3923005690);

std::cout<<v.count()<<" microseconds are :"
<<" "<<std::chrono::duration_cast<std::chrono::hours>(v).count()<<" hours"
<<" "<<std::chrono::duration_cast<std::chrono::minutes>(v%1h).count()<<" minutes"
<<" "<<std::chrono::duration_cast<std::chrono::seconds>(v%1min).count()<<" seconds"
<<" and "<<std::chrono::duration_cast<std::chrono::microseconds>(v%1s).count()<<" microseconds"
<<"\n"
;
```

CUST-REQ-0042 : Acquisition rate : 50 Hz

```
auto handle = xTimerCreateStatic(
     "acquisition", // timer name
     20_ticks, // period
     true, // is_auto_reload
     pv_parameter, // callback parameter
     callback, // callback
     &timer_context // FreeRTOS callback context
);
```

Is implementation correct ?

```
auto handle = xTimerCreateStatic(
     "acquisition",
     to_os_ticks(to_period(50_Hz)),
     true,
     pv_parameter,
     callback,
     &timer_context
);
```

```
extern "C" TimerHandle_t xTimerCreateStatic(
    const char * const pcTimerName,
    const TickType_t xTimerPeriod,
    const UBaseType_t uxAutoReload,
    void * const pvTimerID,
    TimerCallbackFunction_t pxCallbackFunction,
    StaticTimer_t *pxTimerBuffer
);
```

```
using frequence_t = uint32_t;
constexpr auto operator""_Hz(unsigned long long f)
{
    return frequence_t{f};
}
constexpr auto to_period(frequence_t f) {return std::chrono::milliseconds(1000/f); }
```

```
using tick_duration_t = std::chrono::duration<TickType_t,std::ratio<1,configTICK_RATE_HZ>>;
template<class R, class P>
constexpr TickType_t to_os_ticks(std::chrono::duration<R,P> period_)
{
    return  std::chrono::duration_cast<tick_duration_t>(period_).count();
}
```

Capnovae

Embedded Development As A Service

clock

A **concept** to bundle duration and time points

(there is also the concept of TrivialClock which enforces properties on rep, duration, time_point and now no except

```
template<typename C> concept bool Clock() {
    return requires (C ) {
        typename C::rep;
        Same<typename C::period,std::ratio>;
        Same<typename C::duration,
            std::chrono::duration<typename C::rep,typename C::period>>;
        Same<typename C::time_point,std::chrono::time_point<C>>;
```

<- (a little more subtle in the norm)

```
        {C::is_steady} -> const bool;
        {C::now()} -> typename C::time_point;
    };
}
```

steady is true if now() is ~increasing function
(t1<=t2 for t1 = now() called “before” t2 = now())
and time between two ticks is always the same

`std::chrono::system_clock` wall clock time from the system-wide realtime clock
=> may or may not be steady

`std::chrono::steady_clock` steady clock relative to real time
=> may or may not be system-wide realtime

`std::chrono::high_resolution_clock` clocks with the shortest tick period.
=> the more precise clock you may expect on your system

Playing with concepts

http://coliru.stacked-crooked.com/

g++ -std=c++14 -O2 -Wall -pedantic -pthread main.cpp -fconcepts && ./a.out

```
#include <type_traits>
template<typename T1, typename T2> concept bool Same()
{
  return std::is_same<T1,T2>::value;
};

#include <ratio>
#include <chrono>

template<typename C> concept bool Clock() {
return requires (C ) {
typename C::rep;
Same<typename C::period,std::ratio>;
Same<typename C::duration,
std::chrono::duration<typename C::rep,typename C::period>>;
Same<typename C::time_point,std::chrono::time_point<C>>;

{C::is_steady} -> const bool;
{C::now()} -> std::chrono::time_point<C>
};
}
```

time_point

`time_point<clock,duration>`

The value :

```
constexpr duration time_since_epoch() const;
```

The arithmetic of time points :

```
time_point += duration -> time_point
time_point -= duration -> time_point
time_point + duration -> time_point
duration + time_point -> time_point
time_point - duration -> time_point
~~duration - time_point~~
time_point - time_point  -> duration
```

Comparison :

`<,<=,>,>=,==,!=` between 2 time points

Casting between time durations :

```
time_point<system_clock,seconds> t1sec;
time_point<system_clock,hours> t1hour;

time_point<system_clock,seconds> t2sec = t1hour;
time_point<system_clock,hours> t2hour = time_point_cast<hours>(t1sec);
```

CUST-REQ-0042 : Acquisition rate : 50 Hz

Immediately checked ☺

```
auto handle = xTimerCreateStatic(
     "acquisition", // timer name
     to_os_ticks(to_period(50_Hz)), // period
     true, // is_auto_reload
     pv_parameter, // callback parameter
     callback, // callback
     &timer_context // FreeRTOS callback context
);
```

Coding Rule CR-0042 : No magic number !!!

Use constant expressions

How many ways to define a constant in C++?

```
#define CONSTANT_1 1
const int CONSTANT_2 = 2;
static const int CONSTANT_3 = 3;
namespace /* anonymous*/ {
    const int CONSTANT_4 = 4;
}
constexpr int CONSTANT_5 = 5;
static constexpr int CONSTANT_6 = 6;
constexpr int CONSTANT_7() {return 7;}

struct as_int
{
int val;
    constexpr as_int(int v):val{v}{}
    constexpr operator int() const {return val;}
};

static constexpr as_int CONSTANT_8(8);

enum{
    CONSTANT_9 = 9
};
enum class E : int {
    CONSTANT_10 = 10
};

using CONSTANT_11 = std::integral_constant<int,11>;
```

`#define CONSTANT_1 1`	not type safe, quite dangerous
`const int CONSTANT_2 = 2;`	name crosses translation-unit
`static const int CONSTANT_3 = 3;`	type safe, name scoped to translation unit

`namespace /* anonymous*/`
`{ const int CONSTANT_4 = 4; }`
	merely same as previous (differences in linkage)

`constexpr int CONSTANT_5 = 5;`	Explicitly indicates that the expression is known at compile-time
`static constexpr int CONSTANT_6 = 6;`	static and constexpr are orthogonal just like static and const
`constexpr int CONSTANT_7() {return 7;}`	Function can be constant expression. Must respect constraints merely of simplicity

`struct as_int {`
`  int val;`
`  constexpr as_int(int v):val{v}{}`
`  constexpr operator int() const {return val;}`
`};`
`static constexpr as_int CONSTANT_8(8);`
	Can define class with constexpr constructor and functions and then define constexpr objects

`enum {CONSTANT_9 = 9};`	C legacy. Badly typed. Limited to integer values
`enum class E : int {CONSTANT_10 = 10};`	Same as previous even if type is more explicit

`using CONSTANT_11 =`
`std::integral_constant<int,11>;`
	Unless meta –prog, this seems a little “over”

Difference between `const` and `constexpr`

```
const int CONSTANT_2 = 2;
```

Code can't change the value, but the variable can have its value changed :

```
volatile const int CONSTANT_2 = 2;
```

```
constexpr int CONSTANT_5 = 5;
```

Code can't change the value AND the value is known at compile-time.

~~volatile constexpr int CONSTANT_5 = 5;~~

λ

```
#include <iostream>
#include <cstdint>
#include <algorithm>
#include <iterator>
#include <string>
#include <sstream>

using age_t = uint16_t;
struct entry
{
    std::string name;
    age_t age;
};

std::vector<entry> v = {
        {"dupont",32}
        ,{"durand",44}
        ,{"martin",23}
        ,{"legrand",67}
        ,{"perrin",74}
        ,{"bernard",54}
    };
```

λ

```
int main()
{
    std::sort(begin(v),end(v),[](auto l, auto r){
        return l.name<r.name;
    });
    std::transform(begin(v),end(v),std::ostream_iterator<std::string>(std::cout,"\n"),
        [](auto e){
            std::ostringstream oss;
            oss<<e.name<<" ( "<<e.age<<" )";
            return oss.str();
        }
    );

    std::cout<<"younger is "<<std::min_element(begin(v),end(v),
        [](auto l,auto r){return l.age<r.age;})->name<<"\n";

    age_t max_age = 65;
    std::cout<<"Retired : \n";
    std::vector<entry> retired;
    std::copy_if(begin(v),end(v),std::back_inserter(retired),
        [max_age](auto e){return e.age>=max_age;});
    std::transform(begin(retired),end(retired),std::ostream_iterator<std::string>(std::cout,"\n"),
        [](auto e){return e.name;});

    return 0;
}
```

λ

Lambda must begin with this introducer

parameters

body

```
auto i_plus = [](int i,int j){return i+j;};
std::cout<<i_plus(1,2)<<"\n";
```

```
auto i_plus = [](int i,int j) -> int {return i +j;};
```

return type when not trivial

```
auto i_plus = [](int i,int j) -> decltype(i+j) {return i +j;};
```

deduced

λ

```
[](int i,int j){return i+j;};

 struct /* */{
    auto operator()(int i, int j) const
    {
        return i+j;
    }
 };
```

```
[](auto i,auto j){return i+j;};

 struct /* */{
    template<class T1, class T2>
    auto operator()(T1 i, T2 j) const
    {
        return i+j;
    }
 };
```

λ

```
int step = 5;
auto shift = [step](int i){return i+step;};
std::cout<<shift(1)<<"\n";
step = 10;
std::cout<<shift(1)<<"\n";
```

Capture list : by value

```
6
6
```

```
int step = 5;
auto shift = [&step](int i){return i+step;};
std::cout<<shift(1)<<"\n";
step = 10;
std::cout<<shift(1)<<"\n";
```

Capture list : by reference

```
6
11
```

λ

[](*parameters*){*body*}	Capture nothing
[a,&b](*parameters*){*body*}	by value or by reference
[this](*parameters*){*body*}	capturing `this` pointer
[&](*parameters*){*body*}	all by reference
[=](*parameters*){*body*}	all by value
[=,&i](*parameters*){*body*}	all by value except i by reference
[&,i](*parameters*){*body*}	all by value except i by value

λ

```
struct command
{
    auto delay()
    {
        return [this]{do_it();};
    }

private:
    void do_it() const
    {
        std::cout<<"command::do_it"<<"\n";
    }
};

int main()
{
    command c;
    auto d = c.delay();
    d();
    return 0;
}
```

λ

```
[capture-list](parameters){body};

            struct /* */{
                auto operator()(parameters) const
                {
                    body
                }
            };

[capture-list](parameters) mutable {body};

            struct /* */{
                auto operator()(parameters)
                {
                    body
                }
            };

[capture-list](parameters) constexpr {body};
```

C++17

```
            struct /* */{
                constexpr auto operator()(parameters)
                {
                    body
                }
            };
```

Converting to function pointer for non capturing lambda

```
auto lambda = [](parameters){body};
using ptr_fn_t = ret (*)(parameters);
ptr_fn_t pfn = lambda;
pfn(1,2);
```

Converting to `std::function` (can capture lambda)

```
int step = 5;
auto shift = [&step](int i){return i+step;};
std::function<int (int)> fn2 = shift;
std::cout<<fn2(37)<<"\n";
```

```
class ITask
{
public:
    void run()
    {
        do_run();
    }
private:
    virtual void do_run()=0;
};

class Task : public ITask
{
public:
    virtual ~Task()=default;
private:
    virtual void do_run() final override
    {
        do_init();
        initEvtGuard.waitall();
        while(1){
            do_loop();
        }
    }

    virtual void do_init() = 0;
    virtual void do_loop()=0;

private:
    SyncEventGuard initEvtGuard;
};
```

```
struct watchdog_task : Task
{
private:
    void do_init()
    {/*..*/}
    void do_loop()
    {/*..*/}
};
```

```
using ITask = std::function<void()>;

int main()
{
    SyncEventGuard initEvtGuard;
    auto make_task = [&initEvtGuard](auto do_init, auto do_loop){
        return std::function<void (void)>([&]{
            do_init();
            initEvtGuard.waitall();
            while(1){
                do_loop();
            }
        });
    };


    ITask wd = make_task([]{/* ... */},[]{/* ... */ });
    wd();
    return 0;
}
```

```
std::vector<Itask> tasks;
```

Merci