

MIDI TECHNO!!!!



Équipe Indu



Intro

```
#include <chrono>  
using namespace std::chrono
```

Introduit dans C++11,
mis à jour dans C++14, C++17, C++20

Adapté de “A <chrono> Tutorial” par
Howard Hinnant

<chrono>

- Pourquoi <chrono>
- Durées
- Point dans le temps
- Exemples

Pourquoi <chrono>?

```
sleep(10);
```

10 secondes?

10 millisecondes?

10 nanosecondes?

```
sleep(10ms);
```

Pourquoi <chrono>?

- Représenter une durée avec un type intégral est ambiguë
- <chrono> aide le compilateur à vous aider à trouver les erreurs de logique.
- Faire de différents concepts différents types.

<chrono>

- Pourquoi <chrono>
- Durée
- Point dans le temps
- Exemples

Durée

- Une période de temps
 - 3 secondes
 - 3 minutes
 - 3 heures

std::chrono::seconds

- Type arithmétique
- `sizeof(seconds) == 8`
- Simple et rapide!

```
class seconds
{
    int64_t sec_;
public:
    seconds() = default;
    // ...
};
```

std::chrono::seconds

- Se construit comme un scalaire:

```
seconds s; // no init
```

```
seconds s{}; // zero init
```

std::chrono::seconds

```
seconds s = 3;  
// error: conversion from 'int' to non-scalar  
type 'std::chrono::seconds'
```

```
seconds s{3};  
// OK
```

std::chrono::seconds

- Pas de conversion implicite de int vers std::chrono::seconds

```
void f(seconds d)
{
    cout << d.count() << "s\n";
}

f(3);
// error: Not implicitly constructible
from int
```

std::chrono::seconds

```
void f(seconds d)
{
    cout << d.count() << "s\n";
}
```

```
f(seconds{3});    // ok, 3s
f(3s);            // ok, 3s
seconds x{3};
f(x);             // ok, 3s
```

std::chrono::seconds

- Opération arithmétique +,-,*,/

```
void f(seconds d) { ... }
```

```
auto x = 3s;
```

```
x += 2s;
```

```
f(x);           // ok, 5s
```

```
x = x - 1s;
```

```
f(x);           // ok, 4s
```

```
f(x + 1);       // error: seconds + int
```

std::chrono::seconds

- Comparaison, ==, !=, >, >=, <, <=

```
void f(seconds d)
{
    if (d <= 2s) // OK
        // ...
    if (d <= 2) // Error: seconds <= int
        // ...
}
```

std::chrono::seconds

- Performance?

```
seconds  
f(seconds x, seconds y)  
{  
    return x + y;  
}
```

```
int64_t  
g(int64_t x, int64_t y)  
{  
    return x + y;  
}
```


std::chrono::seconds

- Performance?

```
cperry@pc-cerry:~/sfl/miditechno/chrono$ g++ -c -O1 -o func.o func.c
cperry@pc-cerry:~/sfl/miditechno/chrono$ objdump -D func.o

func.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <_Z1fNSt6chrono8durationIlSt5ratioILL1ELl1EEEEES3_>:
   0:  48 8d 04 37          lea    (%rdi,%rsi,1),%rax
   4:  c3                  retq

0000000000000005 <_Z1gmm>:
   5:  48 8d 04 37          lea    (%rdi,%rsi,1),%rax
   9:  c3                  retq
```

std::chrono::seconds

- Limites?

```
seconds m = seconds::min();
```

```
seconds M = seconds::max();
```

- ± 292 milliards d'années

std::chrono::seconds

Type	Definition
std::chrono::nanoseconds	duration</*signed integer type of at least 64 bits*/, std::nano>
std::chrono::microseconds	duration</*signed integer type of at least 55 bits*/, std::micro>
std::chrono::milliseconds	duration</*signed integer type of at least 45 bits*/, std::milli>
std::chrono::seconds	duration</*signed integer type of at least 35 bits*/>
std::chrono::minutes	duration</*signed integer type of at least 29 bits*/, std::ratio<60>>
std::chrono::hours	duration</*signed integer type of at least 23 bits*/, std::ratio<3600>>
std::chrono::days (since C++20)	duration</*signed integer type of at least 25 bits*/, std::ratio<86400>>
std::chrono::weeks (since C++20)	duration</*signed integer type of at least 22 bits*/, std::ratio<604800>>
std::chrono::months (since C++20)	duration</*signed integer type of at least 20 bits*/, std::ratio<2629746>>
std::chrono::years (since C++20)	duration</*signed integer type of at least 17 bits*/, std::ratio<31556952>>

Note: each of the predefined duration types up to hours covers a range of at least ± 292 years.

std::chrono::seconds

- Une simple surcouche sur un int64_t
- À quoi bon?!

std::chrono::seconds

- Un gros projet C++ de plusieurs millions de lignes.
- Migrer vers des millisecondes pour plus de précision.



std::chrono::milliseconds

```
class milliseconds
{
    int64_t ms_;
public:
    milliseconds() = default;
    // ...
};
```

std::chrono::milliseconds

- Changer une fonction à la fois.

```
void f(milliseconds d)
{
    cout << d.count() << "ms\n";
}

f(3)                // Error
f(seconds{3});      // OK, 3000ms
f(3s);              // OK, 3000ms
seconds x{3};
f(x);               // OK, 3000ms
```

std::chrono::milliseconds

- <chrono> sait comment convertir de secondes vers milliseconds.
- La conversion est simple mais il est facile de se tromper, surtout dans le cadre d'un gros projet.

std::chrono::milliseconds

- Performance?

```
milliseconds  
f(seconds x)  
{  
    return x;  
}
```

```
int64_t  
g(int64_t x)  
{  
    return x*1000;  
}
```

std::chrono::milliseconds

- Performance?

```
cperry@pc-cerry:~/sfl/miditechno/chrono$ g++ -c -O1 -o conv.o conv.c
cperry@pc-cerry:~/sfl/miditechno/chrono$ objdump -D conv.o

conv.o:          file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <_Z1fNSt6chrono8durationIlSt5ratioILL1ELl1EEEE>:
   0:  48 69 c7 e8 03 00 00    imul    $0x3e8,%rdi,%rax
   7:  c3                    retq

0000000000000008 <_Z1gl>:
   8:  48 69 c7 e8 03 00 00    imul    $0x3e8,%rdi,%rax
  f:  c3                    retq
```

std::chrono::milliseconds

- Arithmétique mixte:

```
auto x = 2s;  
auto y = 3ms;  
f(x + y);    // 2003ms  
f(y - x);    // -1997ms
```

std::chrono::milliseconds

- Si ça compile, ça marche!

.... sauf si vous avez utilisé .count() partout ...

std::chrono::milliseconds

- Conversion milliseconds → seconds
- Si la conversion est sans perte de précision, elle est implicite.

```
seconds x = 3400ms; // error: no conversion
seconds x = duration_cast<seconds>(3400ms); // 3s
seconds x = ceil<seconds>(3400ms); // 4s
seconds x = floor<seconds>(3400ms); // 3s
seconds x = round<seconds>(3400ms); // 3s
```

std::chrono::milliseconds

- Utiliser `duration_cast`, `ceil`, `floor`, `round` seulement si nécessaire.
- Préférer la conversion implicite.

```
std::chrono::hours  
std::chrono::minutes  
std::chrono::seconds  
std::chrono::milliseconds  
std::chrono::microseconds  
std::chrono::nanoseconds
```

```
void f(nanoseconds d)
{
    cout << d.count() << "ns\n";
}

auto x = 2h;
auto y = 3us;
f(x + y);           // 72000000003000ns
```


Représentation généralisée

std::chrono::duration

Defined in header `<chrono>`

```
template<
    class Rep,
    class Period = std::ratio<1>           (since C++11)
> class duration;
```

```
using seconds = duration<int64_t, std::ratio<1>>
```

Représentation généralisée

```
using seconds32 = duration<int32_t>
```



```
using safe_seconds32 =  
duration<safe<int32_t>>
```

Représentation généralisée

```
using fseconds = duration<float>;
```

```
void f(fseconds d)
{
    cout << d.count() << "s\n";
}
```

```
f(45ms + 63us);    // 0.045063s
```

Représentation généralisée

```
using fmilli = duration<float, std::milli>;

void f(fmilli d)
{
    cout << d.count() << "ms\n";
}

f(45ms + 63us);    // 45.063ms
```

Représentation généralisée

```
using nano    = ratio<1, 1'000'000'000>;  
using micro   = ratio<1,      1'000'000>;  
using milli   = ratio<1,      1'000>;
```

```
template <intmax_t N, intmax_t D = 1>  
class ratio  
{  
    ...  
};
```

Représentation généralisée

```
using ntsc_frames = duration<int32_t, ratio<1, 30>>;  
using pal_frames = duration<int32_t, ratio<1, 25>>;
```

```
f(ntsc_frames{1});    // 33.3333ms  
f(45ms + ntsc_frames{5}); // 211.667ms
```



Représentation généralisée

- Simple, seulement complexe si vous en avez besoin.

```
std::seconds = duration<int64_t ratio<1,1>>
```

```
std::string = basic_string<char,  
char_traits<char>, allocator<char>>
```

<chrono>

- Pourquoi <chrono>
- Durées
- Point dans le temps
- Exemples

time_point

- 10000s → n'importe quel 10000s
- Par contre:

```
time_point<system_clock, seconds> tp{10'000s};
```

= 1970-01-01 02:46:40 UTC

time_point

- Un point spécifique dans le temps, par rapport à une horloge avec une précision définie par un durée.

```
template<class Clock, class Duration>
class time_point {
    Duration d_;
public:
    // ...
};
```

time_point

- duration et time_point ont la même représentation, toutefois ils représentent différents concepts.

time_point

- arithmétique

```
auto d = tp1 - tp2;    // OK
```

```
auto tp2 = tp1 + d;    // OK
```

```
auto error = tp1 + tp2; //error: no match  
                        for 'operator+'
```

time_point

- conversion

```
template <class D>
using sys_time = time_point<system_clock, D>;

sys_time<seconds> tp{5s};           // 5s
sys_time<milliseconds> tp2 = tp;   // 5000ms

tp = time_point_cast<seconds>(6200ms); // 6s
```

time_point

- En dernier lieu: time_since_epoch

```
auto t = std::chrono::system_clock::now();
```

```
cout << t.time_since_epoch().count() << "ns\n";
```

<chrono>

- Pourquoi <chrono>
- Durées
- Point dans le temps
- Exemples

Examples

Foo.h

```
std::chrono::time_point<std::chrono::steady_clock> m_lastCounterIncrement{std::chrono::seconds::zero()};
```

Foo.cpp

```
static constexpr int maxCounterIntervalMs{100};  
  
// Temporary solution to check if the OpenSafety connection is valid  
if (duration_cast<milliseconds>(steady_clock::now() - m_lastCounterIncrement).count() > maxCounterIntervalMs)  
{
```

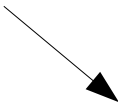

Examples

Foo.h

```
std::chrono::time_point<std::chrono::steady_clock> m_lastCounterIncrement{std::chrono::seconds::zero()};
```

Foo.cpp

```
static constexpr int maxCounterIntervalMs{100};  
  
// Temporary solution to check if the OpenSafety connection is valid  
if (duration_cast<milliseconds>(steady_clock::now() - m_lastCounterIncrement).count() > maxCounterIntervalMs)  
{
```



```
milliseconds maxCounter = 100ms;  
  
// Temporary solution to check if the OpenSafety connection is valid  
if (duration_cast<milliseconds>(steady_clock::now() - m_lastCounterIncrement) > maxCounter)  
{
```

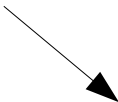
Examples

Foo.h

```
std::chrono::time_point<std::chrono::steady_clock> m_lastCounterIncrement{std::chrono::seconds::zero()};
```

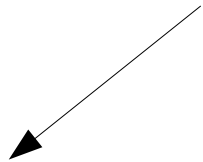
Foo.cpp

```
static constexpr int maxCounterIntervalMs{100};  
  
// Temporary solution to check if the OpenSafety connection is valid  
if (duration_cast<milliseconds>(steady_clock::now() - m_lastCounterIncrement).count() > maxCounterIntervalMs)  
{
```



```
    milliseconds maxCounter = 100ms;  
  
    // Temporary solution to check if the OpenSafety connection is valid  
    if (duration_cast<milliseconds>(steady_clock::now() - m_lastCounterIncrement) > maxCounter)  
    {
```

```
    milliseconds maxCounter = 100ms;  
  
    // Temporary solution to check if the OpenSafety connection is valid  
    if (steady_clock::now() - m_lastCounterIncrement > maxCounter)  
    {
```



Résumé

- Un concept = un type
- `<chrono>` se charge des conversions.
- Conversion implicite si sans perte de précision.
- `duration_cast`, `round`, `ceil`, `floor` si la conversion implique une perte de précision.
- `Count()`, `time_since_epoch()` seulement si nécessaire.

MIDI TECHNO !!!

- Besoin de présentateurs
- Sujet techno
- 30 à 60 min

