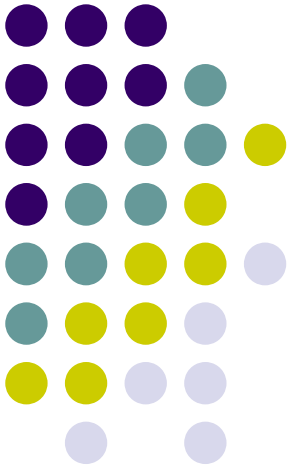
Średniozaawansowane programowanie w C++

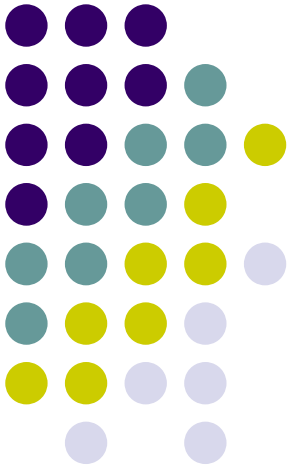
Wykład #4

4 listopada 2020 r.



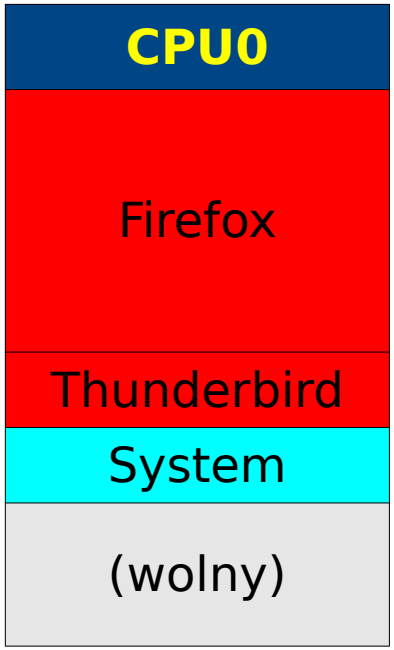
Systemy wieloprocesorowe

Czy więcej znaczy lepiej?

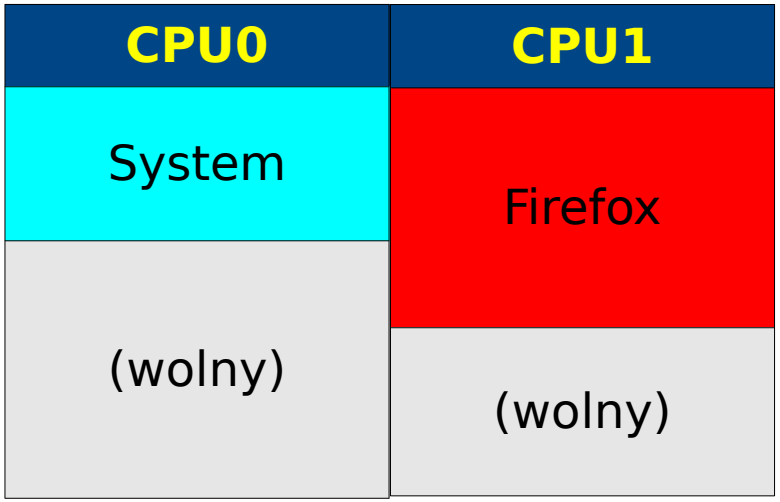


Zarządzanie procesorami przez OS

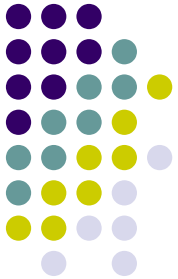
System jednoprocesorowy



System dwuprocesorowy

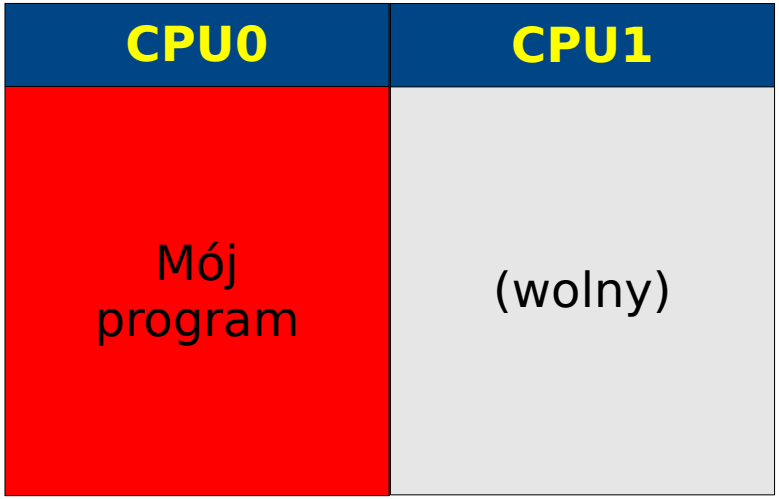


Aplikacje jednowątkowe



System jednoprocesorowy

System dwuprocesorowy

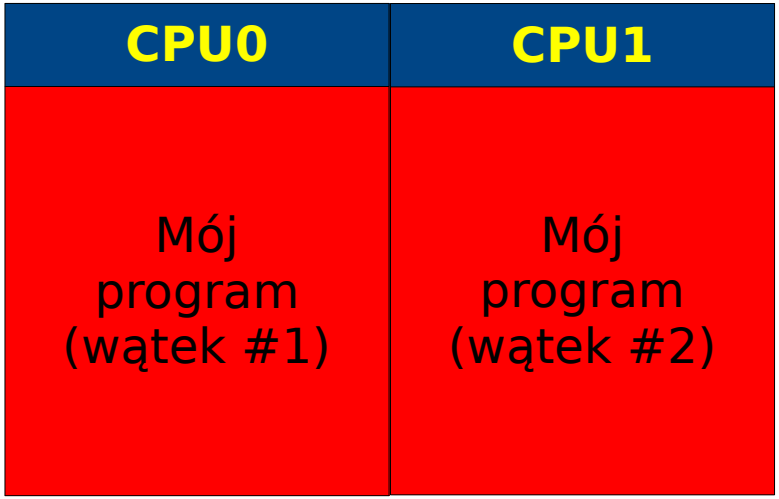


Aplikacje niezaprojektowane jako wielowątkowe będą działać szybciej na komputerach z jednym szybkim rdzeniem, niż z dwoma wolniejszymi!

Aplikacje wielowątkowe

System jednoprocesorowy

System dwuprocesorowy

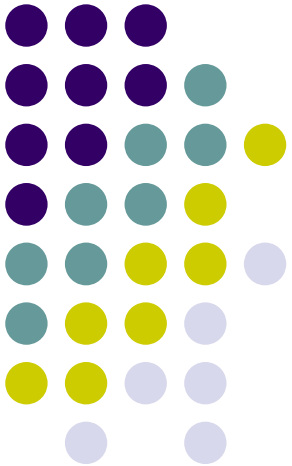


Dopiero jawne wydzielenie drugiego wątku prowadzi do skorzystania z zalet procesorów wielordzeniowych!

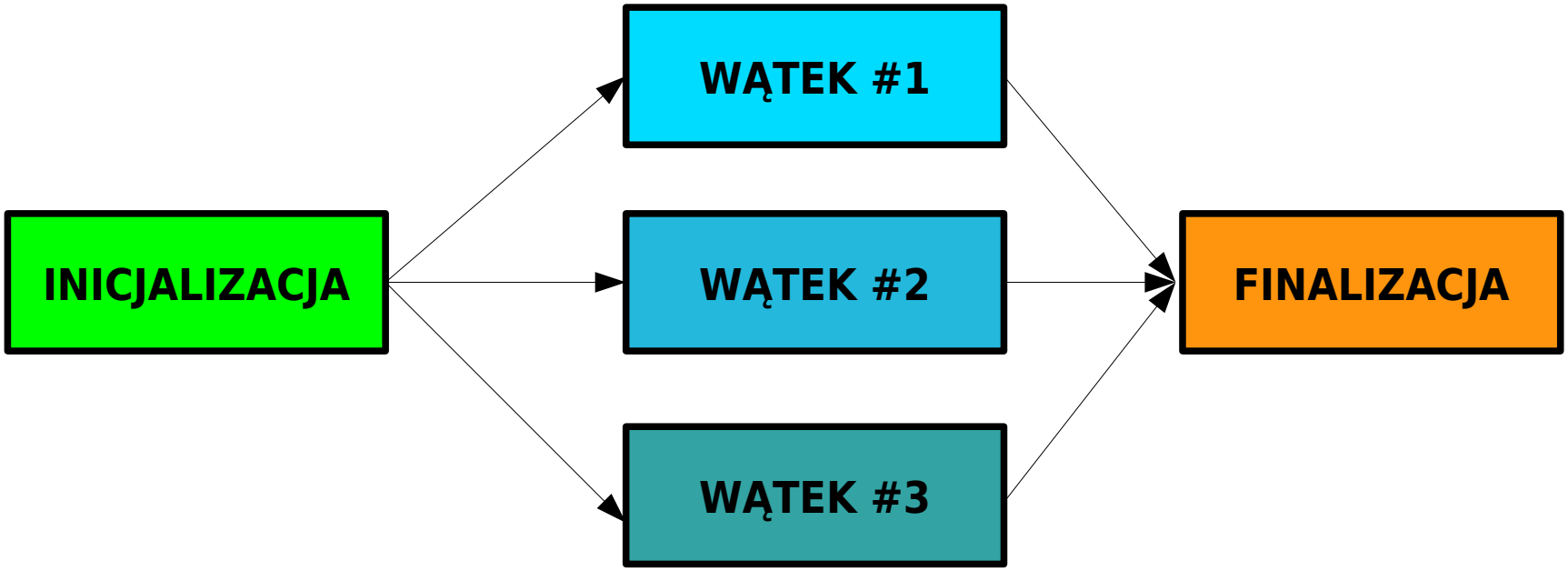
Obliczenia wielowątkowe

Idea

Implementacja



Program wielowątkowy



Nie wszystkie etapy programu można wykonywać wielowątkowo (np. wczytywanie dużego pliku)!
Etapy jedno i wielowątkowe mogą się wielokrotnie przeplatać.

std::thread

```
#include <thread>

void funkcja_drugiego_watku ()
{
        // (…) robi coś
}

int main ()
{
        // (…) wczytujemy dane itp.
        // otwieramy drugi wątek:
        std::thread drugi_watek (funkcja_drugiego_watku);
        // (…) coś dalej miziamy
        drugi_watek.join ();    // czekamy na zakończenie drugiego wątku
        // (…) kończymy obliczenia jednowątkowo
        return 0;
}
```

Zalety: obliczenie równoległe na ogół wykonują się szybko
Wady: funkcja_drugiego_wątku musi być void (void) – problemy ze sterowaniem wątkiem i przechwytywaniem jego wyników (konieczność użycia zmiennych globalnych).

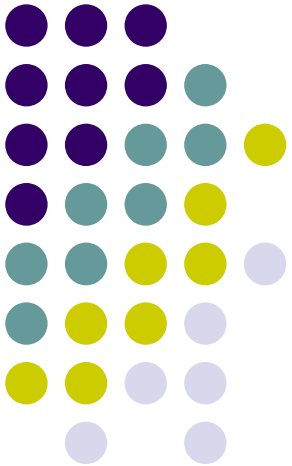
std::mutex

```
#include <mutex>

float tablica [100];    // tablica globalna

class KlasaWspoldzielona
{
        public:
                void kwadrat (int i) {
                        blokada.lock ();        // zakładanie blokady
                        tablica [i] *= tablica [i];
                        blokada.unlock ();      // zdejmowanie blokady
                }
        private:
                std::mutex blokada;
};
```

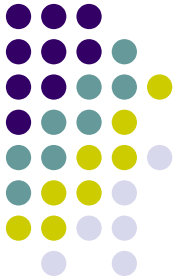
Blokada blokuje sekcję na wyłączność dla jednego wątku. Pozostałe wątki czekają w kolejce na zwolnienie blokady.
Dla wygody można używać std::unique_lock

Przykład

Obliczanie wartości średniej

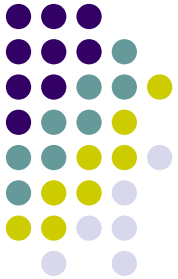


srednia.hpp



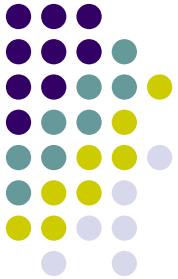
```
#ifndef _srednia_hpp_
#define _srednia_hpp_
#include <vector>

typedef std::vector<double>::const_iterator iter;

class Srednia
{
  public:
    // Konstruktor zapamiątuje „jego” zakres tablicy
    Srednia (const iter &pocz, const iter &konc);
    double wartosc () const {return srednia_;}
    void operator() (); // wykonuje obliczenia
    int ilosc () const {return n_;}
  private:
    double srednia_;
    int n_;     // liczba elementów w zakresie
    iter pocz_; // początek zakresu
    iter konc_; // pierwszy element za zakresem
};

#endif
```

srednia.cpp



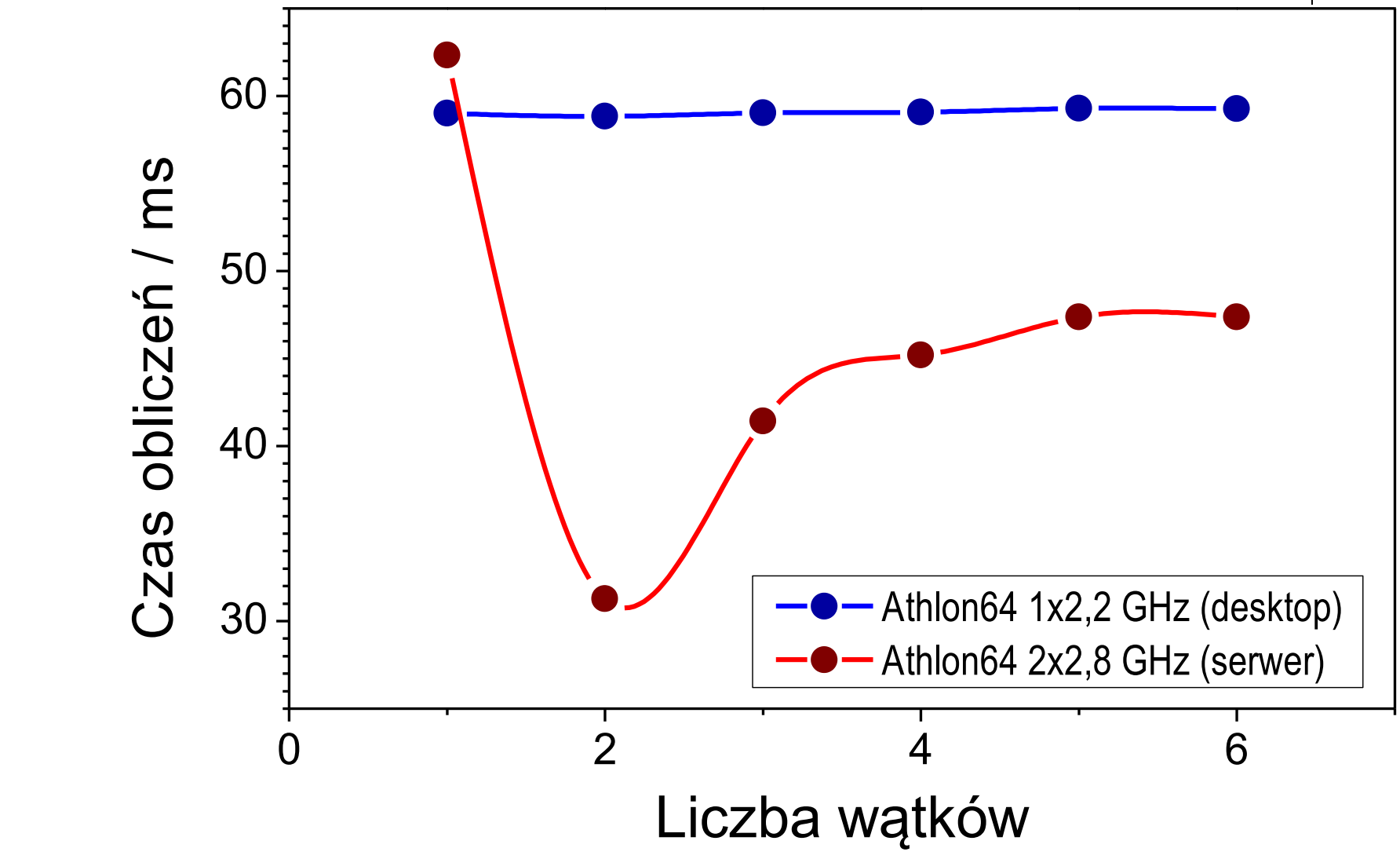
```
#include "srednia.hpp"
#include <algorithm>
#include <boost/lambda/lambda.hpp>

Srednia::Srednia (const iter &pocz, const iter &konc) :
        pocz_ (pocz), konc_ (konc)
{
  n_ = konc_ - pocz_;
}

void Srednia::operator() ()     // wykonuje obliczenia
{
  srednia_ = 0.0;
  std::for_each (pocz_, konc_, srednia_ += boost::lambda::_1);
  srednia_ /= n_;
}
```

main.cpp



```
#include <boost/thread.hpp>
#include "srednia.hpp"

double licz_srednia (const std::vector <double> &dane, int p) // p – liczba wątków
{
  std::vector <Srednia*> srednie;
  boost::thread_group watki;

  int n = dane.size();

  for (int i = 0; i < p; ++i) // przydziela zakresy poszczególnym obiektom
    srednie.push_back (new Srednia (dane.begin() + n*i/p, dane.begin() + n*(i+1)/p));
  for (int i = 0; i < p; ++i) // tworzy wątki
    watki.create_thread (std::ref (*(srednie [i])));


  watki.join_all (); // czeka na zakończenie wszystkich obliczeń

  double srednia = 0.0;
  for (int i = 0; i < p; ++i) // oblicza średnią średnich
    srednia += srednie[i]->wartosc() * srednie[i]->ilosc();
  srednia /= dane.size ();

  for (int i = 0; i < p; ++i)
    delete srednie[i];

  return srednia;
}
```

Testy wydajności

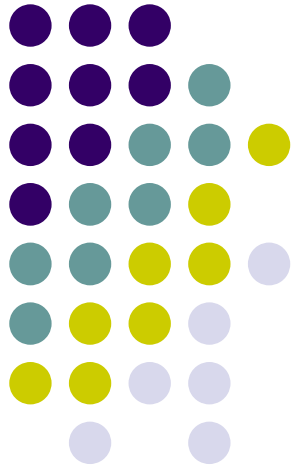


std::chrono

He who controls the past commands the future

He who commands the future, conquers the past.

Kane



C++11

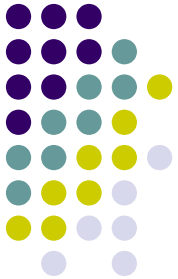
Dostępne zegary

```
#include <chrono>

using namespace std::chrono;
```

Zegar	Opis
system_clock	Wall clock time from the system-wide realtime clock
steady_clock	Monotonic clock that will never be adjusted
high_resolution_clock	The clock with the shortest tick period available

Metody *now*() ww. klas zwracają obiekt daty/czasu typu *std::chrono::time_point*.

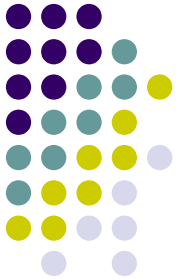
time_point / duration – przykłady



```cpp
#include <iostream>
#include <chrono>
#include <ctime>

int main()
{
    std::chrono::time_point<std::chrono::system_clock> start, end;
    start = std::chrono::system_clock::now();
    std::cout << "f(42) = " << fibonacci(42) << '\n';
    end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(end);

    std::cout << "finished computation at " << std::ctime(&end_time)
              << "elapsed time: " << elapsed_seconds.count() << "s\n";
}
```

Więcej na:
http://en.cppreference.com/w/cpp/chrono

std::literals::chrono_literals



D

d (since C++20)

H

h (since C++14)

M

min (since C++14)
ms (since C++14)

N

ns (since C++14)

S

s (since C++14)

U

us (since C++14)

Y

y (since C++20)

Helper types

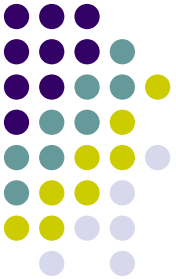
Type	Definition
std::chrono::nanoseconds	duration</*signed integer type of at least 64 bits*/, std::nano>
std::chrono::microseconds	duration</*signed integer type of at least 55 bits*/, std::micro>
std::chrono::milliseconds	duration</*signed integer type of at least 45 bits*/, std::milli>
std::chrono::seconds	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.

Each of the predefined duration types days, weeks, months and years covers a range of at least ±40000 years. years is equal to 365.2425 days (the average length of a Gregorian year). months is equal to 30.436875 days (exactly 1/12 of years). (since C++20)

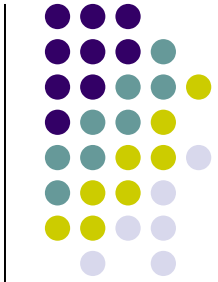
Więcej na:
https://en.cppreference.com/w/cpp/chrono/duration
https://en.cppreference.com/w/cpp/symbol_index/chrono_literals

Przykłady (1)



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

constexpr auto year = 31556952ll; // seconds in average Gregorian year

int main()
{
    using shakes = std::chrono::duration<int, std::ratio<1, 100000000>>;
    using jiffies = std::chrono::duration<int, std::centi>;
    using microfortnights = std::chrono::duration<float, std::ratio<14*24*60*60, 1000000>>;
    using nanocenturies = std::chrono::duration<float, std::ratio<100*year, 1000000000>>;

    std::chrono::seconds sec(1);

    std::cout << "1 second is:\n";

    // integer scale conversion with no precision loss: no cast
    std::cout << std::chrono::microseconds(sec).count() << " microseconds\n"
              << shakes(sec).count() << " shakes\n"
              << jiffies(sec).count() << " jiffies\n";

    // integer scale conversion with precision loss: requires a cast
    std::cout << std::chrono::duration_cast<std::chrono::minutes>(sec).count()
              << " minutes\n";

    // floating-point scale conversion: no cast
    std::cout << microfortnights(sec).count() << " microfortnights\n"
              << nanocenturies(sec).count() << " nanocenturies\n";
}
```

Przykłady (2)



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

int main()
{
    using namespace std::chrono_literals;
    auto day = 24h;
    auto halfhour = 0.5h;
    std::cout << "one day is " << day.count() << " hours\n"
              << "half an hour is " << halfhour.count() << " hours\n";
}
```

Output:

```
one day is 24 hours
half an hour is 0.5 hours
```

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

int main()
{
    using namespace std::chrono_literals;
    auto halfmin = 30s;
    std::cout << "half a minute is " << halfmin.count() << " seconds\n"
              << "a minute and a second is " << (1min + 1s).count() << " seconds\n";
}
```

Output:

```
half a minute is 30 seconds
a minute and a second is 61 seconds
```

Programowanie jest fantastyczne!!!