C++17 i STL

Ograniczenia liczbowe
Statyczna arytmetyka liczb wymiernych
Cechy typowe
Czasomierze

Ograniczenia liczbowe

- Typy numeryczne posiadają ograniczenia zależne od platformy – ograniczenia te dostępne są w bibliotece standardowej C++ we wzorcu `numeric_limits<>`.
- Nowa koncepcja ograniczeń liczbowych ma dwie zalety:
  - oferuje większe bezpieczeństwo typologiczne;
  - umożliwia programiście definiować wzorce, które obliczają te ograniczenia.
- Klasa `numeric_limits<>` jest szablonem, który można wykorzystać do realizacji wspólnego interfejsu, jaki będzie zaimplementowany dla każdego typu numerycznego (całkowitoliczbowego i zmiennopozycyjnego) poprzez specjalizację:

```
template <typename T> class numeric_limits { … };
template <> class numeric_limits<int> { … };
…
```

Ograniczenia liczbowe

- Ogólny wzorzec `numeric_limits<>` oraz jego specjalizacje zdefiniowane są w pliku nagłówkowym `<limits>`.
- Specjalizacje zdefiniowane są dla wszystkich typów podstawowych, które mogą reprezentować wartości liczbowe: bool, char, signed char, unsigned char, char16_t, char32_t, wchar_t, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, long double.
- Można je łatwo uzupełnić specjalizacjami dla typów definiowanych przez użytkownika.
- Wszystkie składowe we wzorcu `numeric_limits<>` są deklarowane jako stałowyrażeniowe ze słowem `constexpr`.

Ograniczenia liczbowe

Składowa	Znaczenie
`is_signed`	Typ ze znakiem.
`is_integer`	Typ jest całkowity.
`is_exact`	Obliczenia nie generują błędów zaokrągleń (wartość `true` dla wszystkich typów całkowitych).
`is_bounded`	Zbiór reprezentowalnych wartości jest skończony (wartość `true` dla wszystkich typów wbudowanych).
`is_modulo`	Dodanie dwóch liczb dodatnich może w wyniku dać mniejszą wartość (w przypadku przekroczenia zakresu).

Ograniczenia liczbowe

Składowa	Znaczenie
`min()`	Minimalna wartość skończona (dla typów zmiennoprzecinkowych z denormalizacją jest to minimalna dodatnia wartość znormalizowana; ma znaczenie, jeśli is_bounded || !is_signed).
`max()`	Maksymalna wartość skończona (ma znaczenie, jeśli is_bounded).
`lowest()`	Maksymalna ujemna wartość skończona (ma znaczenie, jeśli is_bounded).
`has_infinity`	Typ posiada reprezentację dla dodatniej nieskończoności.
`infinity()`	Reprezentacja dodatniej nieskończoności (jeśli jest dostępna).

Ograniczenia liczbowe

Składowa	Znaczenie
`digits`	Typ znakowy i całkowity: liczba bitów (cyfr binarnych) bez znaku. Typ zmiennoprzecinkowy: liczba cyfr radix (patrz niżej) w mantysie.
`digits10`	Liczba cyfr dziesiętnych (ma znaczenie, jeśli is_bounded).
`radix`	Typ całkowity: podstawa reprezentacji (praktycznie zawsze 2). Typ zmiennoprzecinkowy: podstawa reprezentacji wykładniczej.
`min_exponent`	Minimalny ujemny wykładnik całkowity dla podstawy radix.
`max_exponent`	Maksymalny dodatni wykładnik całkowity dla podstawy radix.
`epsilon()`	Różnica wartości jeden i najmniejszej wartości większej od jednego.

Ograniczenia liczbowe

- Przykłady:

```
cout << "max(long): "
     << numeric_limits<long>::max()
     << endl;
     // max(long): 2147483647
cout << "is_signed(char): "
     << numeric_limits<char>::is_signed
     << endl;
     // is_signed(char): true
cout << "is_specialized(string): "
     << numeric_limits<string>::is_specialized
     << endl;
     // is_specialized(string): false
```

Cechy typowe

- Cechy typowe (ang. type traits) stanowią mechanizm definiowania zachowania zależnego od typu – można je wykorzystywać do optymalizacji kodu dla tych typów, które posiadają pewne właściwości (cechy).
- Cecha typowa jest środkiem reagowania na właściwości przynależne typowi – jest to szablon, który w czasie kompilacji określa jakiś typ albo wartość w zależności od jednego bądź wielu argumentów szablonu (zazwyczaj argumentów typowych).
- Cechy typowe są zazwyczaj definiowane w pliku nagłówkowym `<type_traits>`.

Cechy typowe

Cecha typowa	Wynik
is_void<T>	Typ void.
is_integral<T>	Typ całkowitoliczbowy (z typami bool, char, char16_t, char32_t i wchar_t włącznie).
is_floating_point<T>	Typ zmiennoprzecinkowy (float, double, long double).
is_arithmetic<T>	Typ liczbowy (zmiennoprzecinkowy albo całkowitoliczbowy, z typem bool i typami znakowymi włącznie).
is_const<T>	Typ z kwalifikatorem const.
is_volatile<T>	Typ z kwalifikatorem volatile.
is_array<T>	Klasyczny typ tablicowy (nie std::array).
is_enum<T>	Typ wyliczeniowy.

Cechy typowe – elastyczne przeciążanie dla typów wskaźnikowych

- Załóżmy, że definiujemy funkcję `foo()`, która powinna być różnie implementowana dla argumentów, które są wartościami i dla argumentów, które są wskaźnikami.
- Dzięki cechom typowym można skonstruować taką implementację:

```
template <typename T>
void foo_impl (const T &val, true_type) {
    cout << "foo() ze wskaźnikiem do " << *val << endl;
}
template <typename T>
void foo_impl (const T &val, false_type) {
    cout << "foo() z wartoscia " << val << endl;
}
template <typename T>
void foo (T val) {
    foo_impl (val, std::is_pointer<T>());
}
```

Cechy typowe

Cecha typowa	Wynik
is_class<T>	Klasa lub struktura, ale nie unia.
is_union<T>	Unia.
is_pointer<T>	Typ wskaźnikowy (ze wskaźnikami do funkcji włącznie, ale bez wskaźników do niestatycznych składowych klas).
is_reference<T>	Referencja do l-wartości albo r-wartości.
is_lvalue_reference<T>	Referencja do l-wartości.
is_rvalue_reference<T>	Referencja do r-wartości.
is_scalar<T>	Typ skalarny (całkowitoliczbowy, z typem bool i typami znakowymi włącznie, zmiennoprzecinkowy, wyliczeniowy, wskaźnik, wskaźnik do składowej, std::nullptr_t).

Cechy typowe – elastyczne przeciążanie dla typów całkowitoliczbowych

- Załóżmy, że definiujemy funkcję `foo()`, która powinna być różnie implementowana dla argumentów typów całkowitoliczbowych i argumentów typów zmiennoprzcinkowych.
- Dzięki cechom typowym można skonstruować taką implementację:

```
template <typename T>
void foo_impl (T val, true_type) { … }
template <typename T>
void foo_impl (T val, false_type) { … }
template <typename T>
void foo (T val) {
    foo_impl (val, std::is_integral<T>());
}
```

Cechy typowe

Cecha typowa	Wynik
is_same<T1, T2>	T1 i T2 to ten sam typ (z dokładnością do kwalifikatorów const/volatile).
is_base_of<B, D>	B jest klasą bazową typu D.
is_convertible<T1, T2>	T1 daje się konwertować na typ T2.
is_constructible<T, Args...>	Wartość typu T może być inicjalizowana argumentami Args.
is_assignable<T1, T2>	Można przypisać wartość typu T2 do wartości typu T1.
conditional<B, T, F>	Zwraca T, jeśli stałowartościowe wyrażenie B ma wartość true, F w przeciwnym razie.
common_type<T, ...>	Typ wspólny przekazanych typów.

Cechy typowe

Cecha typowa	Wynik
is_polymorphic<T>	Klasa z wirtualną funkcją składową albo jej klasa pochodna.
is_abstract<T>	Klasa abstrakcyjna (z przynajmniej jedną czysto wirtualną funkcją składową).
has_virtual_destructor<T>	Klasa z wirtualnym destruktorem.
is_default_constructible<T>	Klasa pozwalająca na konstrukcję bezparametrową (konstruktorem domyślnym).
is_copy_constructible<T>	Klasa pozwalająca na konstrukcję kopiującą.
is_move_constructible<T>	Klasa pozwalająca na konstrukcję przenoszącą.

Cechy typowe – sprowadzanie do wspólnego typu

- Jedną z użyteczności cech typowych jest sprowadzanie dwóch lub więcej typów do typu wspólnego – jest to typ (jeśli taki istnieje), który pozwoli na przeprowadzenie poprawnej semantycznie operacji na wartościach dwóch różnych typów.
- Przykład dla funkcji `min()` z argumentami różnych typów:

```
template <typename T1, typename T2>
typename std::common_type<T1, T2>::type
min (const T1& x, const T2& y);
```

Statyczna arytmetyka liczb wymiernych

- Począwszy od C++11, biblioteka standardowa udostępnia interfejs do definiowania statycznych (obliczanych w czasie kompilacji) wyrażeń arytmetycznych na liczbach wymiernych (ułamkowych).
- Narzędzia obliczeń statycznych na liczbach wymiernych definiowane są w pliku nagłówkowym <ratio>, z klasą ratio<>.
- Klasa ratio<> wychwytuje wszystkie błędy obliczeń (w rodzaju przepełnienia zakresu wartości czy dzielenia przez zero) już w czasie kompilacji. Jest wykorzystywana w bibliotekach duration<> i time_point<> do efektywnego tworzenia jednostek czasu.

Statyczna arytmetyka liczb wymiernych

- Definiicja klasy ratio<>:

```
namespace std {
    template <intmax_t N, intmax_t D = 1>
    class ratio {
    public:
        typedef ratio<num,den> type;
        static constexpr intmax_t num;
        static constexpr intmax_t den;
    };
}
```

- Typ intmax_t to typ liczby całkowitej ze znakiem, zdolny do reprezentowania dowolnej wartości dowolnego typu całkowitoliczbowego ze znakiem – jest on zdefiniowany w pliku nagłówkowym <cstdint> jako typ o rozmiarze co najmniej 64 bitów.
- Licznik (składowa num, od ang. numerator) i mianownik (składowa den, od ang. denominator) są składowymi publicznymi, a ich wartości są automatycznie skracane do jak najmniejszych.

Statyczna arytmetyka liczb wymiernych

- Można wykonywać obliczenia statyczne na wartościach typu ratio – cztery podstawowe statyczne działania arytmetyczne (suma, różnica, iloczyn i iloraz) są zdefiniowane jako ratio_add, ratio_subtract, ratio_multiply i ratio_divide. Typ wynikowy operacji tych operacji to zawsze ratio<>. Przykład:

```
ratio_add<ratio<2,7>, ratio<2,6>>::type
// wynik: ratio<13,21>
```

- Dwa typy ułamkowe ratio<> można ze sobą porównywać (operacje ratio_equal i ratio_not_equal) i ustalać relacje między nimi (ratio_less, ratio_less_equal, ratio_greater i ratio_greater_equal). Typ wynikowy operacji tych operacji to wartość logiczna true_type bądź false_type, 5.4.2), a jego składowa value ma stałą wartość true albo false. Przykład:

```
ratio_equal<ratio<5,3>, ratio<25,15>>::value
// wynik: true
```

Statyczna arytmetyka liczb wymiernych

- Dla wygody podawania bardzo dużych i bardzo małych wartości ułamkowych udostępniono predefiniowane symbole, dzięki którym można określać duże wartości ułamkowe bez żmudnego liczenia wpisanych zer.
- Predefiniowane jednostki dla potęg liczby 10 to: deca (10), hecto (100), kilo (1000), mega (1000000), giga, tera, peta, exa, zetta, yotta.
- Predefiniowane jednostki dla potęg liczby 0.1 to: deci (0.1), centi (0.01), milli (0.001), micro (0.000001), nano, pico, femto, atto, zepto, yocto.
- Na przykład std::nano jest równoważne std::ratio<1,1000000000>.

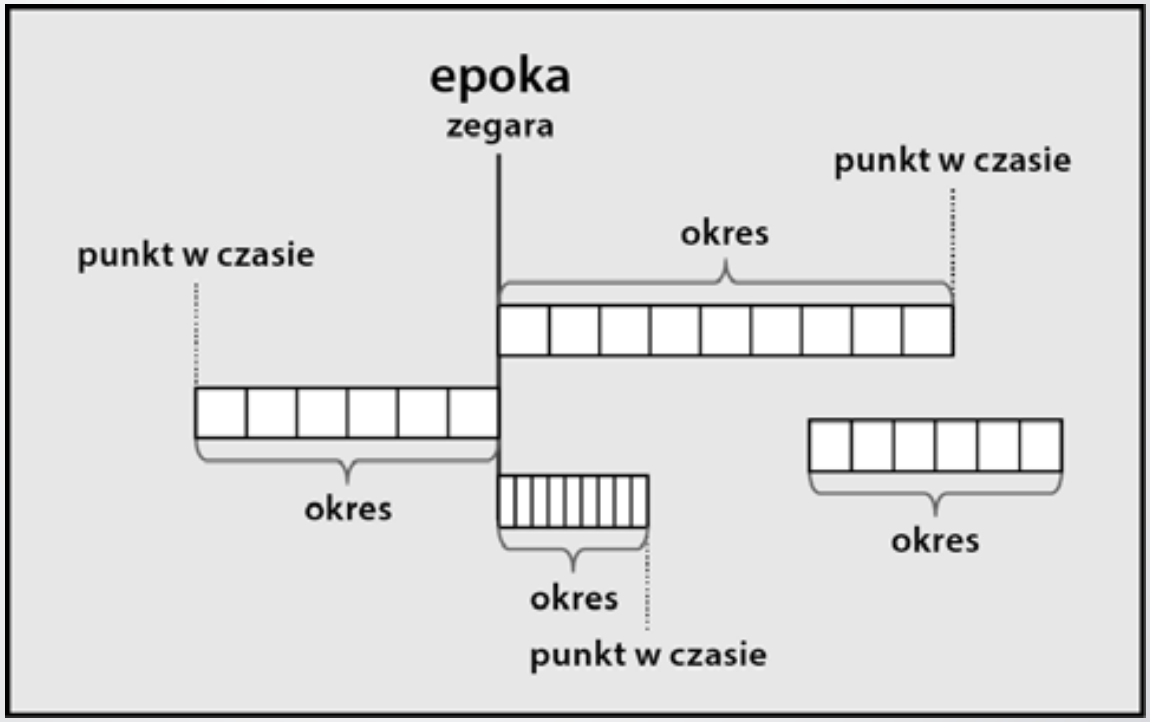
Obsługa dat i pomiar upływu czasu

- W przeszłości podstawowym interfejsem obsługi czasu systemowego były funkcje języka C i standardu POSIX, których rozdzielczość była stopniowo zwiększana z sekund do milisekund, potem do mikrosekund i wreszcie do nanosekund.
  - Niestety, każde zwiększenie precyzji oznaczało udostępnienie nowego interfejsu.
- W C++11 zaprojektowano bibliotekę neutralną wobec precyzji pomiaru i reprezentacji czasu, opisaną w pliku nagłówkowym <chrono>.
  - Biblioteka standardowa C++ udostępnia także klasyczne interfejsy języka C i standardu POSIX do obsługi dat kalendarzowych.
  - Ponadto w bibliotece wątków, obecnej w standardzie począwszy od C++11, jest możliwość zawieszenia wykonania wątku albo programu głównego (wątku funkcji main()) w oczekiwaniu na upływ okresu czasu.
  - Definicje wszystkich narzędzi do obsługi dat i czasu zostały umieszczone w przestrzeni nazw std::chrono.
  - Podstawową jednostką wyrażania czasu jest sekunda.

Przegląd biblioteki <chrono>

- Biblioteka <chrono> została zaprojektowana pod kątem możliwości operowania zegarami i czasomierzami o różnej precyzji reprezentacji czasu.
  - Celem było odcięcie się od problemu zmiennej precyzji pomiaru czasu.
- Okres – określona liczba taktów wybranej jednostki czasu.
- Punkt w czasie – kombinacja okresu i punktu wyznaczającego początek pomiaru upływu czasu (tzw. epoka).
- Zegarem – obiekt definiujący epokę określającą punkt w czasie. Różne zegary mogą być osadzone w różnych epokach.
  - Operacje angażujące wiele punktów w czasie, takie jak wyznaczanie okresu (różnicy pomiędzy dwoma punktami w czasie), wymagają stosowania zegarów z tej samej epoki.

Przegląd biblioteki <chrono>



Okresy

- Okres jest obiektem klasy duration<>.
- Okres jest kombinacją wartości reprezentującej liczbę taktów zegara oraz ułamka reprezentującego jednostkę upływu czasu (wyrażoną w sekundach).
- Do określania ułamka jednostki upływu czasu wykorzystywana jest klasa arytmetyki statycznej ratio<>.
- Przykłady:

```
std::chrono::duration<int>
    twentySeconds(20);
std::chrono::duration<double,std::ratio<60>>
    halfAMinute(0.5);
std::chrono::duration<long,std::ratio<1,1000>>
    oneMillisecond(1);
```

Okresy

- Pierwszy argument konkretyzacji szablonu duration<> określa typ taktu, opcjonalny drugi argument definiuje typ jednostki określającej ułamek jednostki upływu czasu.
- Tak więc pierwszy zdefiniowany powyżej okres operuje sekundami, drugi operuje minutami (60/1 sekund) a trzeci milisekundami (1/1000 sekundy).

Okresy

- Na okresach można wykonywać pewne obliczenia (arytmetyczne, relacyjne):
  - wyznaczać sumę, różnicę, iloczyn i iloraz dwóch okresów;
  - dodawać lub odejmować takty albo inne okresy;
  - porównywać dwa okresy.
- Jednostki czasu określające okresy uczestniczące w operacji mogą być różne – okres wynikowy będzie wyrażony jednostką będącą największym wspólnym dzielnikiem jednostek obu operandów.
- Przykład:

```
chrono::seconds d1(42); // 42 sekundy
chrono::milliseconds d2(10); // 10 milisekund
…
d1 – d2 // wynik: 41990 taktów milisekundowych
```

Okresy

- Możliwe jest też konwertowanie okresów na okresy wyrażane w innych jednostkach, o ile tylko istnieje niejawna konwersja typu liczbowego jednostki.
- Przykład:

```
chrono::seconds twentySeconds(20); // 20 sekund
chrono::hours aDay(24); // 24 godziny
chrono::milliseconds ms; // 0 milisekund
ms += aDay;
    // 86400000 milisekund
--ms; // 86399999 milisekund
ms *= 2; // 172839998 milisekund
```

Okresy

- Liczbę taktów okresu określaliśmy za pomocą funkcji składowej count() .
- Klasa duration<> definiuje też trzy funkcje statyczne: zero() zwracającą okres o długości 0 sekund, a także min() i max() zwracające odpowiednio okres o minimalnej i maksymalnej długości danego typu.
- Z klasy duration<> można również pobrać typ taktu za pomocą duration::rep oraz typ jednostki czasu za pomocą duration::period.
- Operator przypisania przypisuje jeden okres *do drugiego* z ewentualną niejawną konwersją typu okresu.

Okresy

- Niejawne konwersje z typu mniej precyzyjnego na bardziej precyzyjny są zawsze możliwe, ale już konwersje na typ mniej precyzyjny muszą być jawne, bo potencjalnie wiążą się z utratą informacji. Na przykład zamiana wartości całkowitoliczbowej 42010 milisekund na sekundy dałaby wartość 42, co oznacza utratę dokładności 10 milisekund.
- Mimo to jawnie taka konwersja może być wymuszona za pośrednictwem szablonu duration_cast<>.
- Przykład:

```
chrono::seconds sec(55);
chrono::minutes m1 = sec; // Błąd
chrono::minutes m2 =
    chrono::duration_cast<chrono::minutes>(sec); // OK
```

Okresy

- Typowym przykładem konieczności użycia podobnych konwersji jest kod dzielący okres na segmenty różnych jednostek, jak w poniższym przykładzie, gdzie okres wyrażany milisekundami jest dzielony na segmenty wyrażane w pełnych godzinach, minutach, sekundach i wreszcie milisekundach:

```
// przykładowy czas
milliseconds ms(7255042);
…
// podziel na godziny, minuty, sekundy i milisekundy
hours hh = duration_cast<hours>(ms);
minutes mm =
    duration_cast<minutes>(ms % chrono::hours(1));
seconds ss =
    duration_cast<seconds>(ms % chrono::minutes(1));
milliseconds msec =
    duration_cast<milliseconds>(ms % chrono::seconds(1));
```

Zegary i punkty w czasie

- Zegar jest obiektem klasy clock<>.
- Zegar definiuje epokę (początkowy punkt w czasie) i długość taktu. Dodatkowo zegar określa typ punktów w czasie określanych zgodnie z tym zegarem.
- Interfejs zegara definiuje funkcję składową now() zwracającą obiekt bieżącego punktu w czasie.
- Punkt w czasie reprezentuje konkretny moment wyrażony jako dodatni albo ujemny okres względem danego zegara.

Zegary i punkty w czasie

- Interfejs punktu w czasie daje możliwość określania epoki oraz minimalnego i maksymalnego punktu w czasie dostępnego dla danego zegara, a także umożliwia arytmetykę punktów w czasie.
- Z klasy clock<> można również pobrać typ okresu zegara za pomocą clock::duration, typ taktu zegara za pomocą clock::rep oraz typ jednostki czasu za pomocą clock::period.
- Typ punktu w czasie danego zegara możemy pobrać za pomocą clock::time_point.
- Za pomocą clock::is_steady określa, czy zegar jest monotoniczny.

Zegary

- Biblioteka standardowa C++ definiuje trzy zegary:
  - Zegar systemowy **system_clock** reprezentuje punkty w czasie odpowiadające typowemu zegarowi czasu rzeczywistego danego systemu. Zegar ten definiuje również funkcje pomocnicze to_time_t() i from_time_t() do konwersji pomiędzy dowolnymi punktami w czasie a systemowym typem reprezentacji czasu w języku C time_t (innymi słowy, funkcje te pozwalają na konwersje pomiędzy punktami w czasie a czasem kalendarzowym).
  - Zegar miarowy **steady_clock** gwarantuje monotoniczność, co oznacza, że kolejne punkty w czasie generowane z tego zegara będą miały rosnące a w każdym razie niemalejące wartości (zegar nie podlega korektom cofającym). Do tego wartości punktów w czasie przyrastają w równym tempie względem upływu czasu rzeczywistego.
  - Zegar wysokiej rozdzielczości **high_resolution_clock** reprezentujący czas rzeczywisty z najkrótszym możliwym taktem dostępnym dla danego systemu.

Zegary

- Przykład pomiaru czasu:

```
auto poczatek =
    chrono::high_resolution_clock::now();
// … procedura obliczeniowa …
auto koniec =
    chrono::high_resolution_clock::now();

// obliczenie i wypisanie czasu działania procedury
auto dlugosc = koniec - poczatek;
auto sek =
    chrono::duration_cast<chrono::seconds>(diff);
cout << "program dzialal przez: „
    << sek.count() << " sekund" << endl;
```

Punkty w czasie

- Każdy z zegarów (gotowy biblioteczny a także zegar definiowany przez programistę) pozwala na generowanie punktów w czasie.
- Punkt w czasie jest reprezentowany klasą time_point<> o następującym interfejsie parametryzowanym typem zegara:

```
namespace std {
    namespace chrono {
        template <typename Clock,
            typename Duration = Clock::duration>
        class time_point;
    }
}
```

Punkty w czasie

- Cztery punkty w czasie pełnią szczególną rolę:
  - **Epoka**, czyli punkt w czasie zwracany przez konstruktor domyślny klasy time_point, odpowiednio do zegara.
  - **Czas bieżący**, pobierany ze statycznej funkcji składowej now() każdego z zegarów;
  - **Minimalny punkt** w czasie, zwracany przez statyczną funkcję składową min() klasy time_point, odpowiednio do zegara.
  - **Maksymalny punkt** w czasie, zwracany przez statyczną funkcję składową max() klasy time_point, odpowiednio do zegara.

Punkty w czasie

- Przykładowy program przypisuje punkty w czasie do zmiennej tp,
  a potem konwertuje je na czas kalendarzowy i wypisuje na wyjście:

```
string asString (const chrono::system_clock::time_point& tp)
{
    // konwersja na czas systemowy:
    time_t t = chrono::system_clock::to_time_t(tp);
    string ts = ctime(&t); // konwersja na czas kalendarzowy
    ts.resize(ts.size() - 1); // pomijanie odstępów z końca
    return ts;
}
int main()
{
    // wypisz epokę zegara systemowego:
    chrono::system_clock::time_point tp;
    cout << "epoka: " << asString(tp) << endl;
    // wypisz bieżący czas zegara systemowego:
    tp = chrono::system_clock::now();
    cout << "teraz: " << asString(tp) << endl;
    // wypisz minimalny czas zegara systemowego:
    tp = chrono::system_clock::time_point::min();
    std::cout << "minimum: " << asString(tp) << std::endl;
    // wypisz maksymalny czas zegara systemowego:
    tp = chrono::system_clock::time_point::max();
    cout << "maksimum: " << asString(tp) << endl;
}
```

Punkty w czasie

- Obiekty klasy time_point<> posiadają tylko jedną składową reprezentującą okres wyrażony względem epoki zegara, z którego uzyskano punkt czasowy.
- Wartość punktu czasowego może być pozyskana za pośrednictwem wywołania time_since_epoch().
- W zakresie arytmetyki punktów w czasie dostępne są wszystkie praktyczne kombinacje punktów w czasie i punktów w czasie z okresami.

Funkcje daty i czasu języka C i standardu POSIX

- Biblioteka standardowa C++ udostępnia standardowe interfejsy POSIX i C do obsługi dat i czasu.
- Makrodefinicje, typy i funkcje w pliku nagłówkowym języka C <time.h> w bibliotece standardowej C++ są deklarowane w przestrzeni nazw std i dostępne po dołączeniu pliku nagłówkowego <ctime>.
- Poza nimi biblioteka standardowa definiuje także makrodefinicję CLOCKS_PER_SEC, określającą jednostkę upływu czasu dla funkcji clock().

Konwersje pomiędzy punktami w czasie a czasem kalendarzowym

```
// konwertuj punkt w czasie zegara systemowego
// na łańcuch czasu kalendarzowego
inline string asString (const
chrono::system_clock::time_point& tp)
{
    // konwersja na czas systemowy:
    time_t t = chrono::system_clock::to_time_t(tp);
    // konwertuj na czas kalendarzowy
    string ts = ctime(&t);
    // wytnij znak nowego wiersza z końca
    ts.resize(ts.size()-1);
    return ts;
}
```

Konwersje pomiędzy punktami w czasie a czasem kalendarzowym

```
// konwertuj czas kalendarzowy na punkt
// w czasie zegara systemowego
inline chrono::system_clock::time_point makeTimePoint
(int year, int mon, int day, int hour, int min, int
sec=0)
{
    struct tm t;
    t.tm_sec = sec; // sekunda w minucie (0 .. 59)
    t.tm_min = min; // minuta w godzinie (0 .. 59)
    t.tm_hour = hour; // godzina w dniu (0 .. 23)
    t.tm_mday = day; // dzień w miesiącu (1 .. 31)
    t.tm_mon = mon-1; // miesiąc w roku (0 .. 11)
    t.tm_year = year-1900; // rok, liczony od 1900
    t.tm_isdst = -1; // sprawdź, czy czas jest letni
    time_t tt = mktime(&t);
    if (tt == -1) throw „błąd w czasie systemowym";
    return chrono::system_clock::from_time_t(tt);
}
```

Czasowe wstrzymywanie wykonania

- Okresy i punkty w czasie można wykorzystywać nie tylko do mierzenia upływu czasu w programie, ale również do czasowego wstrzymywania wykonywania wątków (albo całego programu, to znaczy wątku funkcji main()).
- Takie wstrzymanie może być bezwarunkowe, ale może też służyć do ustalenia maksymalnego okresu oczekiwania na zwolnienie muteksu, na zmianę wartości zmiennej warunkowej czy na zakończenie wykonania innego wątku.

Czasowe wstrzymywanie wykonania

- Funkcje sleep_for() i sleep_until() z przestrzeni nazw this_thread służą do blokowania wykonania wątku na zadany okres albo do zadanego punktu w czasie.
- Przykład 1: wstrzymanie wykonywania bieżącego wątku (może to być wątek główny programu) na okres 10 sekund.

```
this_thread::sleep_for(chrono::seconds(10));
```

- Przykład 2: wstrzymanie wykonania bieżącego wątku do momentu, aż zegar systemowy osiągnie punkt w czasie, ustalony jako odległy o 10 sekund od bieżącego.

```
this_thread::sleep_until(
    chrono::system_clock::now() +
    chrono::seconds(10)
);
```