Programowanie Obiektowe

Marcin Kamil Bączyk

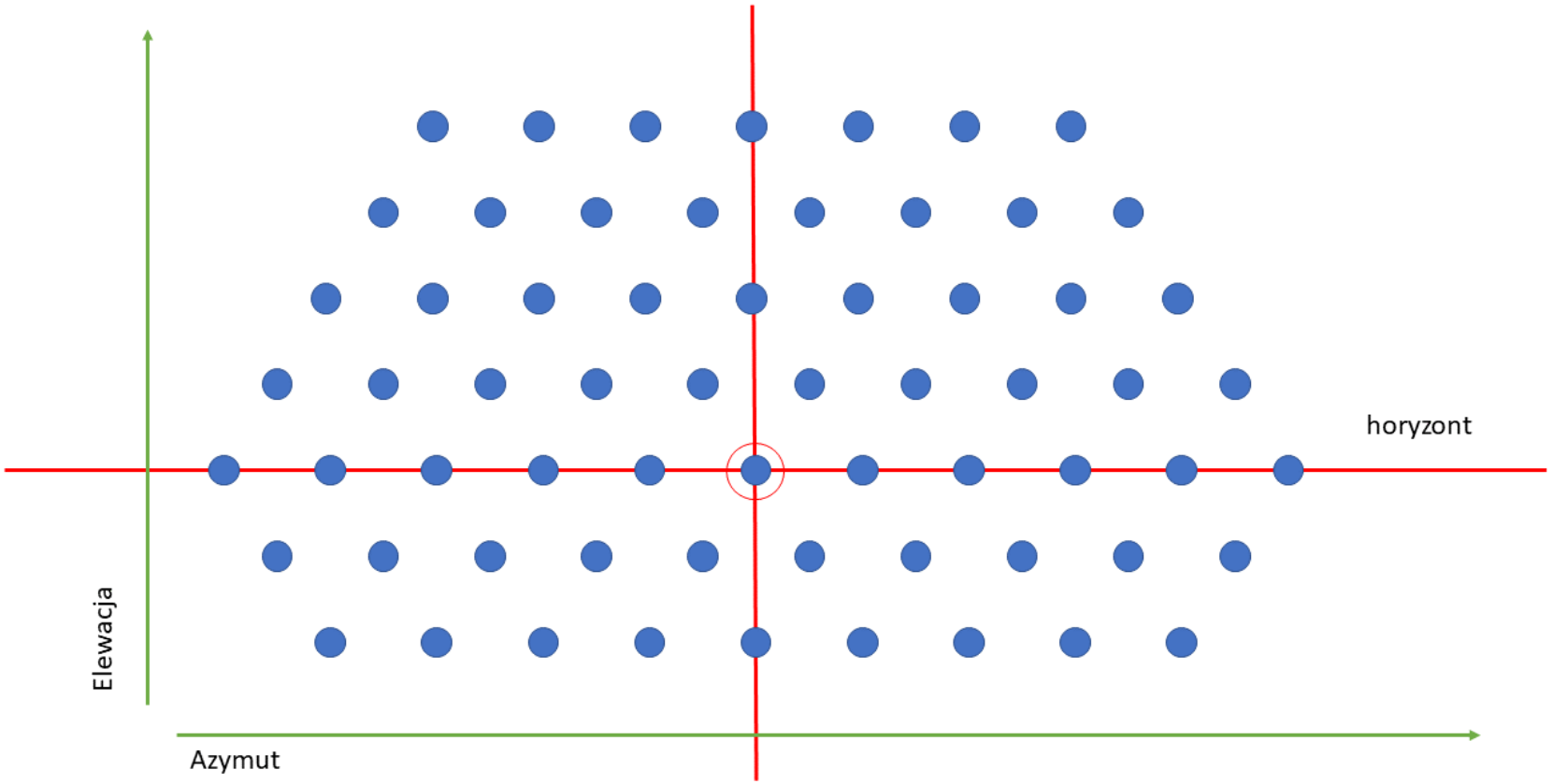
Wykład 7

19 listopada 2020

Treść dzisiejszego wykładu

- przyklad
- biblioteka standardowa
- kontenery
- iteratory
- alokatory
- przypomnienie

przykład



biblioteka standardowa

- STL - ang. *Standard Template Library*
- zawiera wiele użytecznych modułów w tym wzorce klas (szablony) najpopularniejszych komponentów programistycznych
- umożliwia skupienie się na rozwiązaniu danego problemu a nie powtarzaniu implementacji tych samych wzorców
- jest częścią języka C++, należy do standardu
- wciąż rozwijana, wraz z rozwojem języka C++
- wszystkie elementy z biblioteki standardowej znajdują się w przestrzeni nazw `std::`
- obszerna dokumentacja na stronie `https://en.cppreference.com/w/`, `https://pl.cppreference.com/w/Strona_główna`

biblioteka standardowa

Biblioteka standardowa udostępnia cały szereg funkcjonalności:

- biblioteka narzędzi (ang. *general utilities library*)
- łańcuchy znaków (ang. *strings library*)
- kontenery (ang. *containers library*)
- algorytmy (ang. *algorithms library*)
- iteratory (ang. *iterators library*)
- biblioteka numeryczna (ang. *numerics library*)
- operacje wejścia / wyjścia (ang. *input/output library*)
- wyrażenia regularne (ang. *regular expressions library*)
- wsparcie dla wielowątkowości (ang. *thread support library*)
- system plików (ang. *filesystem library*)

przegląd kontenerów z biblioteki standardowej - std::vector

std::vector

- definicja w nagłówku `<vector>`
- odpowiada tablicy obiektów
- nie ma określonego rozmiaru / potrafi dynamicznie zmieniać rozmiar przydzielonej pamięci
- obiekty przechowywane są w ciągłym obszarze pamięci
- możliwe dodawanie lub usuwanie elementów na końcu kolekcji
- możliwy dostęp do elementu o *n*-tym indeksie
- w przypadku próby dobrania się do nieistniejącego elementu może zostać rzucony wyjątek `std::out_of_range`

```
struct RadianAngle
{
    float value = 0;

    RadianAngle() = default;
    RadianAngle(float deg)
        : value(deg)
    {
    }
};
```

std::vector

- konstrukcja

```
std::vector<int> tab_1;                          // rozmiar tablicy - 0;
std::vector<int> tab_2 = { 1, 2, 3 };    // rozmiar tablicy - 3;
std::vector<RadianAngle> tab_3 = { {1}, {2}, {3} };
```

- przypisanie rozmiaru tablicy

```
tab_3.reserve(10);   // alokuje pamiec dla 10 obiektow typu RadianAngle
                     // aby uniknac ciaglego przepisywania pamieci
                     // przy dodawaniu nowych obiektow
```

- dodawanie elementu na końcu kolekcji

```
RadianAngle deg(4);
tab_3.push_back(deg);                  // kopiuje obiekt do tablicy
tab_3.push_back( RadianAngle(5) );  // konstruuje obiekt i
                                       // przenosi go do tablicy
tab_3.emplace_back(6);                 // tworzy obiekt w tablicy wywolujac
                                       // konstruktor klasy RadianAngle
```

std::vector

- dostęp do elementu o zadanym indeksie

```
std::cout << tab_3[5].value;      // nie sprawdza zakresu
std::cout << tab_3.at(5).value; // sprawdza zakres, moze zglosic wyjatek
std::cout << tab_3.front().value;    // pierwszy element w kolekcji
std::cout << tab_3.back().value;     // ostatni element w kolekcji
```

- aktualny rozmiar tablicy

```
std::cout << tab_3.size();        // zwraca ilosc elementow w tablicy
std::cout << tab_3.capacity();   // zwraca ilosc elementow ktore moga
                                 // byc przechowywane w tablicy
std::cout << tab_3.empty();       // sprawdza czy tablica nie jest pusta
```

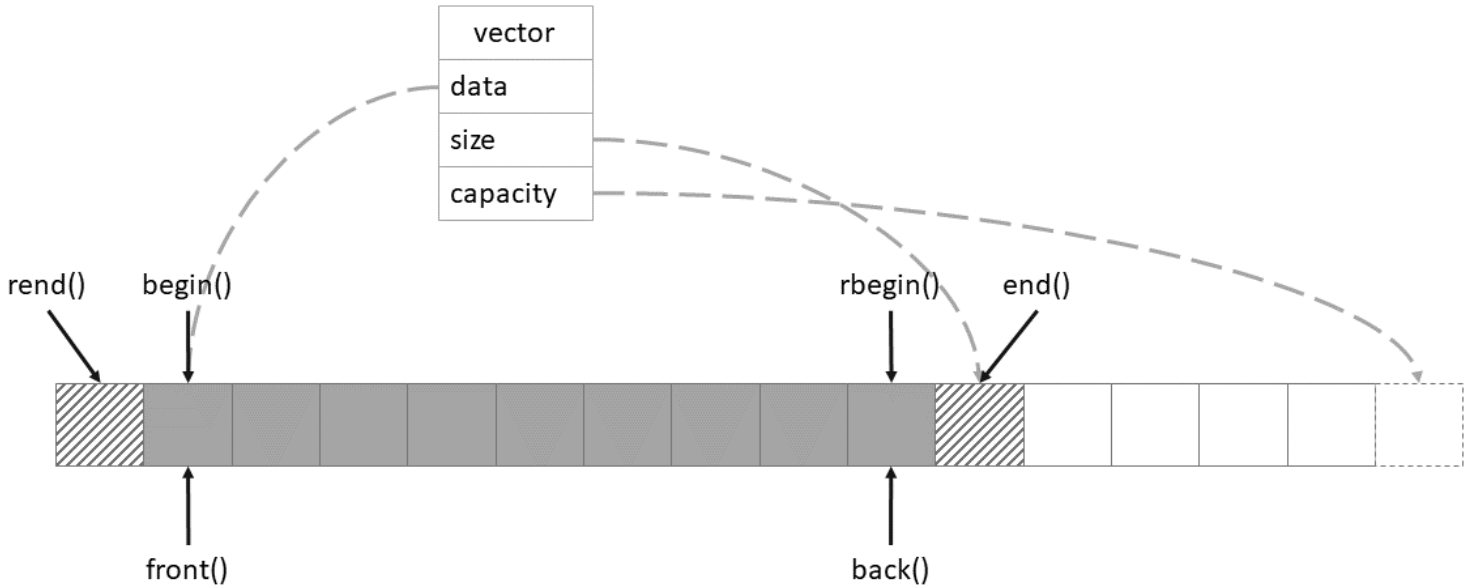
- zmiana rozmiaru tablicy

```
tab_3.resize(2);
tab_3.resize(10, RadianAngle(10));  // w puste miejsce wstawia
                                    // kopie obiektu przekazanego jako
                                    // drugi argument
```

- usuwanie elementow tablicy

```
tab_3.pop_back();    // usuniecie ostatniego, ilosc elementow - 9
tab_3.clear();       // usuniecie wszystkich, ilosc elementow - 0
```

przegląd kontenerów z biblioteki standardowej - std::vector



container_type<T>::iterator

- jest klasą pomocniczą implementowaną w kontenerach w celu usprawnienia oraz ujednolicenia dostępu do obiektów w nim przechowywanych (w kontenerze)
- jest niejako wskaźnikiem na obiekt w kontenerze danego typu
- w przeciwieństwie do wskaźników kontroluje przekroczenie zakresu
- implementuje semantykę wskaźników
- większość typów kontenerów udostępnia metody pozwalające na uzyskanie iteratora wskazującego na początek lub koniec kolekcji:

```
iterator begin() noexcept;
iterator end() noexcept;
reverse_iterator rbegin() noexcept;
reverse_iterator rend() noexcept;
```

- dostępne są również odpowiednie metody z kwalifikatorem `const` zwracające iterator niepozwalający na zmianę obiektu na który wskazuje

container_type<T>::iterator

W zależności od typu kontenera jego iteratory udostępniają różne funkcjonalności:

- *Input Iterator* umożliwia odczyt obiektu, na który wskazuje w kolekcji oraz inkrementację
- *Forward Iterator* umożliwia dodatkowo porównywanie iteratorów
- *Bidirectional Iterator* umożliwia dodatkowo dekrementację
- *Random Access Iterator* umożliwia dodatkowo dostęp do dowolnego elementu w kolekcji
- *Output Iterator* umożliwia zapis do obiektu, na który wskazuje w kolekcji oraz inkrementację
- *Contiguous Iterator* gwarantuje, że obiekty w kolekcji znajdują się w spójnym obszarze pamięci

std::vector<T>::iterator

Iteratory typu `vector<T>::iterator` reprezentują wszystkie 6 rodzajów iteratorów i posiadają następujące metody:

```
using pointer = T*;
using reference = T&;
using difference_type = ...;
using iterator = ...;

reference operator*() const;
pointer operator->() const;
iterator& operator++();
iterator operator++(int);
iterator& operator--();
iterator operator--(int);
iterator& operator+=(const difference_type _Off);
iterator operator+(const difference_type _Off) const;
iterator& operator-=(const difference_type _Off);
iterator operator-(const difference_type _Off) const;
difference_type operator-(const iterator& _Right) const;
reference operator[](const difference_type _Off) const;
```

std::vector<T>::iterator

```
#include <iostream>
#include <vector>
/*...*/
int main(void){
    std::vector<RadianAngle> tab;

    for (int i = 0; i < 10; ++i)
        tab.emplace_back(i);

    // iterowanie w stylu C
    for (size_t i = 0; i < tab.size(); ++i)
        std::cout << tab[i].value;

    // iterowanie w stylu C++
    for ( std::vector<RadianAngle>::iterator it = tab.begin();
            it != tab.end();
           ++it)
        std::cout << (*it).value;

    // iterowanie w stylu C++11
    for (auto it = tab.begin(); it != tab.end(); ++it)
        std::cout << it->value;

    // iterowanie w kierunku przeciwnym
    for (auto it = tab.rbegin(); it != tab.rend(); ++it)
        std::cout << it->value;
}
```

pętla for

W nowym standardzie C++ istnieje czytelniejsza forma pętli for.

```
#include <iostream>
#include <vector>

int main(void)
{
    std::vector<RadianAngle> tab;

    for (int i = 0; i < 10; ++i)
        tab.emplace_back(i);

    // uwaga na zbedne kopiowanie obiektow
    for (auto it : tab)
        std::cout << it.value;

    // tutaj nie ma kopiowania obiektow
    for (auto& it : tab)
        std::cout << it.value;

    return 0;
}
```

przegląd kontenerów z biblioteki standardowej - std::list

std::list

- definicja w nagłówku `<list>`
- odpowiada liście podwójnie powiązanej
- nie ma określonego rozmiaru / potrafi dynamicznie zmieniać rozmiar przydzielonej pamięci
- obiekty **nie** są przechowywane w ciągłym obszarze pamięci
- możliwe dodawanie lub usuwanie elementów w dowolnym miejscu kolekcji
- losowy dostęp do kolekcji nie jest możliwy
- iteratory nie są unieważniane o ile obiekty na które wskazują nie zostaną usunięte z kontenera

std::list

- dodawanie elementu na końcu oraz początku kolekcji

```
std::list<int> list_1 = { 1, 2, 3 };     // rozmiar listy - 3;

list_1.push_back(4);
list_1.push_front(5);
list_1.emplace_back(6);
list_1.emplace_front(7); // 7 5 1 2 3 4 6
list_1.emplace(std::next(list_1.begin(),2), 21); // 7 5 21 1 2 3 4 6
```

- usuwanie elementów listy

```
list_1.pop_back();
list_1.pop_front();
```

- sortowanie elementów w kolekcji

```
list_1.sort();
```

- scalanie dwóch posortowanych list, usuwanie powtórzeń

```
auto list_2 = list_1;
list_1.merge({ 0, 11, 17 });
list_1.merge(list_2);
list_1.unique();
```

std::list

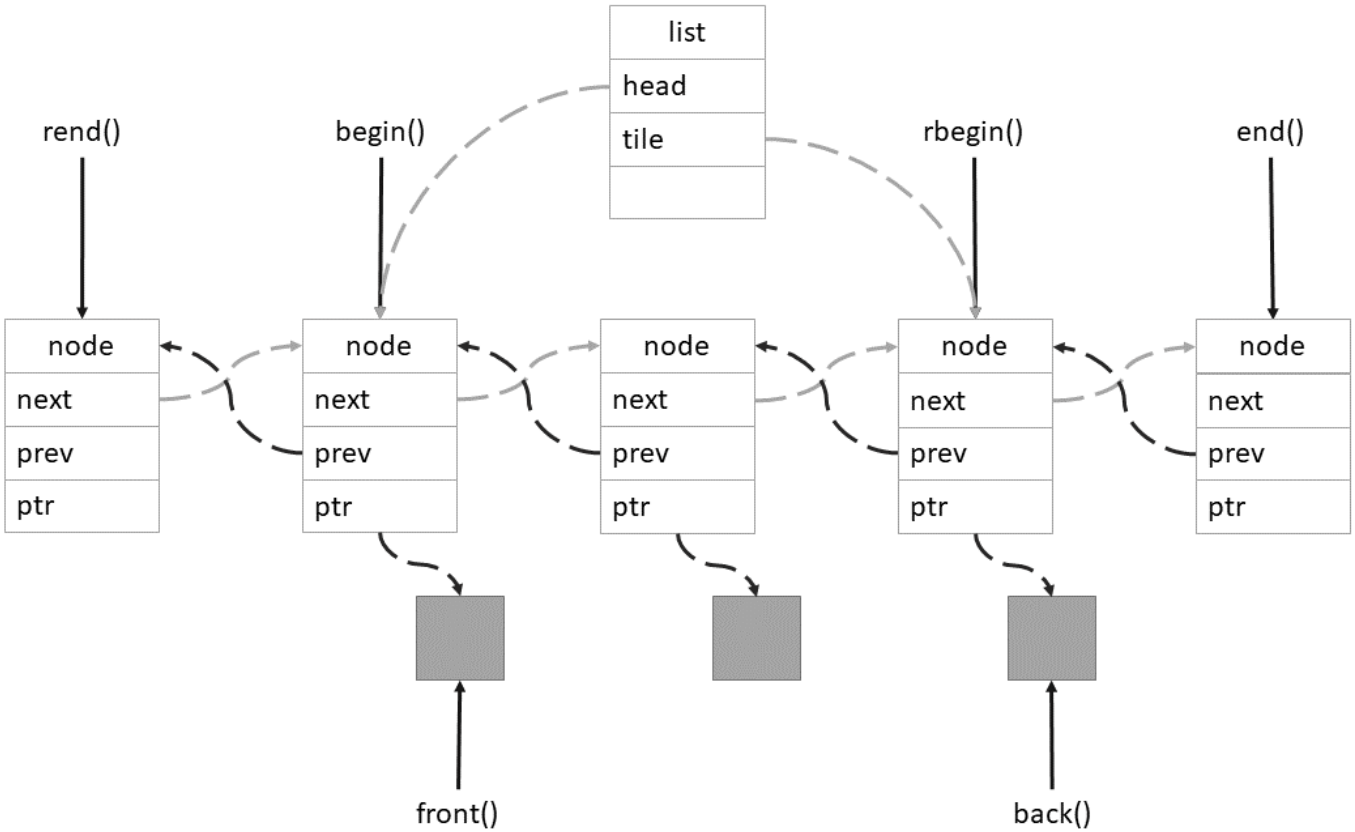
```
int main(void)
{
    std::list<int> list_1;
    for (int i = 0; i < 10; ++i)
    {
        list_1.push_back(i);
        list_1.push_front(i);
    }

    for (auto it = list_1.begin(); it != list_1.end(); ++it)
        list_1.erase(it++);

    for (auto& it : list_1)
        std::cout << it;

    return 0;
}
```

przegląd kontenerów z biblioteki standardowej - std::list



std::deque

- definicja w nagłówku `<deque>`
- kontener sekwencji indeksowanej (kolejka podwójnie zakończona)
- umożliwia wstawianie obiektów zarówno na początku jak i na końcu kolekcji
- wewnętrznie implementowany jako kolekcja wektorów o stałym rozmiarze
- w przeciwieństwie do `vector` elementy nie są przechowywane w ciągłym obszarze pamięci
- wymaga podwójnej dereferencji

przegląd kontenerów z biblioteki standardowej - std::set

std::set

- definicja w nagłówku `<set>`
- kolekcja unikalnych posortowanych elementów
- obiekty przechowywane w kontenerze muszą być porównywalne
- wstawianie elementów odbywa się zawsze w przeznaczonym miejscu zależnym od stanu kontenera
- wewnętrznie implementowany jako drzewo czerwono-czarne
- przeszukiwanie, usuwanie oraz wstawianie ma złożoność logarytmiczną zależną od ilości elementów w kolekcji

std::pair

- definicja w nagłówku cppkeyword<utility>
- struktura agregująca dwa pola

```
template<class T1, class T2 >
struct pair
{
    T1 first;
    T2 second;
};
```

- biblioteka standardowa udostępnia funkcje szablonową tworzącą powyższą strukturę

```
// ponizsze dwa zapisy stworza jednakowe obiekty
auto pair_1 = std::make_pair("pi", 3.14);
std::pair<const char*, double> pair_2 = { "pi", 3.14 };
```

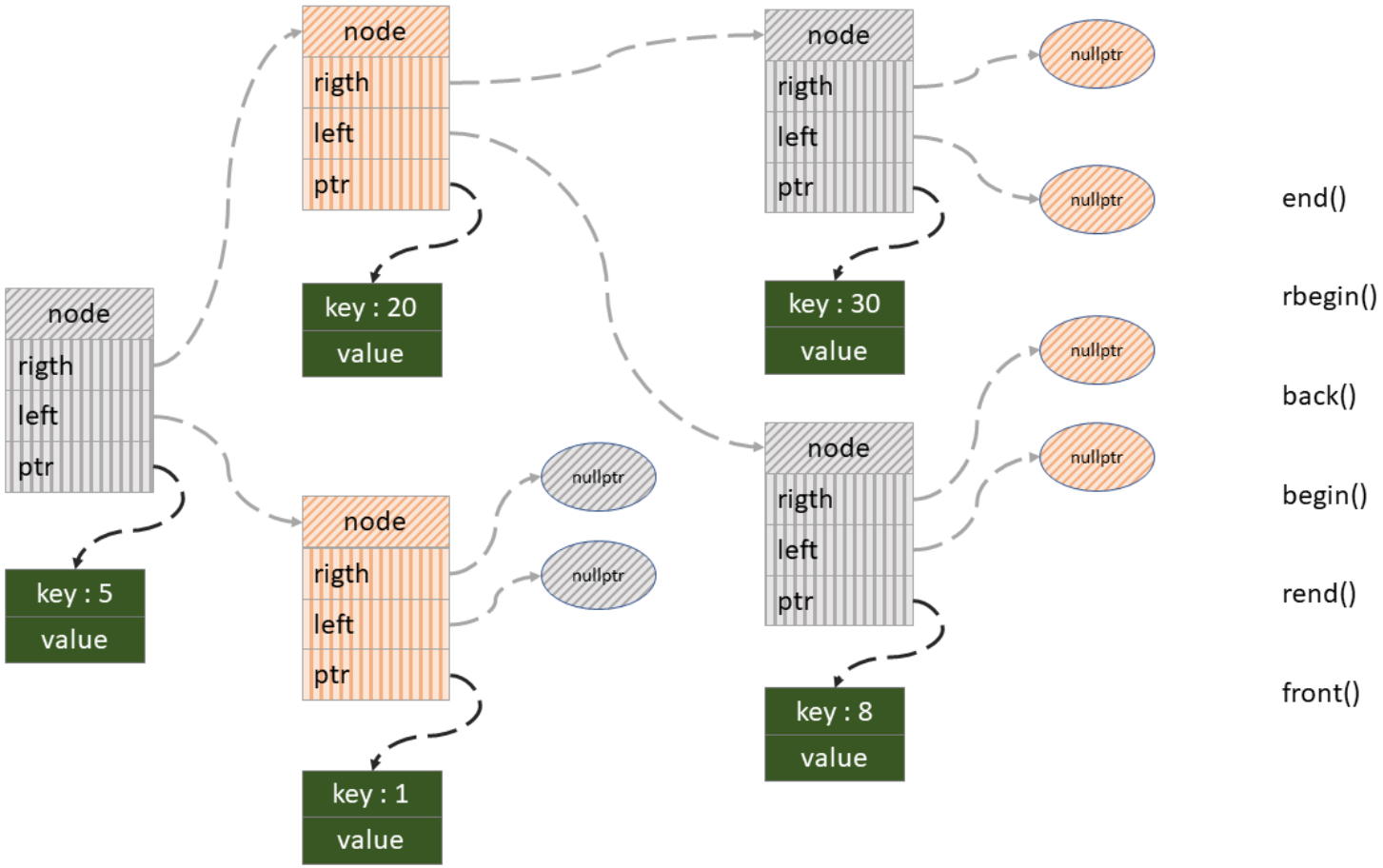
std::map

- definicja w nagłówku `<map>`
- kolekcja obiektów typu `std::pair`, gdzie pierwszy element pary stanowi klucz, drugi zaś przechowywany obiekt
- kolekcja zapewnia, że wartość klucza jest unikalna w całym kontenerze
- obiekty reprezentujące klucz muszą być porównywalne
- wstawianie elementów odbywa się zawsze w przeznaczonym miejscu zależnym od stanu kontenera
- wewnętrznie implementowany jako drzewo czerwono-czarne
- przeszukiwanie, usuwanie oraz wstawianie ma złożoność logarytmiczną zależną od ilości elementów w kolekcji

std::map

- definicja w nagłówku `<map>`
- kolekcja obiektów typu `std::pair`, gdzie pierwszy element pary stanowi klucz, drugi zaś przechowywany obiekt
- kolekcja zapewnia, że wartość klucza jest unikalna w całym kontenerze
- obiekty reprezentujące klucz muszą być porównywalne
- wstawianie elementów odbywa się zawsze w przeznaczonym miejscu zależnym od stanu kontenera
- wewnętrznie implementowany jako drzewo czerwono-czarne
- przeszukiwanie, usuwanie oraz wstawianie ma złożoność logarytmiczną zależną od ilości elementów w kolekcji

przegląd kontenerów z biblioteki standardowej - std::map



std::map

- konstrukcja

```
std::map<std::string, double> prefix_symbol = { {"T", 1e12},
                                                {"G", 1e9},
                                                {"M", 1e6},
                                                {"k", 1e3},
                                                {"h", 1e2},
                                                {"da",1e1} };
std::map<std::string, double> prefix_text;
```

- dodawanie elementu do kolekcji

```
prefix_text.insert(std::make_pair("tera",   1e12));
prefix_text.insert(std::make_pair("giga",   1e9));
prefix_text.insert(std::make_pair("mega",   1e6));
prefix_text.insert(std::make_pair("kilo",   1e3));
prefix_text.insert(std::make_pair("hecto",  1e2));
prefix_text.insert(std::make_pair("deca",   1e1));
```

- dostęp do elementu o zadanym kluczu

```
std::cout << "1 MHz = " << 1 * prefix_symbol["M"] << "Hz" << std::endl;
```

alokatory

- Alokator to szablon klasy zawierający strategię alokacji pamięci.
- Zastosowanie alokatorów pozwala oddzielić zarządzanie pamięcią przydzielaną na potrzeby przechowywania danych od logiki związanej z tymi danymi.
- Użytkownik może przekazać własny alokator pamięci jako argument konkretyzacji szablonu kontenera.
- Alokator pamięci zdefiniowany przez użytkownika musi zapewnić ten sam interfejs co alokator domyślny.
- Domyślny, wykorzystywany przez wszystkie kontenery z biblioteki standardowej, alokator pamięci to `std::allocator`.
- Obiekty typu `std::allocator<T>` nie posiadają stanu, nie zawierają informacji na przykład na temat przydzielonej pamięci.
- Pamięć przydzielona przez jeden obiekt typu `std::allocator<T>` może być zwrócona przez inny obiekt tego samego typu.

alokatory

```
template<    class T,
             class Allocator = allocator<T>>
class vector { /* ... */ };


template<    class T,
             class Allocator = std::allocator<T>>
class list{ /* ... */ };


template<    class T,
             class Allocator = std::allocator<T>>
class deque { /* ... */ };


template<    class Key,
             class Compare = std::less<Key>,
             class Allocator = std::allocator<Key>>
class set{ /* ... */ };


template<    class Key,
             class T,
             class Compare = std::less<Key>,
             class Allocator = std::allocator<std::pair<const Key, T>>>
class map { /* ... */ };
```

podsumowanie

Biblioteka standardowa języka C++ jest jego częścią i należy do standardu. Uzupełnia sam język logicznymi strukturami czyniąc go bardziej użytecznym.
STL (ang. *Standard Template Library*) jest częścią (rdzeniem) biblioteki standardowej należącą do języka C++ i jak sama nazwa wskazuje jest biblioteką wzorców klas. W jej skład wchodzą m.in.: kontenery, iteratowy i algorytmy.
Umożliwia programowanie na wyższym poziomie abstrakcji gdzie nie skupiamy się na implementacji mechanizmów, ale na rozwiązaniu problemu.
Wszystkie elementy biblioteki standardowej umieszczone są w przestrzeni nazw `std`.

podsumowanie

Biblioteka kontenerów to ogólny zbiór szablonów klas i algorytmów, które umożliwiają programistom łatwe wdrażanie typowych struktur danych, takich jak kolejki, listy i stosy. Istnieją trzy klasy kontenerów - kontenery sekwencji, kontenery asocjacyjne i nieuporządkowane kontenery asocjacyjne - z których każdy jest przeznaczony do obsługi innego zestawu operacji.
Kontener zarządza pamięcią przydzieloną dla jej elementów i zapewnia funkcje składowe dostępu do nich, bezpośrednio lub za pomocą iteratorów (obiekty o właściwościach podobnych do wskaźników).
Większość kontenerów ma co najmniej kilka wspólnych funkcji, które udostępnia. To, który pojemnik jest najlepszy dla danej aplikacji, zależy nie tylko od oferowanej funkcjonalności, ale także od wydajności w przypadku różnych obciążeń.

przypomnienie

```
template<typename T>
class Vector
{
public:
    Vector(size_t capacity);
    Vector(void);
    ~Vector(void);
    size_t size(void);
    size_t capacity(void);
    void push_back(const T& rhs);
    void pop_back(const T& rhs);
    T& operator[](size_t at);
    const T& operator[](size_t at) const;
    void reserve(size_t capacity);
    void resize(size_t capacity);
    void clear(void);
private:
    T* data_ = nullptr;
    size_t size_ = 0;
    size_t capacity_ = 0;
};
```

Parametrami szablonu mogą być też typy, które wykonują określone czynności na rzecz konkretyzowanego szablonu. Tego typu technikę nazywamy dodawaniem wytycznych bądź polityk (ang. *policy*).

```
template<typename T, typename checking_policy>
class Vector
{
public:
/* ... */
    T& operator[](size_t at);
/* ... */
};

template<typename T, typename checking_policy>
T& Vector<T,checking_policy>::operator[](size_t at);
{
    checking_policy::check(at, size_);
    return data_[at];
}
```

przypomnienie

Wytyczne są zwykłymi klasami (lub nawet klasami szablonowymi !), które udostępniają wymaganą funkcjonalność. Wszystkie wytyczne wchodzące w skład danej polityki muszą udostępniać wymaganą funkcjonalność. W przykładzie wymagamy aby taka klasa udostępniała statyczną metodę przyjmującą dwie wartości całkowitoliczbowe.

```
class no_check_policy
{
    static void check(size_t /*at*/, size_t /*size*/) {}
};

class check_range_policy
{
    static void check(size_t at, size_t size)
    {
        assert(at < size);
    }
};
```

Możliwe jest (co jest zalecane) dodanie domyślnej wytycznej, która będzie wykorzystana w przypadku, gdy żadna nie zostanie podana jako parametr szablonu.

```
template<
    typename T,
    typename checking_policy = check_range_policy
>
class Vector
{
/* ... */
};

int main(void)
{
    Vector<double> v1;
    Vector<double, check_range_policy> v2;
    Vector<double, no_check_policy> v3;

    return 0;
}
```