STL #1

KONTENERY SEKWENCYJNE

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

AGENDA

1. Przypomnienie STLa
2. Jak czytać dokumentację
3. `std::vector`
4. `std::list`
5. `std::forward_list`
6. `std::array`
7. `std::deque`

ZADANIA

Repo GH `coders-school/stl`

https://github.com/coders-school/stl/tree/master/module1

Zadania wykonywane podczas zajęć online nie wymagają ściągania repo. Pliki będą tworzone od zera.

KRÓTKIE PRZYPOMNIENIE

CO JUŻ WIEMY

- co zapamiętaliście z poprzednich zajęć?
- co sprawiło największą trudność?
- co najłatwiej było wam zrozumieć?

KONTENERY SEKWENCYJNE

STL

KRÓTKIE PRZYPOMNIENIE

CODERS SCHOOL

STL - STANDARD TEMPLATE LIBRARY

JAK CZYTAĆ DOKUMENTACJĘ

HTTPS://EN.CPPREFERENCE.COM

Q&A

std::vector<T>

TABLICA O DYNAMICZNYM ROZMIARZE

CODERS SCHOOL

CECHY `std::vector<T>`

- Cache-friendly, tzn. iterując po wektorze, zostanie on cały załadowany do pamięci podręcznej procesora, co przyspieszy odczytywanie danych.
- Typ `<T>` może być dowolny. Zarówno typ wbudowany jak `int`, `double`, jak i własny zdefiniowany przez nas typ.
- Elementy są ułożone obok siebie w pamięci, tak jak w zwykłej tablicy.
- Dodawanie nowego elementu, gdy zajdzie zależność `vec.size() > vec.capacity()` spowoduje alokację dodatkowego miejsca w pamięci. W krytycznej sytuacji, gdy przy obecnym obszarze nie ma już miejsca na dodatkowe dane, cały wektor zostanie przeniesiony w inne miejsce w pamięci.
- Usuwanie elementu z wektora jest szybkie, gdy usuwamy ostatni element, ale kosztowne, gdy usuwamy ze środka lub z początku.

OPERACJE NA `std::vector<T>`

- dodawanie elementu: `push_back`, `emplace_back()`, `insert()`
- modyfikowanie/dostęp do elementu: `at()`, `operator []`
- pierwszy/ostatni element: `back()`, `front()`
- rozmiar/czy kontener jest pusty: `size()`, `empty()`
- zarezerwowane miejsce: `capacity()`
- rezerwowanie miejsca w pamięci: `reserve()`
- wyczyszczenie nieużywanej pamięci z wektora: `shrink_to_fit()`
- iterator początku/końca: `begin()`, `end()`
- odwrócony (ang. reverse) iterator: `rbegin()`, `rend()`
- stały iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- wyczyszczenie kontenera: `clear()`
- przygotowanie elementu do usunięcia: `remove()` (nie jest metodą `std::vector<T>`)
- wymazanie elementów z pamięci: `erase()`
- podmiana całego kontenera: `swap()`

WSTAWIANIE ELEMENTÓW #1

`std::vector<T>::insert()`

```
iterator insert( const_iterator pos, const T& value );
```

W celu dodania elementu do wektora, możemy wykorzystać iterator:

```
std::vector<int> vec{1, 2, 3, 4};
auto it = vec.begin();
vec.insert(it, 20); // {20, 1, 2, 3, 4};
```

WSTAWIANIE ELEMENTÓW #2

std::vector<T>::insert()

```
iterator insert( const_iterator pos, size_type count, const T& value );
```

Możemy także określić ile elementów chcemy dodać:

```
std::vector<int> vec{1, 2, 3, 4};
auto it = vec.begin();
vec.insert(it, 5, 20); // {20, 20, 20, 20, 20 1, 2, 3, 4};
```

WSTAWIANIE ELEMENTÓW #3

std::vector<T>::insert()

```
template< class InputIt >
iterator insert( const_iterator pos, InputIt first, InputIt last );
```

Istnieje też możliwość wstawienia elementów z jednego kontenera do drugiego:

```
std::vector<int> vec{1, 2, 3, 4};
std::list<int> list{10, 20, 30, 40};
vec.insert(vec.begin(), list.begin(), list.end());
// vec = {10 20 30 40 1 2 3 4}
```

ITEROWANIE OD KOŃCA

std::vector<T>::rbegin(), std::vector<T>::rend()

```
std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8, 9};
for (auto it = vec.crbegin() ; it != vec.crend() ; ++it) {
    // cr = (r)everse iterator to (c)onst value
    std::cout << *it << ' ';
}
```

Output: 9 8 7 6 5 4 3 2 1

```
std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8, 9};
for (auto it = vec.rbegin() ; it != vec.rend() ; ++it) {
    *it *= 2;
}
for (auto it = vec.crbegin() ; it != vec.crend() ; ++it) {
    std::cout << *it << ' ';
}
```

Output: 18 16 14 12 10 8 6 4 2

(PRAWIE) USUWANIE 😉

`std::remove()` Z NAGŁÓWKA `<algorithm>`

```
template< class ForwardIt, class T >
ForwardIt remove( ForwardIt first, ForwardIt last, const T& value );
```

Ponieważ najszybciej usuwane są elementy z końca wektora, biblioteka STL umożliwia nam przygotowanie `std::vector<T>` do usunięcia elementów poprzez przeniesienie tych poprawnych na początek kontenera. W wyniku tego część wartości do usunięcia jest nadpisywana wartościami z końca wektora, które nie powinny zostać usunięte. Dlatego na końcu wektora pozostają "śmieci", które należy wymazać (ang. erase) z pamięci.

```
std::vector<int> vec{1, 4, 2, 4, 3, 4, 5};
std::remove(vec.begin(), vec.end(), 4);
// for example: vec {1 2 3 5 3 4 5}
```

`std::remove()` zwróci nam iterator, który wskaże początek danych przeznaczonych do usunięcia.

USUWANIE

std::vector<T>::erase()

```
template< class T, class Alloc, class U >
constexpr typename std::vector<T,Alloc>::size_type
    erase(std::vector<T,Alloc>& c, const U& value);
```

Dzięki funkcji erase, możemy teraz usunąć niepotrzebne dane z kontenera:

```
std::vector<int> vec{1, 4, 2, 4, 3, 4, 5};
auto it = std::remove(vec.begin(), vec.end(), 4);
vec.erase(it, vec.end());
// vec {1, 2, 3, 5}
```

Możemy też zapisać to wszystko w jednej linii (Erase-Remove Idiom)

```
vec.erase(std::remove(vec.begin(), vec.end(), 4), vec.end());
```

ZADANIE 1

- Otwórz dokumentację wektora na cppreference.com
- Stwórz nowy plik cpp i napisz funkcję `main()`
- Stwórz wektor o wartościach { 1, 2, 4, 5, 6 }
- Usuń pierwszą wartość
- Dodaj wartość 5 na końcu wektora
- Dodaj wartość 12 na początku wektora metodą `emplace`
- Wypisz rozmiar wektora i maksymalny możliwy rozmiar
- Wypisz zawartość wektora
- Wyczyść wektor
- Wypisz rozmiar wektora

ZADANIE 2

- Otwórz dokumentację wektora na cppreference.com
- Stwórz nowy plik cpp i napisz funkcję `main()`
- Stwórz pusty wektor
- Wypisz rozmiar i pojemność wektora
- Zmień rozmiar wektora na 10 i wypełnij go wartościami 5
- Wypisz rozmiar i pojemność wektora
- Zarezerwuj pamięć na 20 elementów
- Wypisz rozmiar i pojemność wektora
- Zredukuj pojemność wektora metodą `shrink_to_fit()`
- Wypisz rozmiar i pojemność wektora

Q&A

std::array<T, N>

TABLICA O STAŁYM ROZMIARZE

CODERS SCHOOL

CECHY `std::array<T, N>`

- Cache-friendly, tzn. iterując po `std::array<T, N>`, zostanie on cały załadowany do pamięci podręcznej procesora, co przyspieszy odczytywanie danych
- Typ `<T>` może być dowolny. Zarówno typ wbudowany jak int, double, jak i własny zdefiniowany przez nas typ
- Typ `<N>` oznacza rozmiar tablicy, który musi być znany już w czasie kompilacji
- Elementy są ułożone obok siebie w pamięci, tak jak w zwykłej tablicy
- `std::array` jest najprymitywniejszym `wrapperem` na zwykłe tablice, używanie jej jest praktycznie tak samo wydajne
- Nie możemy dodać nowego elementu, możemy jedynie dokonać modyfikacji już istniejących pól
- Odczyt oraz modyfikacja elementów jest bardzo szybka
- Tak jak nie możemy dodać elementów, tak też nie możemy ich usunąć (rozmiar się nie zmienia)

OPERACJE NA `std::array<T, N>`

- dodawanie elementu: nie da się
- modyfikowanie/dostęp do elementu: `at()`, `operator[]`
- pierwszy/ostatni element: `back()`, `front()`
- rozmiar/czy kontener jest pusty: `size()`, `empty()`
- iterator początku/końca: `begin()`, `end()`
- odwrócony (ang. reverse) iterator: `rbegin()`, `rend()`
- stały iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- wyczyszczenie kontenera: nie da się, jednak mamy metodę `fill`, którą możemy np. wyzerować wszystkie elementy
- przygotowanie elementu do usunięcia: nie da się
- wymazanie elementów z pamięci: nie da się
- podmiana całego kontenera: `swap()`

PRZEKAZANIE `std::array<T, N>` DO FUNKCJI

Ponieważ `std::array<T, N>` ma 2 parametry szablonu, niektórzy mogą mieć problem przy przekazywaniu jej do funkcji, gdyż pisząc `std::array<T>` zapominając o rozmiarze tablicy.

```cpp
void print(const std::array<int, 10>& arr) {
    for (auto el : arr) {
        std::cout << el << ' ';
    }
    std::cout << "\n";
}

int main() {
    std::array<int, 10> arr{1, 2, 3, 4, 5, 6, 7, 8, 9 , 10};
    print(arr);

    return 0;
}
```

ZADANIE

- Znajdź dokumentację `std::array` na cppreference.com
- Stwórz nowy plik cpp i napisz funkcję `main()`
- Stwórz `std::array` o rozmiarze 10
- Wypełnij ją wartościami 5
- Do czwartego elementu przypisz wartość 3
- Stwórz inną tablicę o tym samym rozmiarze
- Podmień tablice
- Wypisz obie tablice, każdą w osobnej linii

Q&A

std::list<T>

LISTA DWUKIERUNKOWA

CODERS SCHOOL

CECHY `std::list<T>`

- Elementy porozrzucane po pamięci
- Każdy element (węzeł, ang. node) posiada wskaźnik na poprzedni i następny element
- Typ `<T>` może być dowolny. Zarówno typ wbudowany jak `int`, `double`, jak i własny zdefiniowany przez nas typ
- Nie jest cache-friendly
- Dodawanie nowego elementu jest proste. Program zaalokuje potrzebną pamięć dla węzła i przekaże sąsiednim węzłom (o ile istnieją) informacje o swoim położeniu
- Usuwanie elementu jest szybkie, program zwalnia pamięć zaalokowaną dla danego węzła oraz informuje o tym sąsiednie węzły, aby mogły zmienić swoje wskaźniki
- Wyszukiwanie węzła (np. do usunięcia lub wstawienia za nim nowego elementu) jest już kosztowne, gdyż musimy się przeiterować kolejno przez wszystkie węzły, aż odnajdziemy poszukiwany (nawet, jeżeli dokładnie wiemy, że jest on np. 40-tym elementem listy)

OPERACJE NA `std::list<T>`

- dodawanie elementu: `push_back()`, `emplace_back()`, `push_front()`, `emplace_front()`, `insert()`
- modyfikowanie/dostęp do elementu: należy samodzielnie odnaleźć element
- pierwszy/ostatni element: `back()`, `front()`
- rozmiar/czy kontener jest pusty: `size()`, `empty()`
- iterator początku/końca: `begin()`, `end()`
- odwrócony (ang. reverse) iterator: `rbegin()`, `rend()`
- stały iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- wyczyszczenie kontenera: `clear()`
- posortowanie listy: `sort()`
- odwrócenie listy: `reverse()`
- usunięcie duplikatów: `unique()`
- usunięcie elementów z listy: `remove()`
- wymazanie elementów z pamięci: `erase()`
- podmiana całego kontenera: `swap()`

std::list<T>::remove() && std::list<T>::erase()

Ponieważ lista zawiera swoją metodę `remove()`, nie musimy już korzystać z `erase()`.

```
std::list<int> list{1, 4, 2, 4, 3, 4, 5};
list.remove(4);
// list {1, 2, 3, 5}
```

`erase()` używamy podobnie jak dla `std::vector<T>`

```
std::list<int> list{1, 2, 3, 4, 5, 6, 7, 8};
auto it = list.begin();
std::advance(it, 3); // like on pointer ptr += 3
list.erase(list.begin(), it);
// list {4, 5, 6, 7, 8}
```

`std::advance()` służy do inkrementowania iteratorów. W naszym przypadku przesuwamy się o 3 elementy do przodu.

ZADANIE 4

- Znajdź dokumentację `std::list` na cppreference.com
- Stwórz nowy plik cpp i napisz funkcję `main()`
- Stwórz listę zawierającą elementy od 0 do 5
- Wyświetl listę
- Usuń trzeci element z listy
- Dodaj na początek i koniec listy wartość 10
- Wyświetl listę
- Dodaj na czwartej pozycji liczbę 20
- Przepisz listę do `std::array`
- Wyświetl `std::array`

Q&A

std::forward_list<T>

LISTA JEDNOKIERUNKOWA

CODERS SCHOOL

CECHY `std::forward_list<T>`

- Elementy porozrzucane po pamięci
- Każdy element (węzeł -> ang. node) posiada wskaźnik na następny element
- Typ `<T>` może być dowolny. Zarówno typ wbudowany jak `int`, `double`, jak i własny zdefiniowany przez nas typ.
- Nie jest cache-friendly
- Dodawanie nowego elementu jest proste. Program zaalokuje potrzebną pamięć dla węzła i przekaże poprzedniemu węzłowi (o ile istnieje) informacje o swoim położeniu.
- Usuwanie elementu jest szybkie, program zwalnia pamięć zaalokowaną dla danego węzła oraz informuje o tym poprzedni węzeł, aby mógł zmienić swój wskaźnik.
- Wyszukiwanie węzła (np. do usunięcia lub wstawienia za nim nowego elementu) jest już kosztowne, gdyż musimy się przeiterować kolejno przez wszystkie węzły, aż odnajdziemy poszukiwany (nawet, jeżeli dokładnie wiemy, że jest on np. 40-tym elementem listy)

OPERACJE NA `std::forward_list<T>`

- dodawanie elementu: `push_front()`, `emplace_front()`, `insert_after()`, `emplace_after()`
- modyfikowanie/dostęp do elementu: należy samodzielnie odnaleźć element
- pierwszy/ostatni element: `front()`
- rozmiar/czy kontener jest pusty: nie mamy `size()`, dostępny jest tylko `empty()`
- iterator początku/końca: `begin()`, `end()`
- iterator wskazujący element przed `begin()`: `before_begin()`
- stały iterator: `cbegin()`, `cend()`, `cbefore_begin()`
- wyczyszczenie kontenera: `clear()`
- posortowanie listy: `sort()`
- odwrócenie listy: `reverse()`
- usunięcie duplikatów: `unique()`
- usunięcie elementów z listy: `remove()`
- wymazanie elementów z pamięci: `erase_after()`
- wymazanie elementów z pamięci używając `<algorithm>`: `erase()`
- podmiana całego kontenera: `swap()`

std::forward_list<T>::insert_after() && std::forward_list<T>::before_begin()

Lista jednokierunkowa (ang. singly linked list) umożliwia nam wstawianie elementu za konkretnym węzłem. Jeżeli chcemy wstawić wartość na początku listy używając metody `insert_after`, musimy podać jej specjalny iterator `before_begin`, który wskazuje element przed pierwszym węzłem listy. W ten sposób, metoda `insert_after()` wstawi pożądaną przez na wartość dokładnie jako pierwszy element listy.

```
std::forward_list<int> list {1, 2, 3, 4, 5, 6};
list.insert_after(list.begin(), 10);
print(list);
list.insert_after(list.before_begin(), 0);
print(list);
```

Output:

```
1 10 2 3 4 5 6
0 1 10 2 3 4 5 6
```

`std::forward_list<T>::remove()`

Ponieważ lista zawiera swoją metodę `remove()`, nie musimy już korzystać z `erase()`.

```
std::forward_list<int> list {1, 4, 2, 4, 3, 4, 5};
list.remove(4);
// {1, 2, 3, 5}
```

std::forward_list<T>::erase_after()

Erase_after służy do usunięcia węzłów od miejsca za elementem wskazanym przez pierwszy iterator do momentu wskazanego przez drugi iterator (bez elementu wskazywanego przez niego).

```
std::forward_list<int> list {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto it = list.begin();
std::advance(it, 4);
std::cout << "it: " << *it << std::endl;
auto it2 = list.begin();
std::advance(it2, 7);
std::cout << "it2: " << *it2 << std::endl;
list.erase_after(it, it2);
print(list);
```

Output:

```
it: 5
it2: 8
1 2 3 4 5 8 9 10
```

ZADANIE 5

- Znajdź dokumentację `std::forward_list` na cppreference.com
- Skorzystaj z kodu z zadania z `std::list`
- Stwórz listę jednokierunkową zawierającą elementy od 0 do 5
- Wyświetl listę
- Usuń 3 element z listy
- Dodaj na początek i koniec listy wartość 10
- Wyświetl listę
- Dodaj na czwartej pozycji liczbę 20
- Wyświetl listę

Q&A

std::deque<T>

KOLEJKA DWUSTRONNA

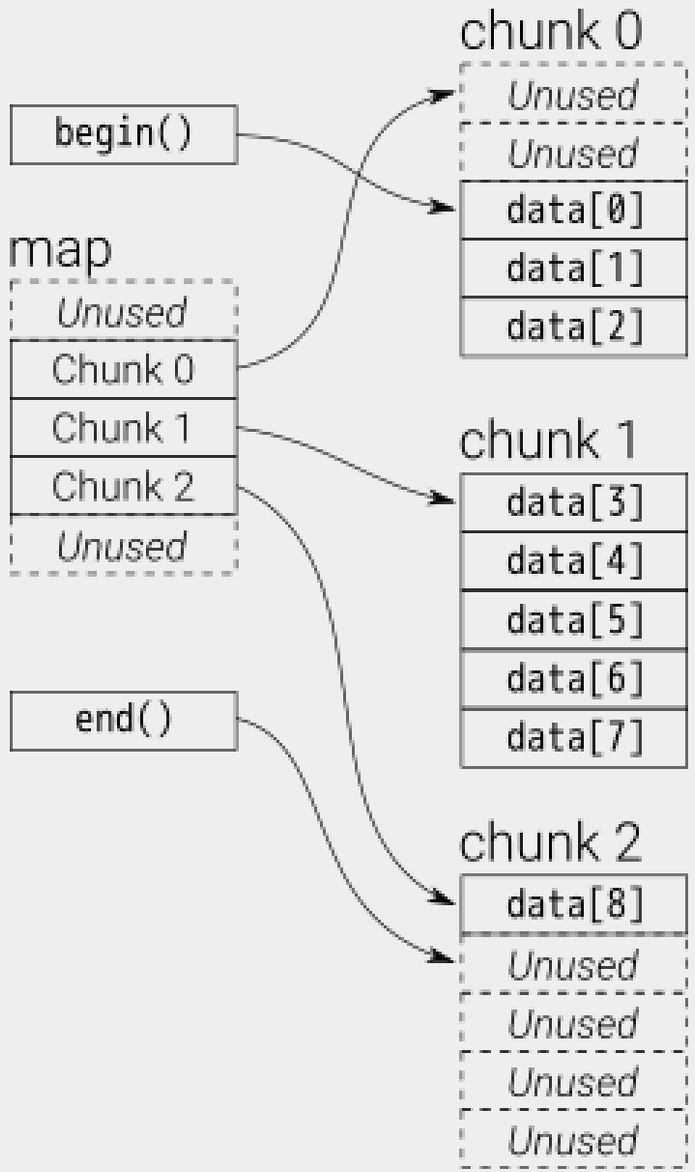
DEQUE = DOUBLE ENDED QUEUE

CODERS SCHOOL

CECHY `std::deque<T>`

- Hybryda listy oraz wektora
- `deque` dzieli się na kawałki (ang. chunk), które są tablicami porozrzucanymi po pamięci
- O wielkości takiego kawałka decyduje kompilator (nie ma jednej reguły)
- Dodatkowo deque wyposażony jest w jeszcze jeden wektor, który przechowuje wskaźniki wskazujące początek każdego `chunka` w pamięci.
- W ten sposób zyskujemy 2 rzeczy:
  - Dodawanie nowego elementu, jest szybsze, gdyż alokujemy zawsze pamięć dla całego `chunka` i nie będziemy przenosić elementów jak w `std::vector<T>`, gdy zabraknie nam miejsca na alokacje dodatkowej pamięci
  - Dane załadowane z jednego `chunka` są cache-friendly

STRUKTURA `std::deque<T>`



CECHY `std::deque<T>` CD.

- Częściowo cache-friendly, czyli poszczególne `chunki` znajdą się w pamięci podręcznej procesora
- Typ `<T>` może być dowolny. Zarówno typ wbudowany jak `int`, `double`, jak i własny zdefiniowany przez nas typ
- Każdy `chunk` jest reprezentowany w pamięci jak tablica, natomiast same `chunki` nie sąsiadują ze sobą i są porozrzucane jak węzły listy
- Dodanie nowego elementu jest szybkie
  - Jeżeli dany `chunk` ma jeszcze miejsce to dopisujemy go na koniec
  - Jeżeli nie, to alokowany jest nowy chunk i tam wpisywany jest nowy element
- Usuwanie z początku i końca jest szybkie, bo powoduje jedynie przesunięcie iteratorów `begin()` lub `end()`
- Usuwanie elementów ze środka jest kosztowne
- Odczyt i modyfikacja jest szybka
  - Znamy rozmiar `chunka`, więc wiemy dokładnie, z którego pola w naszym wektorze pomocniczym powinniśmy odczytać adres `chunka`
  - Wiemy także, z którego pola odczytać daną, gdyż `chunk` jest ułożony jak tablica.

DOSTĘP DO ELEMENTU

Matematycznie ujmując: jeżeli `chunk` ma 16 elementów a my chcemy dostać się do 100 elementu to:

- `x = 100 / 16 -> x = 6` (ucinamy część po przecinku)
- `y = 100 % 16 -> y = 4`

Zatem wiemy, że jest to 4-ty element w 6-tym `chunku`

Ta wiedza jest zupełnie niepotrzebna przy użytkowaniu `std::deque`. Kontener zajmuje się tym automatycznie.

OPERACJE NA `std::deque<T>`

- dodawanie elementu: `push_back()`, `emplace_back()`, `push_front()`, `emplace_front()`, `insert()`
- modyfikowanie/dostęp do elementu: `at()`, `operator[]`
- pierwszy/ostatni element: `back()`, `front()`
- rozmiar/czy kontener jest pusty: `size()`, `empty()`
- wyczyszczenie nieużywanej pamięci: `shrink_to_fit()`,
- iterator początku/końca: `begin()`, `end()`
- odwrócony (ang. reverse) iterator: `rbegin()`, `rend()`
- stały iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- wyczyszczenie kontenera: `clear()`
- przygotowanie elementu do usunięcia: `remove()` (nie jest metodą `std::deque`),
- wymazanie elementów z pamięci: `erase()`
- podmiana całego kontenera: `swap()`

PRZYKŁAD UŻYCIA

```cpp
#include <iostream>
#include <deque>

int main() {
    // Create a deque containing integers
    std::deque<int> d = {7, 5, 16, 8};

    // Add an integer to the beginning and end of the deque
    d.push_front(13);
    d.push_back(25);

    // Iterate and print values of the deque
    for(int n : d) {
        std::cout << n << ' ';
    }
    std::cout << '\n';
}
```

Output:

```
13 7 5 16 8 25
```

ZADANIE

- Znajdź dokumentację `std::deque` na cppreference.com
- Stwórz nowy plik cpp i napisz funkcję `main()`
- Stwórz pusty `deque`
- Dodaj do niego 5 dowolnych wartości
- Wyświetl deque
- Usuń 2-gi i 4-ty element
- Dodaj na początek i koniec wartość 30
- Wyświetl deque
- Dodaj na 4 pozycji liczbę 20
- Wyświetl deque

Q&A

STL #1

KONTENERY SEKWENCYJNE

PODSUMOWANIE

CODERS SCHOOL

CO PAMIĘTASZ Z DZISIAJ?

NAPISZ NA CZACIE JAK NAJWIĘCEJ HASEŁ

1. Przypomnienie STLa
2. Jak czytać dokumentację
3. `std::vector<T>`
4. `std::list<T>`
5. `std::forward_list<T>`
6. `std::array<T, N>`
7. `std::deque<T>`

PRACA DOMOWA

POST-WORK

Ta praca domowa może już stanowić pewne wyzwanie. Działaj dużo z dokumentacją cppreference.

- Poczytaj o formacie grafiki PGM - Wiki ENG. Ta wiedza może się przydać w zadaniu 3 i 4.
- Zadanie 1 - `removeVowels()` (5 punktów)
- Zadanie 2 - `lengthSort()` (6 punktów)
- Zadanie 3a - `compressGrayscale()` (7 punktów)
- Zadanie 3b - `decompressGrayscale()` (7 punktów)

BONUSY

- 2 punkty za każde zadanie dostarczone przed 14.06.2020 (niedziela) do 23:59
- 3 punkty za pracę w grupie dla każdej osoby z grupy. Zalecamy grupy 3 osobowe.

ZADANIA W REPO

PRE-WORK

- Znajdź na cppreference.com opisy algorytmów i zapoznaj się z nimi. Popatrz też na przykłady użycia.
- Poczytaj o złożoności obliczeniowej np. na Samouczku programisty
- Obejrzyj i zapamiętaj jaka jest złożoność operacji na poszczególnych kontenerach STL

ZADANIE 1 - `removeVowels()`

Napisz funkcję `removeVowels()`, która przyjmie `std::vector<std::string>` oraz usunie wszystkie samogłoski z tych wyrażeń.

- Input: `{"abcde", "aabbbccabc", "qwerty"}`
- Output: `{"bcd", "bbccbc", "qwrt"}`

ZADANIE 2 - `lengthSort()`

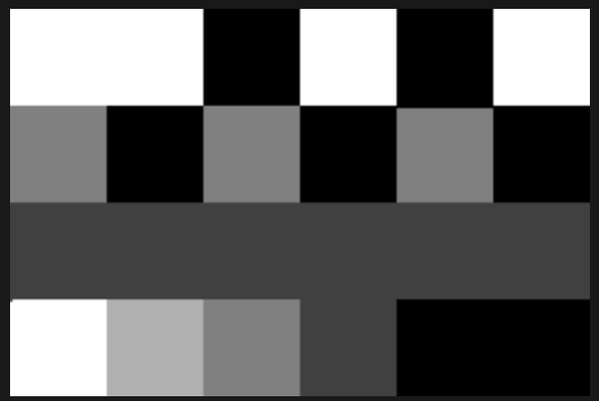
Napisz funkcję `lengthSort()`.

Ma ona przyjąć `std::forward_list<std::string>` i zwrócić `std::deque<std::string>` z posortowanymi słowami od najkrótszego do najdłuższego. Jeżeli dwa lub więcej słów ma tyle samo znaków posortuj je leksykograficznie.

- Input: `std::forward_list<std::string>{"Three", "One", "Four", "Two"}`
- Output: `std::deque<std::string>{"One", "Two", "Four", "Three"}`

ZADANIE 3A - `compressGrayscale()`

Zadaniem będzie kompresja obrazka w odcieniach szarości o wymiarach 240x160 pikseli. Każdy piksel może mieć wartość od 0 (kolor czarny) do 255 (kolor biały). Im większa wartość tym jaśniejszy odcień piksel reprezentuje. Przykład małego obrazka o rozmiarach 6x4 piksele:



```
255 255 0   255 0   255   // 0xFF 0xFF 0x00 0xFF 0x00 0xFF
128 0   128 0   128 0     // 0x80 0x00 0x80 0x00 0x80 0x00
64  64  64  64  64  64    // 0x40 0x40 0x40 0x40 0x40 0x40
255 192 128 64  0   0     // 0xFF 0xB0 0x80 0x40 0x00 0x00
```

ZADANIE 3A - `compressGrayscale()` - OPIS

Napisz funkcję `compressGrayscale()`. Powinna ona przyjąć jeden argument typu `std::array<std::array<uint8_t, 240>, 160>` określający rozkład odcieni szarości na obrazku 2D (który w dalszej części nazywać będziemy bitmapą) i zwróci `std::vector<std::pair<uint8_t, uint8_t>>` zawierający skompresowaną bitmapę.

Kompresja powinna przebiegać w następujący sposób:

- Bitmapę rysujemy od górnego lewego rogu przechodząc w prawo, następnie poziom niżej.
- Jeżeli obok siebie występuje ten sam kolor więcej niż 1 raz, funkcja powinna wrzucić do `std::vector<>` wartość tego koloru (liczba z przedziału 0 – 255) jako pierwszy element pary oraz ilość jego powtórzeń jako drugi element pary.
- Jeżeli obok siebie występują różne odcienie to funkcja powinna wypełnić `std::vector<>` wartością odcienia oraz liczbą wystąpień równą 1 (w tym przypadku pogarszamy optymalizację, gdyż przechowujemy 2x tyle danych, jednak najczęściej te same kolory są położone obok siebie).

ZADANIE 3A - `compressGrayscale()` - PRZYKŁAD

```
input: {{0 0 0 1 1 2 3 0 0 0},
        {0 0 4 4 4 1 1 1 1 1},
        {2 2 2 2 2 1 2 2 2 2}}
output: {{0, 3}, {1, 2}, {2, 1}, {3, 1}, {0, 3}, {0, 2}, {4, 3}, {1, 5}, {2, 5}, {1, 1}
```

W przypadku powyższej konwersji zamiast 30 bajtów (wymiary 10x3) zużyjemy 22 (11x2). Więc skompresowaliśmy dane o 26,7%.

Nie przejmujemy się na razie tym jak `uint_8` będzie zamieniany na kolor. Ważne w tym zadaniu jest, aby poćwiczyć korzystanie z kontenerów oraz wykonywania na nich różnych operacji.

Chętni mogą także zrefaktoryzować (czyli napisać czytelniej, ulepszyć) testy tak, aby te skomplikowane pętle, które wypełniają tablice były uniwersalną funkcją, możliwą do wywołania w obecnie istniejących i przyszłych testach (podobnie jak funkcja `getBitmap()`). Po wydzieleniu i refaktoringu funkcji generującej, postarajcie się dopisać także przypadki dla 1/16, 1/32 i 1/64 mapy.

CONTROLS

Jill of the Jungle

HI SCORES

LOOK	100
OUT	100
WORLD	100
HERE	100
COMES	100
EPIC	100
MEGA	100
GAMES	100
SSD	35
	0

INVENTORY

ZADANIE 3B - `decompressGrayscale()`

Napisz funkcję `decompressGrayscale()`, która zdekompresuje obrazek skompresowany w zadaniu 3 za pomocą funkcji `compressGrayscale()`.

Jako argument funkcja `decompressGrayscale()` przyjmie `std::vector<std::pair<uint8_t, uint8_t>>` natomiast zwróci `std::array<std::array<uint8_t, 240>, 160>` i przeprowadzi operacje mające na celu rekonstrukcję pierwotnego formatu bitmapy.

ASCII ART

Dla chętnych (bez punktów) polecamy także napisać sobie funkcję `printMap()`, która wyświetli mapę. Domyślnie `std::cout` potraktuje `uint8_t` jako `unsigned char`, dlatego też możecie sobie wypisać mapę z kodów ASCII.

Chętni mogą także zrefaktoryzować testy, tak by funkcja sprawdzająca mapę była generyczna (adekwatnie do funkcji `expectBitmap()`. Po refaktoringu funkcji sprawdzającej, postarajcie sie dopisać także przypadki dla 1/16, 1/32 i 1/64 mapy.

CODERS SCHOOL