STL #2

FUNKTORY, LAMBDY, ALGORYTMY

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

ZADANIA

Repo GH `coders-school/stl`

https://github.com/coders-school/stl/tree/master/module2

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?

AGENDA

- Funktory
- Wyrażenia lambda
- Algorytmy niemodyfikujące kontenery
- Algorytmy modyfikujące kontenery

FUNKTORY

OBIEKTY FUNKCYJNE

CODERS SCHOOL

CZYM JEST FUNKTOR - OBIEKT FUNKCYJNY

Funktor jest to obiekt, który może zostać wywołany jak zwykła funkcja. Każda klasa oraz struktura, która posiada zdefiniowany `operator()` może pełnić rolę funktora. Alternatywną nazwą dla funktora jest obiekt funkcyjny.

```
struct Functor {
    void operator() () {
        std::cout << "I'm a functor!\n";
    }
};

int main() {
    Functor funct;
    funct();      // obiekt nazwany (linię wyżej)
    Functor{}();  // obiekt tymczasowy

    return 0;
}
```

output:

```
I'm a functor!
I'm a functor!
```

CZYM JEST FUNKTOR - FUNKCJA

Funkcja także jest traktowana jako funktor, ponieważ również możemy ją wywołać poprzez `()`.

```
void function() {
    std::cout << "I'm a functor!\n";
}

int main() {
    function();

    return 0;
}
```

output: `I'm a functor!`

WYKORZYSTANIE FUNKTORÓW

Funktory możemy wykorzystać w algorytmach STL. Przykładowo algorytm `for_each` dla każdego elementu wywołuje on przekazany funktor. Sam funktor za swój jedyny argument musi przyjmować typ elementu z zakresu, na którym pracuje.

```
struct Functor {
    void operator()(int el) {
        std::cout << el << ' ';
    }
};

int main() {
    std::vector<int> vec {1, 2, 3, 4, 5};
    std::for_each(begin(vec), end(vec), Functor{});
    std::cout << '\n';

    return 0;
}
```

output: 1 2 3 4 5

Q&A

WYRAŻENIA LAMBDA

NOWOCZESNE FUNKTORY

CODERS SCHOOL

CZYM JEST WYRAŻENIE LAMBDA?

- Wprowadzone w C++11, ulepszane w C++14, C++17, C++20
- Lambda to obiekt funkcyjny, który może zostać wywołany dla konkretnych parametrów i zwrócić wynik
- Prosta w budowie - `[ ] ( ) { }`
- Służy do zwięzłego zapisu obiektu funkcyjnego, który normalnie zająłby nam kilka razy więcej miejsca
- Zyskujemy lepszą czytelność oraz większą swobodę w działaniu
- Typ lambdy nazywa się domknięciem (ang. closure) i jest znany tylko kompilatorowi
- Aby przypisać wyrażenie lambda do zmiennej musimy być ona typu auto, ponieważ tylko kompilator zna typ tego wyrażenia

UTWORZENIE PROSTEGO WYRAŻENIA LAMBDA

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    []() { std::cout << "Hello"; }    // lambda printing Hello (not called)

    std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8, 9};
    vec.erase(std::remove_if(vec.begin(),
                             vec.end(),
                             [](int num) { return num % 2; }),
              vec.end());

    auto print = [](int num) { std::cout << num << ' '; };
    std::for_each(vec.begin(), vec.end(), print);

    return 0;
}
```

LISTA PRZECHWYTUJĄCA (CAPTURE LIST) - PRZYKŁAD

```
int main() {
    int number = 10;
    auto add10 = [number](int num) { return num + number; };
    std::cout << add10(20) << '\n';  // 30

    // auto multiplyByX = [](int num) { return num * number; };
    // compilation error: number not declared

    return 0;
}
```

LISTA PRZECHWYTUJĄCA (CAPTURE LIST)

- Kwadratowy nawias `[ ]` określa, jakie parametry chcemy przechwycić do naszego wyrażenia. Możemy np. chcieć przechwycić jakąś zmienną, z którą będziemy chcieli porównywać każdy element kontenera
- Wartości możemy przechwycić przez referencję `[&value]`
- Możemy je także przechwycić przez kopię `[value]`
- Możemy też mieszać obie możliwości `[&by_ref, by_copy, by_copy2]`
- Wyrażenia lambda umożliwia także przechwycenie wszystkiego, co potrzebujemy:
  - poprzez kopię `[=]`
  - poprzez referencję `[&]`
- O ile `[=]` jest bezpieczne, to `[&]` nie zawsze jest zalecane

Pytanie: kiedy [&] może byc niebezpieczne?

GENERYCZNE LAMBDY

- Od C++14 możemy pisać tzw. generyczne lambdy
- Są to lambdy wielokrotnego użytku (dla różnych typów) i używamy w nich typu `auto` jako parametru
  - `[](const auto first, const auto& second, auto third) {}`
- Pisanie generycznych lambd jest opłacalne, ponieważ łatwo można je wielokrotnie wykorzystać

```
int multiply(int first, int second) {
    return first * second;
}

int main() {
    int number = 10;
    auto multiplyByX = [&number](auto num) { return multiply(num, number); };
    std::cout << multiplyByX(20) << '\n';

    return 0;
}
```

TYP ZWRACANY

- Nie podajemy typu zwracanego, gdyż domyślnie wyrażenie lambda dedukuje ten typ poprzez dane zawarte w jej ciele `{}`
- Domyślnie typ zwracany przez wyrażenie lambda jest dedukowany na podstawie wyrażeń `return`

```
[i{0}](const int el){ return el + i; }; // return type is int
```

- Jeżeli chcemy narzucić konkretny typ zwracany robimy to poprzez `->`

```
[i{0}](const auto el) -> double { return el + i; };
```

Q&A

TYM RAZEM PRZED ZADANIAMI, BO TEMAT TRUDNY :)

ZADANIE

1. Utwórz funktor sprawdzający czy podana liczba typu `int` jest podzielna przez 6
2. Utwórz lambdę, która przyjmie 2 argumenty typu `int` oraz zwróci ich iloczyn
3. Utwórz lambdę, która do podanego ciągu znaków doda cudzysłów. np.
   - `krowa -> "krowa"`
4. Utwórz lambdę, która wypisze ciąg znaków `*`. Przy każdym zawołaniu funkcji powinniśmy dostać ciąg dłuższy o jedną `*`. Kolejno:
   - `*`
   - `**`
   - `***`
   - itd.

ZADANIE

1. Utwórz `std::vector<int>` i wypełnij go dowolnymi wartościami
2. Utwórz lambdę, która przechwyci ten wektor, oraz wyświetli jego zawartość
3. Utwórz lambdę, która w swoim argumencie przyjmie `int` i go wyświetli
4. Wykorzystaj tą lambdę w algorytmie `std::for_each()` do wyświetlenia całego kontenera

ALGORYTMY NIEMODYFIKUJĄCE

CODERS SCHOOL

ALGORYTMY NIEMODYFIKUJĄCE KONTENERÓW

Nie modyfikują one kontenerów na których działają.

Nie mogą:

- zmieniać kolejności elementów w kontenerze
- usuwać elementy
- dodawać elementy

Tutaj znajdziesz tylko popularne lub ciekawe użycia niektórych algorytmów. Pełna lista algorytmów dostępna jest na cppreferene.com

Algorytmy na cppreference.com

std::find_if

```
template< class InputIt, class UnaryPredicate >
InputIt find_if( InputIt first, InputIt last, UnaryPredicate p );
```

- Predykat = funktor, funkcja, lambda, która zwraca `bool` (`true/false`)
- Algorytm wykorzystywany do wyszukiwania interesujących nas elementów. O tym co nas interesuje będzie decydować przekazany predykat. Jeżeli chcemy liczby podzielnie przez 3 użyjemy predykatu:
  - `[](const auto& el){ return (el % 3 == 0); }`
- `std::find_if` różni się od `std::find` tylko tym, że zamiast poszukiwanej wartości, podajemy predykat jaki musi zostać spełniony aby uznać dany element za poszukiwana wartość.
- Typem zwróconym przez `std::find_if` jest iterator, wskazujący na znaleziony element.
- Jeżeli element nie został znaleziony, wartością zwróconą będzie równy `last`.

std::find_if - UŻYCIE

```
std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto found = std::find_if(begin(vec), end(vec), [](const auto& el) {
    return el == 7;
});
if (found != vec.end()) {
    std::cout << *found << '\n';
}
```

Output: `found 7`

std::search

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 search( ForwardIt1 first, ForwardIt1 last,
                   ForwardIt2 s_first, ForwardIt2 s_last );
```

- Najprostsza wersja `std::search`, przyjmuje 2 zakresy i sprawdza, czy zakres drugi `{s_first, s_last}` jest podzakresem `{first, last}`.
  - Jeżeli tak zwraca iterator wskazujący na początek tego podzakresu.
  - Jeżeli podzakres nie zostanie znaleziony, zwrócony iterator będzie równy `last`.
- Istnieją także wersje `std::search`, przyjmujące `binary predicate` oraz typ `searcher`. Zachęcam Was do samodzielnej pracy w celu zdobycia wiedzy, jak wykorzystać te funkcję :).

std::search - UŻYCIE

```
std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> vec2 {4, 5, 6};
auto found = std::search(begin(vec), end(vec),
                         begin(vec2), end(vec2));
if (found != vec.end()) {
    std::cout << "first found element: " << *found << '\n';
}
```

Output: `first found element: 4`

std::count ORAZ std::count_if

```
template< class InputIt, class T >
typename iterator_traits<InputIt>::difference_type
    count( InputIt first, InputIt last, const T &value );
```

```
template< class InputIt, class UnaryPredicate >
typename iterator_traits<InputIt>::difference_type
    count_if( InputIt first, InputIt last, UnaryPredicate p );
```

- `std::count` zlicza dla danego zakresu wystąpienie konkretnej wartości.
- `std::count_if` zlicza dla danego zakresu ilość zwróconych `true` przez predykat.

std::count ORAZ std::count_if - UŻYCIE

```cpp
std::vector<int> vec {1, 2, 3, 4, 5, 1, 1, 1, 6, 7};

std::cout << std::count(begin(vec), end(vec), 1) << '\n';

auto counter = std::count_if(begin(vec), end(vec), [](const auto& el){
    return el % 3 == 0;
});
std::cout << counter << '\n';
```

Output:

```
4
2
```

std::equal

```
template< class InputIt1, class InputIt2 >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2 );
```

```
template< class InputIt1, class InputIt2 >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2, InputIt2 last2 )
```

- Algorytm przyjmuje zakres pierwszego kontenera oraz początek drugiego kontenera. Funkcja będzie dokonywać sprawdzenia, aż nie dojdzie do końca 1 zakresu, nawet jeżeli 2 zakres jest dłuższy. Jeżeli 2 zakres jest krótszy to funkcja zwróci false, gdyż na pewno 1 zakres nie będzie identyczny ponieważ jest dłuższy.
- Wersja druga algorytmu, pozwala przyjąć pełny zakres 1 i 2 kontenera i porównać te zakresy.

`std::equal` - UŻYCIE #1

```
// Missing vec1 and vec2 :D Can you think of the examples of vec1 and 2 that
// will make the output look like below?
std::cout << std::boolalpha << "EQUAL?: "
          << std::equal(begin(vec1), end(vec1), begin(vec2)) << '\n';
std::cout << std::boolalpha << "EQUAL?: "
          << std::equal(begin(vec2), end(vec2), begin(vec1)) << '\n';
std::cout << std::boolalpha << "EQUAL?: "
          << std::equal(begin(vec1), end(vec1),
                        begin(vec2), std::next(vec2.begin(), 5))
          << '\n';
```

Output:

```
EQUAL?: true
EQUAL?: false
EQUAL?: true
```

std::equal - UŻYCIE #2

```
bool is_palindrome(const std::string& s) {
    return std::equal(s.begin(), s.begin() + s.size()/2, s.rbegin());
}
void test(const std::string& s) {
    std::cout << "\"" << s << "\" "
              << (is_palindrome(s) ? "is" : "is not")
              << " a palindrome\n";
}
int main() {
    test("radar");
    test("hello");
}
```

Output:

```
"radar" is a palindrome
"hello" is not a palindrome
```

std::mismatch

```
template< class InputIt1, class InputIt2 >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,
              InputIt2 first2 );
```

```
template< class InputIt1, class InputIt2 >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2 );
```

Działa analogicznie do `std::equal`, z tym wyjątkiem, że zwraca parę iteratorów (dla pierwszego i drugiego zakresu) wskazującą początek niezgodności.

std::mismatch - UŻYCIE

```
std::string mirror_ends(const std::string& in) {
    return std::string(in.begin(),
                       std::mismatch(in.begin(), in.end(), in.rbegin()).first)
}

int main() {
    std::cout << mirror_ends("abXYZba") << '\n'
              << mirror_ends("abca") << '\n'
              << mirror_ends("aba") << '\n';
}
```

Output:

```
ab
a
aba
```

Q&A

ALGORYTMY MODYFIKUJĄCE

CODERS SCHOOL

ALGORYTMY MODYFIKUJĄCE KOLEJNOŚĆ ELEMENTÓW

Modyfikują one kontenery na których działają.

Mogą:

- zmieniać kolejności elementów w kontenerze
- usuwać elementy
- dodawać elementy

Tutaj znajdziesz tylko popularne lub ciekawe użycia niektórych algorytmów. Pełna lista algorytmów dostępna jest na cppreferene.com

Algorytmy na cppreference.com

`std::copy`, `std::copy_if`

```
template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );
```

```
template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt copy_if( InputIt first, InputIt last,
                  OutputIt d_first,
                  UnaryPredicate pred );
```

- Podstawowa wersja `std::copy`, kopiuje podany zakres do innego zakresu. Przykładowo kopiuje elementy z wektora do listy.
- `std::copy_if` kopiuje tylko te elementy, które spełniają podany przez nas predykat. Np. `::is_upper`, `::is::digit`.

`std::copy`, `std::copy_if` - UŻYCIE

```cpp
std::vector<int> vec {1, 2, 3, 4, 5};
std::array<int, 5> arr;
std::copy(begin(vec), end(vec), begin(arr));
print(arr);

std::vector<int> vec2(3);
std::copy_if(begin(vec), end(vec), begin(vec2), [](auto num) {
    return num % 2 == 1;
});
print(vec2);
```

Output:

```
1 2 3 4 5
1 3 5
```

std::fill

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

Funkcja wypełnia podany zakres wartościami `value`

```
int main() {
    std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::fill(v.begin(), v.end(), -1);
    for (auto elem : v) {
        std::cout << elem << " ";
    }
    std::cout << "\n";
}
```

Output: `-1 -1 -1 -1 -1 -1 -1 -1 -1 -1`

`std::transform`

Potężny algorytm, mogący zrobić dużo więcej niż się wydaje na początku :)

```
template< class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first1, InputIt last1,
                    OutputIt d_first,
                    UnaryOperation unary_op );
```

```
template< class InputIt1, class InputIt2, class OutputIt, class BinaryOperation >
OutputIt transform( InputIt1 first1, InputIt1 last1,
                    InputIt2 first2,
                    OutputIt d_first,
                    BinaryOperation binary_op );
```

- Pierwsza wersja `std::transform` przyjmuje zakres `[first1, last1)`, oraz wykonuje na każdym elemencie operację `unary_op`, a następnie zapisuje zmodyfikowane elementy w drugim zakresie (`d_first`).
- Druga wersja, przyjmuje 2 zakresy, pobiera z obu tych zakresów po 1 elemencie i wykonuje na nich operacje `binary_op`, następnie zapisuje wynik w 3 zakresie (`d_first`).

`std::transform` - PRZYKŁAD UŻYCIA #1

Konwersja jednego typu kontenera na drugi

```
int main() {
    std::vector<std::pair<int, std::string>> vec {
        {0, "Zero"},
        {1, "One"},
        {2, "Two"},
        {3, "Three"},
        {4, "Four"},
        {5, "Five"}
    };
    std::vector<int> vec2(5);
    std::transform(begin(vec), end(vec), vec2.begin(), [](const auto& pair) {
        return pair.first;
    });
    print(vec2);
    return 0;
}
```

Output: 0 1 2 3 4 5

`std::transform` - PRZYKŁAD UŻYCIA #2

Konwersja kontenera

```
std::vector<std::pair<int, std::string>> vec {
    {0, "Zero"},
    {1, "One"},
    {2, "Two"},
    {3, "Three"},
    {4, "Four"},
    {5, "Five"}
};
std::vector<std::string> vec2;
std::transform(begin(vec),
               end(vec),
               std::back_inserter(vec2),
               [](const auto& pair){
                   return pair.second + " : " + std::to_string(pair.first);
               });
print(vec2);
```

Output: `Zero : 0, One : 1, Two : 2, Three : 3, Four : 4, Five : 5`

`std::transform` - PRZYKŁAD UŻYCIA #3

Zamiana znaków na małe litery

```
int main() {
    std::vector<std::string> vec {
        "ZeRo", "ONe", "TwO", "ThREe", "FoUr", "FiVe"
    };
    std::transform(begin(vec), end(vec), begin(vec), [](auto str) {
        std::transform(begin(str), end(str), begin(str), [](auto c) {
            return std::tolower(c);
        });
        return str;
    });
    print(vec);

    return 0;
}
```

Output: `zero one two three four five`

`std::transform` PRZYKŁAD UŻYCIA #4

Sumowanie wartości wektora i listy:

```
int main() {
    std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8};
    std::list<int> list {10, 20, 30 , 40 , 50, 60, 70, 80};
    std::transform(begin(vec),
                   end(vec),
                   begin(list),
                   begin(vec),
                   [](auto first, auto second) {
                       return first + second;
                   });
    print(vec);

    return 0;
}
```

Output: `11 22 33 44 55 66 77 88`

std::generate

```
template< class ForwardIt, class Generator >
void generate( ForwardIt first, ForwardIt last, Generator g );
```

Funkcja służąca do generowania danych.

```
int main() {
    std::vector<int> vec(10);
    std::generate(begin(vec), end(vec), [i{0}]() mutable { return i++; });
    print(vec);

    return 0;
}
```

Output: 0 1 2 3 4 5 6 7 8 9

std::swap_ranges

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt2 swap_ranges( ForwardIt1 first1, ForwardIt1 last1,
                        ForwardIt2 first2 );
```

Podmienia pewien zakres danych

```
int main() {
    std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8};
    std::list<int> list {10, 20, 30, 40, 50, 60, 70, 80};
    std::swap_ranges(begin(vec), std::next(begin(vec), 3), std::begin(list));
    print(vec);
    print(list);

    return 0;
}
```

Output:

```
10 20 30 4 5 6 7 8
1 2 3 40 50 60 70 80
```

std::reverse

```
template< class BidirIt >
void reverse( BidirIt first, BidirIt last );
```

Odwraca zakres

```
int main() {
    std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8};
    std::reverse(begin(vec), end(vec));
    print(vec);

    return 0;
}
```

Output: 8 7 6 5 4 3 2 1

std::unique

```
template< class ForwardIt >
ForwardIt unique( ForwardIt first, ForwardIt last );
```

Usuwa duplikaty. Ważne! Kontener musi być posortowany. Tak naprawdę ta funkcja nie usuwa duplikatów, lecz przenosi unikalne wartości na początek kontenera (nie zmieniając ich wzajemnej kolejności) oraz zwraca iterator wskazujący pierwszy element, gdzie zaczynają się duplikaty.

std::unique - PRZYKŁAD

```cpp
int main() {
    std::vector<int> vec {1, 2, 1, 2, 1, 2, 3, 2, 3, 1, 3, 2, 1};
    std::sort(begin(vec), end(vec));
    auto it = std::unique(begin(vec), end(vec));
    print(vec);
    vec.erase(it, end(vec));
    print(vec);

    return 0;
}
```

Output:

```
1 2 3 1 1 2 2 2 2 2 3 3 3
1 2 3
```

ZADANIE

1. Stwórz `std::vector<int>`
2. Wypełnij go elementami nieparzystymi licząc od 1 do 15
3. Odwróć kontener nie używając pętli ani `std::reverse`
4. Przepisz `std::vector<int>` do listy używając `std::copy`
5. Stwórz drugi `std::vector<int>` i wypełnij go liczbami parzystymi od 0 do 14.
6. Znajdź sposób jak połączyć oba wektory w jeden, zawierający wartości od 0 do 15 ułożone po kolei.

ZADANIE

1. Stwórz `std::list<int>` z wartościami od 1 do 10.
2. Utwórz `std::vector<int>` z wartościami od 5 do 10.
3. Przekaż odpowiednie iteratory do funkcji `std::equal`, tak by zwróciła, że oba kontenery są sobie równe.
4. Za pomocą `std::mismatch` oraz `erase`, usuń niepasujące elementy z listy
5. Zawołaj funkcję `std::equal` dla pełnych zakresów aby upewnić się, że są teraz identyczne.

Q&A

STL #2

FUNKTORY, LAMBDY, ALGORYTMY

PODSUMOWANIE

CODERS SCHOOL

CO BYŁO ŁATWE DO ZROZUMIENIA?

CO CHCESZ POWTÓRZYĆ NA SESJI Q&A?

ZAPISZ I WRZUĆ NA KANAŁ #POWTÓRKA NA DISCORDZIE :)

PRACA DOMOWA

POST-WORK

- Zadanie 1 - kontynuacja poprzedniego zadania 3 - `grayscaleImages` (6 punktów)
- Zadanie 2 - `insensitivePalindrom` (6 punktów)
- Zadanie 3 - `transformContainers` (6 punktów)

PR zgłaszajcie również na gałąź `master` w repo `coders-school/stl`, ale dodajcie obowiązkowo w tytule `STL#2`. U was gałąź może być dowolna.

BONUSY

- 2 punkty za każde zadanie dostarczone przed 21.06.2020 (niedziela) do 23:59
- 3 punkty za pracę w grupie dla każdej osoby z grupy. Zalecamy grupy 3 osobowe (takie jak ostatnio)

ZADANIA W REPO

PRE-WORK

- Przygotuj listę pytań na sesję Q&A :)
- Upewnij się, że działa Ci mikrofon i możesz rozmawiać na Discordzie
- Jeśli masz wątpliwości lub niedziałające implementacje i chcesz je skonsultować - podeślij link do repo na kanale #powtórka na Discordzie.
  - Pytania możesz oznaczyć komentarzami w kodzie
  - Pytania możesz też napisać w komentarzu do PR
  - Dodatkowe pytania możesz też później zadać na żywo podczas omawiania kodu

ZADANIE 1 - `grayscaleImages` CD.

Ulepsz program `grayscaleImages` z poprzednich zajęć STL#1 (kompresja, dekompresja) obrazków, tak, aby zamiast pętli wykorzystać algorytmy. Działaj na swojej/waszej wersji implementacji tego zadania.

ZADANIE 2 - `insensitivePalindrom`

Napisz program, który będzie sprawdzał, czy podany ciąg wyrazów jest palindromem. Program powinien ignorować znaki specjalne jak `,.?()[]`, wielkość liter oraz białe znaki jak spacja czy znak nowej linii.

Tutaj wywnioskuj po testach jak nazwać funkcję. Stwórz własne pliki źródłowe i nagłówkowe i dodaj je do CMakeLists.txt, aby się budowały. Szczegóły w README.md do tego zadania.

ZADANIE 3 - `transformContainers`

1. Napisz funkcję `transformContainers`, która przyjmie `std::list<std::string>` oraz `std::deque<int>`
2. Usunie duplikaty z obu kontenerów
3. Na koniec skonwertuje to na `std::map<int, std::string>` i ją zwróci. Użyj `std::transform`.

Tutaj wywnioskuj po testach jak nazwać funkcję. Stwórz własne pliki źródłowe i nagłówkowe i dodaj je do CMakeLists.txt, aby się budowały. Szczegóły w README.md do tego zadania.

CODERS SCHOOL