PODSTAWY C++ #3

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

AGENDA

1. Zakresy (scopes)
2. Referencje
3. Wskaźniki
4. Zagrożenia przy stosowaniu referencji i wskaźników
5. Typ wyliczeniowy `enum` i `enum class`

ZADANIA

Repo GH `coders-school/kurs_cpp_podstawowy`

https://github.com/coders-school/kurs_cpp_podstawowy/tree/module3/module3

KRÓTKIE PRZYPOMNIENIE

CO JUŻ WIEMY

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

PODSTAWY C++

ZASIĘG ZMIENNYCH

CODERS SCHOOL

ZMIENNE LOKALNE

Zmienne lokalne są to zmienne, które są widziane w obrębie jakiegoś zakresu.

```
{
    int local_variable = 5;
    // ...
}
local_variable = 10;   // error -> local_variable doesn't exists
```

Zakres zawsze tworzą nawiasy klamrowe m.in:

- same nawiasy - `{ /* ... */ }`
- ciała funkcji - `void fun() { /* ... */ }`
- instrukcje warunkowe - `if (condition) { /* ... */ }`
- pętle - `while (condition) { /* ... */ }`

ZMIENNE GLOBALNE

Zmienna globalna, jest widoczna dla wszystkich zakresów. Zawsze możemy się do niej odwołać.

```
int global_value = 5;

void foo() {
    std::cout << global_value;
}

int main() {
    std::cout << global_value;
}
```

Tworzenie zmiennych globalnych zazwyczaj jest złą praktyką.

CO WYPISZE SIĘ NA EKRANIE?

```
int number = 1;

int main() {
    int number = 2;
    {
        int number = 3;
        std::cout << number;
        std::cout << ::number;
    }
    std::cout << number;
    std::cout << ::number;
}
```

3121

PRZESŁANIANIE NAZW

- możemy mieć wiele zmiennych o takiej samej nazwie, jeśli będą w różnych zakresach
  - aby unikać niejednoznaczności nie jest to raczej polecane
- nazwa z lokalnego zakresu zawsze przesłania tę z szerszego zakresu (np. globalnego)
- można odwoływać się do nazw z globalnego zakresu stosując `::` (operator zakresu)

PODSTAWY C++

REFERENCJE

CODERS SCHOOL

&

Magiczny znaczek `&` oznacza referencję.

```
int value = 5;
int & number = value;
```

Powyższy zapis oznacza zmienną `num` typu `int&`, czyli referencję na typ `int`.

Nie ma znaczenia, czy referencję dokleimy do typu, czy nazwy zmiennej, ale referencja jest oddzielnym typem, więc sugerujemy nie doklejać jej do nazwy zmiennej.

```
int& number = value;  // lewak
int &number = value;  // prawak (odradzane)
int & number = value; // neutralny
```

CZYM JEST REFERENCJA?

Spójrzmy na fragment kodu.

```
int number = 5;
int& refer = number;

std::cout << refer << '\n';   // 5
std::cout << ++refer << "\n"; // 6
std::cout << number << "\n";  // 6
```

- Referencja odwołuje się do istniejącego obiektu
- Jeżeli utworzymy obiekt `int value` to poprzez referencje `int& reference = value` będziemy mogli się do niego bezpośrednio odwoływać.
- Referencja to inna, dodatkowa nazwa dla tej samej zmiennej (alias)
- Modyfikacja referencji = modyfikacja oryginalnego obiektu

CO ZYSKUJEMY W TEN SPOSÓB?

- Nie musimy kopiować elementów. Wystarczy, że przekażemy referencje.
  - W ten sposób możemy swobodnie w wielu miejscach programu odczytywać wartość tej zmiennej, bez zbędnego jej kopiowania.
- Referencja zajmuje w pamięci tyle, ile zajmuje adres (4 lub 8 bajtów).
- Tworzenie referencji do typu `int` (zazwyczaj 4 bajty) nie zawsze ma sens optymalizacyjny, chyba, że chcemy zmodyfikować ten element wewnątrz funkcji.
- Przekazywanie argumentów przez referencje nabierze więcej sensu, kiedy poznamy już klasy i obiekty :)

Ile miejsca zajmuje referencja? - stackoverflow.com

JAK PRZEKAZAĆ ELEMENT PRZEZ REFERENCJĘ?

```
void foo(int& num) {
    std::cout << num; // good
    num += 2;         // good
}
```

Jeśli chcemy mieć pewność, że funkcja nie zmodyfikuje nam wartości (chcemy ją przekazać tylko do odczytu) dodajemy `const`.

```
void bar(const int& num) {
    std::cout << num; // good
    num += 2;         // compilation error, num is const reference
}
```

Wywołanie funkcji to po prostu:

```
int num = 5;
foo(num);
bar(num);
```

ZADANIE

Zaimplementuj funkcję `foo()`. Ma ona przyjąć i zmodyfikować przekazany tekst. Na ekranie chcemy zobaczyć `"Other string"`.

```cpp
#include <iostream>
#include <string>

// TODO: Implement foo()
// It should modify passed string to text "Other string"

int main() {
    std::string str("Some string");
    foo(str);
    std::cout << str << '\n';
    return 0;
}
```

PODSUMOWANIE

- referencja jest aliasem (inną nazwą dla zmiennej)
- modyfikacja referencji to modyfikacja oryginalnego obiektu
- przy przekazywaniu parametru przez referencję:
  - unikamy zbędnych kopii
  - modyfikacja obiektu będzie skutkowała zmodyfikowaniem oryginału przekazanego do funkcji

PODSTAWY C++

WSKAŹNIKI

CODERS SCHOOL

WSKAŹNIKI - ANALOGIA

Poza referencjami istnieją także wskaźniki. Wskaźniki działają podobnie jak referencje.

Wyobraźmy sobie, że planujemy wycieczkę na Majorkę. Wsiadamy do samolotu i lecimy. Na miejscu okazuje się, ze zapomnieliśmy jaki jest adres hotelu :(W celu znalezienia go musimy zadzwonić do biura podróży, poczekać na obsługę, wytłumaczyć całą zawiłą historię, aż w końcu po długim czasie otrzymujemy adres naszego hotelu. Proces zdobycia tych informacji był dla nas czasochłonny.

Wyobraźmy sobie jednak, że uprzednio zapisaliśmy sobie w telefonie adres naszego hotelu. Aby przypomnieć sobie, gdzie on się znajdował wystarczy, że sprawdzimy telefon i już wiemy. Proces ten zajął nam dużo mniej czasu.

WSKAŹNIKI W C++

Podobnie jest w C++. Wskaźniki służą do wskazywania miejsca w pamięci, gdzie znajduje się pożądany przez nas obiekt.

Procesor nie musi odpytywać każdorazowo magistrale pamięci, gdzie znajduje się podana zmienna, tylko od razu wie, jaki jest jej adres (unikamy pośredników jak telefon do biura obsługi).

Ponadto jeżeli funkcja przyjmuje wskaźnik, nie musi ona kopiować całej zawartości obiektu, co jest czasochłonne. Można dużo szybciej wskazać gdzie ten obiekt już istnieje.

JAK PRZEKAZAĆ ELEMENT PRZEZ WSKAŹNIK?

```
void foo (int* num) {
    std::cout << *num;  // good
    *num += 2;          // good
}
```

Gdy chcemy mieć pewność, że nikt nie zmodyfikuje nam wartości (chcemy ją przekazać tylko do odczytu) dodajemy `const`.

```
void bar (int const* num) {
    std::cout << *num;  // good
    *num += 2;          // compilation error, num is a pointer to const
}
```

Wywołanie funkcji to:

```
int num = 5;
foo(&num);
bar(&num);
```

GDZIE DAĆ CONST?

CO TO JEST?

```
const int * ptr;
```

Wskaźnik na stałą (`const int`).

```
int const * ptr;
```

Również wskaźnik na stałą (`const int = int const`).

```
int * const ptr;
```

Stały wskaźnik na zmienną (`int`).

STAŁE WSKAŹNIKI A WSKAŹNIKI NA STAŁE

```
int const * const ptr;
const int * const ptr;
```

Stały wskaźnik na stałą (`int const = const int`).

Jest to częste pytanie z rozmów kwalifikacyjnych. Aby stały był wskaźnik, `const` musi być za gwiazdką.

RÓŻNICE

WSKAŹNIK NA STAŁĄ

```
const int * ptr = new int{42};
*ptr = 43;      // compilation error: assignment of read-only location ‘* ptr’
ptr = nullptr; // ok
```

- Nie możemy zmodyfikować obiektu wskazywanego przez wskaźnik
  - Odwołania z * nie mogą modyfikować obiektu
- Możemy zmodyfikować sam wskaźnik, np. aby wskazywał na inny obiekt
  - Odwołania bez * mogą modyfikować wskaźnik

RÓŻNICE

STAŁY WSKAŹNIK

```
int * const ptr = new int{42};
*ptr = 43;      // ok
ptr = nullptr; // compilation error: assignment of read-only variable ‘ptr’
```

- Możemy zmodyfikować obiekt wskazywany przez wskaźnik
  - Odwołania z * mogą modyfikować obiekt
- Nie możemy zmodyfikować samego wskaźnika, np. aby wskazywał na inny obiekt
  - Odwołania bez * nie mogą modyfikować wskaźnika

STAŁY WSKAŹNIK NA STAŁĄ

```
const int * const ptr = new int{42};
*ptr = 43;      // compilation error: assignment of read-only location ‘* ptr’
ptr = nullptr; // compilation error: assignment of read-only variable ‘ptr’
```

- Nie możemy zmodyfikować obiektu wskazywanego przez wskaźnik
  - Odwołania z * nie mogą modyfikować obiektu
- Nie możemy zmodyfikować samego wskaźnika, np. aby wskazywał na inny obiekt
  - Odwołania bez * nie mogą modyfikować wskaźnika

ZADANIE

Zaimplementuj funkcje `foo()` i `bar()`.

`foo()` powinno zmodyfikować wartość przekazaną przez wskaźnik na 10, a `bar()` na 20.

Czy `foo()` lub `bar()` mogą przyjąć wskaźnik na stałą lub stały wskaźnik?

```
#include <iostream>

// TODO: Implement foo() and bar()
// foo() should modify value under passed pointer to 10
// bar() should modify value under passed pointer to 20
// Can we have a pointer to const or a const pointer?
int main() {
    int number = 5;
    int* pointer = &number;
    std::cout << number << '\n';
    foo(&number);
    std::cout << number << '\n';
    bar(pointer);
    std::cout << number << '\n';

    return 0;
}
```

RÓŻNICE MIĘDZY WSKAŹNIKIEM I REFERENCJĄ

ODWOŁANIA

- Do referencji odwołujemy się tak samo jak do zwykłego obiektu - za pomocą nazwy
- Aby uzyskać element wskazywany przez wskaźnik musimy dodać * przed nazwą wskaźnika

PRZEKAZYWANIE JAKO ARGUMENT

- Argument jest referencją lub zwykłą zmienną (kopią) - przekazujemy nazwę
- Argument jest wskaźnikiem a przekazujemy zmienną - musimy dodać & przed nazwą zmiennej.

OZNACZENIA

- Symbol * (operator dereferencji) oznacza dostęp do obiektu wskazywanego
- Jeżeli nie damy * przy wskaźniku dostaniemy adres obiektu wskazywanego
- Symbol & oznacza pobranie adresu naszej zmiennej
- Powyższe ma sens, ponieważ wskaźnik wskazuje miejsce w pamięci (adres wskazywanego obiektu)

RÓŻNICE W KODZIE

```
void copy(int a) { a += 2; }
void ref(int& a) { a += 2; }
void ptr(int* a) ( *a += 2; )

void example() {
    int c = 10;
    int& r = a;
    int* p = &a;  // typically int* p = new int{10};
    copy(c);
    copy(r);
    copy(*p);
    ref(c);
    ref(r);
    ref(*p);
    ptr(&c);
    ptr(&r);
    ptr(p);
}
```

CO OZNACZA * W KODZIE?

```
int a = 5 * 4;        // jako operacja arytmetyczna - mnożenie
int* b = &a;          // przy typie - wskaźnik na ten typ
int *c = &a;          // przy typie - wskaźnik na ten typ
std::cout << *b;      // przy zmiennej wskaźnikowej - dostęp do obiektu
int fun(int* wsk);    // w argumencie funkcji - przekazanie wskaźnika (adresu)
```

CO OZNACZA & W KODZIE?

```
int a = 5 & 4;        // jako operacja arytmetyczna - suma bitowa
int& b = a;           // przy typie - referencja na ten typ
int &c = a;           // przy typie - referencja na ten typ
std::cout << &a;      // przy zmiennej - adres tej zmiennej w pamięci
int fun(int& ref);    // w argumencie funkcji - przekazanie adresu
```

PODSTAWY C++

ZAGROŻENIA

W STOSOWANIU REFERENCJI I WSKAŹNIKÓW

CODERS SCHOOL

PUSTE WSKAŹNIKI

```
int* a = nullptr;
std::cout << *a;
```

Dostęp do zmiennej wskazywanej przez pusty wskaźnik to niezdefiniowane zachowanie.

Pusty wskaźnik oznaczamy zawsze używając `nullptr`.

Nie używamy `NULL` znanego z języka C lub wcześniejszych standardów, bo jest on mniej bezpieczny.

```
void foo(int);
foo(NULL);      // bad - no error
foo(nullptr);   // good - compilation error
```

NIEZAINICJALIZOWANE WSKAŹNIKI

```
int* a;
std::cout << *a;
```

Wskaźnik `a` zawiera tzw. śmieci. Dostęp do obiektu wskazywanego przez taki wskaźnik to niezdefiniowane zachowanie.

ODWOŁANIA DO USUNIĘTYCH ZMIENNYCH

Jak już dobrze wiemy, zmienne lokalne są usuwane po wyjściu z zakresu, w którym je utworzyliśmy. Można już domyślać się, jakie problemy sprawią nam wskaźniki i referencje, gdy będą dalej istniały, a obiekt, do którego się odwołują już zostanie zniszczony. Będzie to w najlepszym przypadku „crash", w najgorszym „undefined behaviour".

JAK ZAPOBIEGAĆ TAKIM PRZYPADKOM?

Zawsze musimy zapewnić, aby czas życia zmiennej, był dłuższy niż czas życia jej wskaźnika, czy referencji.

USUNIĘTE ZMIENNE - PRZYKŁAD

```
std::vector<int*> vec;

void createAndAddToVec(int amount) {
    for (int i = 0 ; i < amount ; ++i) {
        vec.push_back(&i);
    }
    // local variable i does not exist here anymore
    // vec contains addresses to not existing local variables
}

int main() {
    createAndAddToVec(5);
    for (const auto& el : vec) {
        std::cout << *el << '\n';   // UB
    }
}
```

JAK SOBIE PORADZIĆ Z TAKIM PROBLEMEM?

Odpowiedzią może być dynamicznie alokowana pamięć.

Najprościej jest to osiągnąć używając biblioteki `#include <memory>`, która posiada `std::shared_ptr<T>`.

Wskaźnik ten nie bez powodu nazywany jest *inteligentnym*. Odpowiada on za zarządzanie dynamiczną pamięcią i sam zwalnia zasób, gdy już go nie potrzebujemy.

JAK TAKI WSKAŹNIK UTWORZYĆ?

```
auto ptr = std::make_shared<int>(5);  // preferred
auto ptr = std::shared_ptr<int>(new int{5});
```

POPRAWIONY LISTING

```
std::vector<std::shared_ptr<int>> vec; // previously: std::vector<int*> vec;

void createAndAddToVec(int amount) {
    for (int i = 0 ; i < amount ; ++i) {
        vec.push_back(std::make_shared<int>(i));
        // previously: vec.push_back(&i);

        // the same in 2 lines:
        // auto num = std::make_shared<int>(i);
        // vec.push_back(num);
    }
}

int main() {
    createAndAddToVec(5);
    for (const auto& el : vec) {
        std::cout << *el << '\n';
    }
}
```

ZADANIE

Napisz funkcję `foo()`. Ma ona przyjmować shared_ptr na int i ma przypisać wartość 20 do wskazywanego przez niego obiektu.

Ponadto `foo()` ma wyświetlić wartość inta wskazywanego przez wskaźnik oraz liczbę shared_ptrów, które wskazują na ten obiekt.

Wyświetl także to samo w `main()` przed i po zawołaniu `foo()`.

```
#include <iostream>
#include <memory>

// TODO: Implement foo()
// It should take shared_ptr to int and assign value 20 to the pointed int.
// It should also display the value of this int and the number of how many pointers are poin
// Display the same information in main() before and after calling foo()

int main() {
    std::shared_ptr<int> number = std::make_shared<int>(10);
    // display the value under number pointer and use_count() of it
    foo(number);
    // display the value under number pointer and use_count() of it

    return 0;
}
```

ZADANIE

Napisz funkcję `foo()`. Ma ona przyjąć 2 wartości typu `int` oraz zwrócić ich iloczyn jako `shared_ptr`. Sprawdź ilu właścicieli posiada `shared_ptr`.

```
#include <iostream>

// TODO: Implement foo()
// It should take 2 int values and return their product as a shared_ptr.
// Additionally, check how many owners are there.

int main() {
    auto number = foo(10, 20);
    std::cout << "num: " << *number << " | owners: " << number.use_count() <<

    return 0;
}
```

INTELIGENTNE WSKAŹNIKI ROZWIĄZANIEM WSZYSTKICH PROBLEMÓW?

Teraz po utworzeniu inteligentnego wskaźnika, nie musimy się martwić o czas życia zmiennej. Możemy spokojnie po wyjściu z funkcji wypisać te wartości.

Jeżeli funkcja potrzebuje przyjąć zwykły wskaźnik (ang. raw pointer), czyli np. `int* i` możemy to zrobić używając funkcji `std::shared_ptr::get()` jak na przykładzie:

```
void foo(int* num) {
    do_sth(num);
}

int main() {
    auto ptr = std::make_shared<int>(5);
    foo(ptr.get())
}
```

PUŁAPKA POWRACA

```
void foo(int* num) {
    if (num) {
        do_sth(num);
    }
}

int main() {
    auto ptr = std::make_shared<int>(5);
    int* raw = ptr.get();
    ptr.reset();    // delete object, deallocate memory
    foo(raw);       // problem, dangling pointer is passed
    foo(ptr.get()); // not a problem, nullptr is passed
}
```

Jeżeli wszystkie obiekty `shared_ptr<T>` odwołujące się do tej zmiennej zostaną usunięte, to zasób zostanie zwolniony.

Nasz zwykły wskaźnik, który pobraliśmy wcześniej za pomocą `get()`, będzie posiadał adres do nieistniejącego już zasobu.

Próba jego użycia spowoduje UB lub crash. Należy bardzo uważać na zwykłe wskaźniki.

WNIOSKI

- wskaźniki mogą nie wskazywać na nic (`nullptr`), referencje muszą wskazywać na jakiś wcześniej stworzony obiekt
- wskaźniki i referencje mogą być niebezpieczne (częściej wskaźniki), jeśli są powiązane z nieistniejącymi już obiektami
  - są to tzw. dangling pointers/references, wiszące wskaźniki/referencje
- referencji nie można przypisać innego obiektu niż ten podany podczas jej inicjalizacji
- wskaźnikom można przypisać nowe adresy, aby wskazywały inne obiekty (za wyjątkiem stałych wskaźników)
- lepiej domyślnie nie używać zwykłych wskaźników (raw pointers)
- lepiej stosować inteligentne wskaźniki

PODSTAWY C++

enum | enum class

CODERS SCHOOL

TYP WYLICZENIOWY

`enum` to po polsku typ wyliczeniowy. W C++11 wprowadzono także `enum class` zwany silnym typem wyliczeniowym.

PRZYKŁAD

Załóżmy, że piszemy oprogramowanie do pralki. Chcielibyśmy utworzyć także interfejs zwracający numer błędu np:

- brak wody
- zbyt duże obciążenie
- problem z łożyskami
- blokada pompy

W celu warto użyć typu `enum` lub lepiej - `enum class`.

IMPLEMENTACJA PRZYKŁADU

```
enum ErrorCode {
    lack_of_water;
    too_much_load;
    bearing_problem;
    block_of_pump;
};

// or better ↓

enum class ErrorCode {
    lack_of_water;
    too_much_load;
    bearing_problem;
    block_of_pump;
};
```

NUMERACJA

Typ `enum` pod spodem numerowany jest od `0` do `n - 1`, gdzie `n` to liczba elementów.

Jeżeli chcemy nadać inne wartości musimy to zrobić ręcznie:

```
enum class ErrorCode {
  lack_of_water = 333;
  to_much_load; // will be 334
  bearing_problem = 600;
  block_of_pump; // will be 601
}
```

`enum` VS `enum class`

`enum` od `enum class` różni się głównie tym, że możemy niejawnie skonwertować typ `enum` na `int` (w końcu to typ wyliczeniowy).

Natomiast typ `enum class` możemy skonwertować na `int`, tylko poprzez jawne rzutowanie. Nie będziemy na razie omawiać rzutowania. Warto tylko pamiętać, że robimy to wywołując:

```
int num = static_cast<int>(ErrorCode::lack_of_water)
```

W jakich innych przepadkach zastosowalibyście typ wyliczeniowy?

`enum` VS `enum class`

Druga różnica - dla `enum` możemy mieć konflikt nazw, dla `enum class` nie.

```
enum Color {
    RED,    // 0
    GREEN,  // 1
    BLUE    // 2
};
```

```
enum TrafficLight {
    GREEN,  // 0
    YELLOW, // 1
    RED     // 2
};
```

```
auto lightColor = getColor();
if (lightColor == RED) {  // 0 or 2?
    stop();
} else {
    go();
}
```

UŻYCIE WARTOŚCI Z `enum class`

Aby uniknąć konfliktu nazw stosujemy `enum class`.

```
enum class Color {
    RED,
    GREEN,
    BLUE,
}
```

```
enum class TrafficLight {
    GREEN,
    YELLOW,
    RED
}
```

```
auto lightColor = getColor();
if (lightColor == TrafficLight::RED) {
    stop();
} else {
    go();
}
```

PODSTAWY C++

PODSUMOWANIE

CODERS SCHOOL

CO PAMIĘTASZ Z DZISIAJ?

NAPISZ NA CZACIE JAK NAJWIĘCEJ HASEŁ

1. Zakresy (scopes)
2. Referencje
3. Wskaźniki
4. Zagrożenia przy stosowaniu referencji i wskaźników
5. Typ wyliczeniowy `enum` i `enum class`

PRACA DOMOWA

POST-WORK

- Zadanie 1 - PasswordCheck

BONUS ZA PUNKTUALNOŚĆ

Za dostarczenie każdego zadania przed 07.06.2020 (niedziela) do 23:59 dostaniesz 2 bonusowe punkty za każde zadanie

ZADANIA W REPO

PRE-WORK

- Playlista na YT odnośnie STLa - obejrzyj ile możesz :)
- Przypomnij sobie czym jest klasa i jak się ją pisze - obejrzyj wideo Mirosława Zelenta. UWAGA! Od 22 minuty możesz zaobserwować trochę złych praktyk :D Spróbuj odgadnąć jakich.

ZADANIE PASSWORDCHECK

Tworzysz moduł do sprawdzania reguł haseł, który otrzymuje dane z formularza kontaktowego z front-endu. Cały moduł to zestaw kilku funkcji. Ich deklaracje mają znaleźć się w pliku nagłówkowym validation.hpp, a implementacje w pliku źródłowym validation.cpp. Twoje zadania to:

1. Zdefiniuj w pliku nagłówkowym nowy typ `ErrorCode` z możliwymi wartościami dla błędów przy ustalaniu nowego hasła (1 punkt)
    - Ok
    - PasswordNeedsAtLeastNineCharacters
    - PasswordNeedsAtLeastOneNumber
    - PasswordNeedsAtLeastOneSpecialCharacter
    - PasswordNeedsAtLeastOneUppercaseLetter
    - PasswordsDoesNotMatch

ZADANIE PASSWORDCHECK CD.

2. Napisz funkcję `getErrorMessage()`, która przyjmie zdefiniowany typ kodu błędu i zwróci stosowny komunikat jako napis. (1 punkt)
3. Napisz funkcję `doesPasswordsMatch()`, która otrzyma dwa hasła i sprawdzi czy są identyczne. Powinna zwrócić odpowiednią wartość typu bool. (1 punkt)
4. Napisz funkcję `checkPasswordRules()`, która przyjmie jedno hasło i losowo zwróci jeden z kodów `PasswordNeedsAtLeast*` lub `Ok`. (2 punkty)
5. Napisz funkcję `checkPassword()`. Powinna ona przyjąć dwa hasła i używać funkcji `doesPasswordsMatch()` w celu określenia czy hasła się zgadzają. W przypadku gdy się nie zgadzają ma ona zwrócić kod `PasswordsDoesNotMatch`, a w przeciwnym przypadku powinna zwrócić kod błędu zwrócony przez wywołanie funkcji `checkPasswordRules()`. (2 punkty)
6. Dla ambitnych (nieobowiązkowe) - zaimplementuj w funkcji `checkPasswordRules()` prawdziwą walidację pozostałych przypadków, które są podane jako kody błędów. Najlepiej, jeśli wykorzystasz funkcje z nagłówka oraz algorytm `std::any_of`. Dopisz/zmodyfikuj odpowiednie testy. (4 punkty)

Razem: 7 punktów (+4 dla ambitnych, +2 za dostarczenie przed 07.06.2020 23:59) Można pracować w parach, najlepiej innych niż ostatnio (bonus +3 punkty/osobę)

ZADANIE PASSWORDCHECK - PRZYKŁAD UŻYCIA

```
int main() {
    std::string password;
    std::string repeatedPassword;
    std::cout << "Set new password: ";
    std::cin >> password;
    std::cout << "Repeat password: ";
    std::cin >> repeatedPassword;
    auto result = checkPassword(password, repeatedPassword);
    std::cout << getErrorMessage(result) << '\n';

    return 0;
}
```

DOSTARCZENIE ZADAŃ

1. Jeśli nie masz jeszcze forka repo kurs_cpp_podstawowy i podpiętego w nim remote coders, to zobacz wcześniejsze prace domowe z Podstaw C++ #2 oraz #1.
2. Zaktualizuj swoje repo z remote'a coders - `git fetch coders`
3. Przełącz się na branch module3 - `git checkout module3`
4. Wyślij branch module3 na swojego forka - `git push origin module3`
5. Utwórz oddzielną gałąź na pracę domową - `git checkout -b homework3`
6. Wyślij od razu tę gałąź na forka, zanim zaczniesz implementację - `git push origin homework3`
7. Rozpocznij implementację samemu lub w parze.
8. Zanim wyślesz swoje zmiany za pomocą `git push origin homework3` synchronizuj się z forkiem, aby sprawdzić, czy druga osoba już czegoś nie dostarczyła - `git pull --rebase origin homework3`. Jeśli będą konflikty to je rozwiąż.
9. Przy zgłoszeniu Pull Requesta wyklikaj, że chcesz go dostarczyć do `coders-school/kurs_cpp_podstawowy` branch `module3`. Opisz go odpowiednio dodając informację o autorach kodu.

KOLEJNE ZAJĘCIA

- Powtórka z podstaw C++ i narzędzi
- Omówienie rozwiązań dotychczasowych zadań
- Omówienie najczęstszych błędów na podstawie Code Review
- Q&A
- Uwagi
- Konsultacje grupowe na Discordzie

CODERS SCHOOL