DOBRE PRAKTYKI #1

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

AGENDA

1. Współpraca zespołowa
2. Konwencje
3. Cpp Core Guidelines
4. Dobre praktyki

WSPÓŁPRACA GRUPOWA

CODERS SCHOOL

FILOZOFICZNE PYTANIA

A.K.A. WSPÓLNE RETRO

- Jak się czuliście podczas współpracy?
- Czy były problemy, których nie przewidzieliście na początku?
- Czy oszacowany czas zadań był zgodny z rzeczywistością?
- Co poszło nie tak?
- A co fajnie zadziałało i moglibyście zasugerować wdrożenie tego innym grupom?

NIESPODZIEWANE PROBLEMY

To, że tak trochę rzuciliśmy was w ten projekt i zmusiliśmy do samoorganizacji było zabiegiem celowym.

Chcieliśmy wam pokazać, ilu rzeczy na początku w ogóle się nie uwzględnia przed rozpoczęciem prac nad projektem.

I nawet mając kilkuletnie doświadczenie w zarządzaniu projektami, pewnych rzeczy nie da się uniknąć.

- choroby członków zespołu
- niespodziewane wyjazdy
- dodatkowe, niezaplanowane zadania
- źle oszacowany czas zadań
- problemy techniczne (restartujący się komputer, git - konflikty, przerywający internet)
- zmieniające się wymagania lub ich interpretacje

ESTYMOWANIE ZADAŃ

Teoria wskazuje, że dość fajnym sposobem na lepsze szacowanie zadań jest pomnożenie początkowej estymaty przez PI 😄

Inne źródła wskazują, że należy zwiększyć rząd wielkości (np. z 5 godzin na 5 dni, z 3 dni na 3 tygodnie, itp.) 😄

W praktyce do estymacji wykorzystuje się pokera

SCRUM POKER

ZASADY

- Zadań nie estymujemy w jednostkach czasu.
- Całym zespołem wybieramy jedno zadanie, które jest najłatwiejsze ze wszystkich innych. Dajemy mu wartość 1.
- Używamy tylko wartości z ciągu Fibonacciego (1, 2, 3, 5, 8, 13, 21, ♾, ? - nie mam pojęcia, ☕). Dzięki temu bardziej skupiamy się na rzędach wielkości niż samych jednostkach, bo pomiędzy 9 a 10 dużej różnicy nie ma.
- Jednostką są Story Points (SP). Jest to zupełnie abstrakcyjna miara. Możecie o niej myśleć jak o dowolnej innej abstrakcyjnej. Np. to zadanie wyceniam na 5 kasztanów. Informacja zupełnie nieprzydatna, ale pozwala porównywać zadania. Skoro tamto jest warte 8 to jest bardziej wartościowe (trudniejsze).
- Dla jednej osoby 1 SP może oznaczać 1 godzinę pracy a dla innej 1 dzień.

PRZEBIEG ESTYMACJI

- Całym zespołem omawiamy sobie jedno wybrane zadanie, aby mieć pewność, że wszyscy rozumieją je tak samo.
- Gdy wszyscy są gotowi _jednocześnie_ wyciągają kartę ze swoją estymatą, aby nie sugerować się innymi
- Jeśli rozbieżności są duże, to znaczy, że członkowie zespołu mają różne zrozumienie zadania i trzeba jeszcze o nim podyskutować. Warto zapytać osoby z najmniejszą i największą estymatą dlaczego tyle dają.
- Jeśli rozbieżności są małe to można je uśrednić lub wspólnie zgodzić się na którąś wartość (zazwyczaj wyższą)

Poszukajcie narzędzi typu Scrum Poker online i użyjcie do estymowania zadań na Planningu.

ODPOWIEDZIALNOŚĆ ZBIOROWA

Ciężko winić kogoś, za to, że zachoruje. Ale można winić za to, że się do czegoś zobowiązał, a nie dał znać że to się nie uda. Dlatego ważne jest daily, aby mieć nawyk aktualizowania statusu co najmniej raz dziennie. Oczywiście można częściej.

Gdy tylko dowiecie się, że ktoś z załogi wam niedomaga, to powinniście od razu pomyśleć co z tym można zrobić. Jeśli były rozpoczęte prace to należy wkomitować / przekazać to co się do tej pory udało zrobić. To kolejny powód dla którego małe, a częste commity są dobrą praktyką. Każdy ma dostęp do bieżącej wersji kodu i łatwo wtedy przekazać pracę.

Dla nas, obserwatorów z zewnątrz nigdy nie możemy ocenić ile kto z zespołu zrobił. Dla nas po prostu zrobił to zespół. Jeżeli czujecie, że robicie za dużo / za mało i coś jest niesprawiedliwe, to możecie napisać nam o waszych problemach i w razie czego zmienić zespół.

DOBRE PRAKTYKI WSPÓŁPRACY ZESPOŁOWEJ

- Pair programming
- Mob programming
- Peer code review
- Jak najszybsze dostarczanie zadań. Lepiej dostarczyć 4 na 8 zadań pracując po 2 osoby nad każdym niż 0/8 mając jednocześnie wszystkie rozpoczęte.
- Brak odgórnych przypisań osób do zadań. Osoba która właśnie ma czas bierze pierwsze z góry zadanie (lub jedno z pierwszych, jeśli nie są zablokowane)
- 1 osoba może maksymalnie mieć przypisane tylko 1 zadanie w danej chwili. Dopiero po zakończeniu poprzedniego zadania można wziąć kolejne.

AGILE MANIFESTO

Odkrywamy nowe metody programowania dzięki praktyce w programowaniu i wspieraniu w nim innych. W wyniku naszej pracy, zaczęliśmy bardziej cenić:

Ludzi i interakcje *od procesów i narzędzi*

Działające oprogramowanie *od szczegółowej dokumentacji*

Współpracę z klientem *od negocjacji umów*

Reagowanie na zmiany *od realizacji założonego planu.*

Oznacza to, że elementy wypisane po prawej są wartościowe, ale większą wartość mają dla nas te, które wypisano po lewej.

Q&A

KONWENCJE

CODERS SCHOOL

DŁUGOŚĆ LINII

Max 120 znaków

DŁUGOŚĆ FUNKCJI

Do 10 linii (+/- 5).

Jeśli jest więcej to trzeba wydzielić funkcjonalności do mniejszych funkcji.

KAŻDY BLOK (SCOPE) = WCIĘCIE

Każdy zakres rozpoczynający się od nawiasu `{` - nawet jeśli go nie ma np. przy jednolinijkowych `if`, `for` - musi mieć dodatkowy poziom wcięcia.

```cpp
int addEven(const std::vector<int>& numbers)
{
    int sum{};
    for (const auto& el: numbers)
    {
        if(el % 2 == 0)
        sum = sum + el;      // bad, no additional indentation
    }
    return sum;
 }
```

```cpp
int addEven(const std::vector<int>& numbers)
{
    int sum{};
    for (const auto& el: numbers)
    {
        if(el % 2 == 0)
        {
            sum = sum + el;      // indentation ok, braces added
        }
    }
    return sum;
 }
```

WYJĄTKI - `switch/case`

Często przy instrukcji `switch/case` spotkacie się z tym, że `case` jest na tym samym poziomie co `switch`. Nie uznajemy tego za błąd.

```
switch (value) {
case 1: doSth();
        break;
default: doSthElse();
}
```

WYJĄTKI CD.

- namespace
- modyfikatory dostępu public, protected, private

```
namespace shm {

class Ship {
public:
    Ship() = default;
    // ...

private:
    size_t maxCrew;
    // ...
}

} // namespace shm
```

TYPY KONWENCJI NAZEWNICZYCH

- lowerCamelCase
- UpperCamelCase (PascalCase)
- snake_case (konwencja Pythona)
- kebab-case (rzadko spotykana w C++, często we front-endzie)

NOTACJA WĘGIERSKA - ZŁA PRAKTYKA

W notacji węgierskiej informacja o typie zawiera się w nazwie zmiennej. Poczytaj na Wiki i nigdy nie stosuj

- ~~`iNumber`~~
- ~~`szName`~~
- ~~`lpcszText`~~

KONWENCJE DLA PÓL KLASY

- postfix *underscore* _ za nazwą
- prefix `m_` przed nazwą

Nie wolno stosować prefixów _. Każda nazwa zaczynająca się od *underscore* jest zarezerwowana dla kompilatora.

stackoverflow.com

NAZWY ZMIENNYCH Z PREFIXAMI - RACZEJ ZŁA PRAKTYKA

- zmienna lokalna `l_variable`
- parametr funkcji `p_variable`
- pole klasy `m_variable`
- zmienne globalne `g_variable`
- zmienne statyczne `s_variable`

To może wydawać się fajne, ale niepotrzebnie dodaje 2 znaki do nazwy zmiennej. Zamiast tego lepiej stosować to co na kolejnym slajdzie.

UNIKANIE PREFIXÓW

Uwaga, to są moje opinie na podstawie moich doświadczeń.

- zmienna lokalna: `variable`
  - Funkcje/bloki kodu mają być małe (do 10 linii). Wtedy bez trudu znajdujemy zmienne lokalne kilka linijek wyżej
- parametr funkcji: `variable`
  - Dla małych funkcji parametry również znajdujemy parę linijek wyżej bez scrollowania
- pole klasy: `variable_`
  - Zmienne klasy siedzą w pliku nagłówkowym i musielibyśmy się przełączyć na inny plik, aby upewnić się, że to pole klasy. Tutaj konwencją jest postfix _
- zmienne globalne: `VARIABLE`
  - Zmienne/stałe pisane dużymi literami są globalne. Są argumenty, aby też tego nie robić, ale ich nie pamiętam.
- zmienne statyczne: `s_variable`
  - Tutaj prefix `s_` może zostać, bo inne sposoby są już zajęte :)

CO JAK NAZYWAĆ?

Wszystko to kwestia przyjętej konwencji i może różnić się między projektami. Najważniejsze, to aby w całym projekcie było jednolicie. My przyjmujemy następujące założenia.

KLASY I STRUKTURY

- UpperCamelCase
- rzeczownik + przymiotniki (opcjonalnie)

```
class SuperWarrior {};  // ok
class listenMusic {};   // bad, verb instead of noun. Maybe MusicPlayer?
```

ZMIENNE

- lowerCamelCase
- rzeczownik(i) + przymiotniki (opcjonalnie)

```
SuperWarrior mightyBarbarian;   // ok
int clickCounter;               // ok
```

FUNKCJE

- lowerCamelCase
- czasownik + przysłówki lub rzeczowniki (opcjonalnie)

```
void generateStructure();        // ok
int calculateCommonSum();        // ok
```

POLA KLASY

- jak zwykłe zmienne
- dodatkowo za nazwą *underscore* _

```
class SuperWarrior
{
    int level_ = 1;
    int strength_ = 50;
    int dexterity_ = 10;
    int mana_ = 0;
};
```

NAZWY PLIKÓW

- nazwa pliku = nazwa klasy
- UpperCamelCase / lowerCamelCase / snake_case / kebab-case

`class SuperWarrior` -> SuperWarrior.hpp / SuperWarrior.cpp

OPISOWE NAZWY ZMIENNYCH

Nie nazywajmy zmiennych tak:

```
int a = 0;           // what is a?
bool b = false;      // what is b?

bool compare(int a, int b);   // what does a and b represent?
```

Gdy po miesiącu spojrzysz na ten kod też nie będziesz wiedzieć co on robi. Tak jest znacznie lepiej:

```
int counter = 0;
bool isValid = false;

bool compare(int lhs, int rhs);
```

DOZWOLONE KRÓTKIE NAZWY

- `i`, `j` - jako indeksy w pętlach
- `it` - jako skrót od `iterator`
- `el`, `elem` - jako skrót od `element` w pętlach po kolekcji
- `lhs`, `rhs` - jako skrót od `leftHandSide` i `rightHandSide` w funkcjach porównujących

RODZAJE NAWIASÓW

EGIPSKIE (K&R, STROUSTRUP)

```
while (x == y) {
    // do sth;
}
```

ALLMAN

```
while (x == y)
{
    // do sth;
}
```

Zobacz inne na Wiki i nigdy nie stosuj

MOJE ULUBIONE FORMATOWANIE

Uwaga, to jest opinia! Dla funkcji - Allman. Dla pętli, if - egipskie. Dla klas - whatever. Powód:

```
class SuperWarrior
{
public:
    SuperWarrior(int level, int strength, int dexterity)
        : level_(level)
        , strength_(strength)
        , dexterity(dexterity)
    {
        if (level_ >= 10) {
            mana_ = 50;
        }
    }

private:
    int level_ = 1;
    int strength_ = 50;
    int dexterity_ = 10;
    int mana_ = 0;
};
```

Listy inicjalizacyjne konstruktorów lub zbyt długie nazwy i typy parametrów funkcji trochę psują czytelność bloków kodu.

Dla kontrastu:

```
class SuperWarrior {
public:
    SuperWarrior(int level, int strength, int dexterity)
            : level_(level)
            , strength_(strength)
            , dexterity(dexterity) {
        if (level_ >= 10) {
            mana_ = 50;
        }
    }

private:
    int level_ = 1;
    int strength_ = 50;
    int dexterity_ = 10;
    int mana_ = 0;
};
```

DOKLEJANIE & I *

- left `int& ref`
  - ok :)
- center `int & ref`
  - osobiście to lubię
- right `int &ref`
  - unikać

OPERATOR TRÓJARGUMENTOWY `?:`

Zapis całości w jednej linii zazwyczaj jest nieczytelny

```
lhs.size() == rhs.size() ? lhs < rhs : lhs.size() < rhs.size()
```

Lepiej zapisać go w dwóch liniach.

```
lhs.size() == rhs.size() ? lhs < rhs
                         : lhs.size() < rhs.size()
```

A czasami nawet 3, jeśli warunek jest długi.

```
lhs.name == rhs.name && lhs.amount == rhs.amount && lhs.price == rhs.price
    ? lhs.value < rhs.value
    : lhs.price < rhs.price
```

Chociaż IMO lepiej już w ostatnim przypadku używać zwykłego `if`.

AUTOFORMATOWANIE

`clang-format`

W projekcie powinien być wkomitowany plik `.clang-format`, który zapewni, że każdy może zastosować opisane w nim formatowanie automatycznie

```
clang-format -style=.clang-format -i *.cpp *.hpp
```

Q&A

PRAKTYKI PROGRAMISTYCZNE

CODERS SCHOOL

ZNANE I LUBIANE PRAKTYKI

- Fail-fast
- Scout rule
- DRY
- DRTW
- KISS
- YAGNI
- RTFM
- No Tests - Don't Touch
- CQRS
- 90-90 rule
- Zen of Python

Więcej na Wikipedii

TO OMÓWIMY NA DOBRYCH PRAKTYKACH #2

- SOLID
- GRASP
- STUPID

FAIL-FAST

Jak najszybciej ubijaj program w przypadku problemów. Lepsze to niż działanie i wykrzaczenie się go z powodu Undefined Behavior później. Powodzenia w szukaniu przyczyny w takim przypadku.

DOZWOLONE RZECZY

- nieobsłużone wyjątki
- asercje (`assert`, `static_assert`)
- w funkcjach sprawdzanie koniecznych warunków na początku

```
void process(std::shared_ptr<int> value) {
    if (!value) {
        return;
    }
    // process normally
}
```

PLUSY

- minimalizowanie UB (Undefined Behavior)
- łatwiejsze debugowanie i szukanie przyczyn problemów

SCOUT RULE - ZASADA SKAUTA

> *Zawsze zostawiaj obozowisko w co najmniej takim stanie w jakim je zastałeś.*

Z kodem tak samo jak z biwakiem. Jeśli nabrudzisz w kodzie - musisz to posprzątać. Bardzo mile widziane jest też posprzątanie po innych.

Jeśli dotkniesz czyjegoś kodu, np. tylko dodając jedną linijkę - spróbuj go ulepszyć.

Sprzątanie ma być w osobnym commicie.

PLUSY

- kod może być tylko coraz czytelniejszy
- nie robimy długu technicznego, który kiedyś trzeba będzie spałcić

DRY

DON'T REPEAT YOURSELF

Często powielenia powstają wskutek prostej operacji "kopipasty".

Kod, który jest taki sam lub podobny w więcej niż 1 miejscu należy wydzielić do funkcji i wywoływać tę funkcję w tym miejscu

PLUSY

- łatwiejszy refaktoring - zmiana tylko w jednym miejscu
- mniej miejsca na błędy. Nie ma ryzyka, że po odkryciu błędu nie zmienimy wszystkich wystąpień

DRTW

DON'T REINVENT THE WHEEL

Nie wynajduj koła na nowa i używaj gotowych bibliotek dostarczających daną funkcjonalność.

PLUSY

- zazwyczaj używasz już przetestowanego kodu
- mniej miejsca na błędne implementacje - możesz nie wymyślić wszystkich przypadków testowych
- oszczędność czasu

KISS / BUZI

KEEP IT SIMPLE, STUPID

BEZ UDZIWNIEŃ ZAPISU, IDIOTO

Pisząc kod, nie wciskamy na siłę tricków i nowości, które może i uczynią go bardziej uniwersalnym i łatwym do przyszłego rozwijania, ale zmniejszą jego czytelność.

Wielu seniorów ma z tym problem i niepotrzebnie komplikuje kod. Na podstawie doświadczeń wiedzą, że dzięki temu kod będzie łatwiej modyfikować w przyszłości. A co w sytuacji, jeśli nie będzie on później nigdy modyfikowany, a będzie jednak czytany wiele razy?

Każde "usprawnienie" wprowadzamy dopiero wtedy, gdy zachodzi taka potrzeba. Nic nie piszemy "na zapas". Patrz - YAGNI.

PLUSY

- kod łatwiejszy do zrozumienia, w szczególności dla juniorów

YAGNI

YOU AREN'T GONNA NEED IT

Nie będziesz tego potrzebować. Nie piszemy kod na zapas, myśląc, że to się wkrótce przyda. Generujemy w ten sposób martwy kod, który nigdy nie jest i raczej nie będzie używany.

Jeśli w przyszłości dana funkcjonalność będzie potrzebna to pewnie zaimplementuje ją inna osoba (nawet Ty po miesiącu zapomnisz, że coś już zdarzyło Ci się napisać) i napisze(sz) to ponownie.

PLUSY

Im mniej kodu tym lepiej:

- krótszy czas kompilacji
- mniej skomplikowane zależności
- mniej do analizowania
- mniej do pamiętania (że coś było zrobione na zapas)
- szybsze użycie narzędzie (`grep` lub `Ctrl+F` mają mniej do skanowania)

RTFM

READ THE F*CKING MANUAL ;)

Zanim zasypiesz ludzi pytaniami o daną funkcjonalność / narzędzie przeczytaj instrukcję.

W Linuxie dla narzędzia `tool` używaj `tool --help` oraz `man tool`.

Jeśli po lekturze dokumentacji nadal nie wiesz co robić, wtedy zapytaj innych (osoby z zespołu, stackoverflow, inne forum, Discord Coders School 😉)

PLUSY

- Bardziej doświadczone osoby często mają napięty kalendarz, więc będą wdzięczne jeśli znajdziesz rozwiązanie samodzielnie lub bardziej się streścisz ;)

NO TESTS - DON'T TOUCH

Jeśli chcesz poprawić jakiś kawałek kodu, ale nie jest on przetestowany, to najpierw napisz testy.

Jeśli "ulepszysz", czy też "poprawisz" kod, do którego nie było testów, to jak udowodnisz, że po Twoich zmianach jego zachowanie się nie zmieniło?

Nigdy nie zmieniaj jednocześnie implementacji oraz testów do niej. Jeśli testy przestaną przechodzić, to nie będziesz wiedzieć, czy to z powodu zepsutej implementacji czy zepsutych testów. Testy testują implementację, a implementacja testuje testy.

PLUSY

- oszczędzanie sobie potencjalnych problemów i czasu
- napisanie testów i refaktoring zajmie mniej czasu niż refaktoring bez testów (serio)

CQRS

COMMAND QUERY RESPONSIBILITY SEGREGATION

Oddziel zapis od odczytu.

- **Query** - pobranie (odczyt) danych - gettery
- **Command** - działanie na danych (zapis)

Powinniśmy unikać metod, które robią jedno i drugie razem, ponieważ:

> *Asking a question should not change the answer*
>
> *-- Bertrand Meyer*

ZASADA 90-90

90% zadania wykonasz w ciągu 90% czasu na nie przeznaczone.

Pozostałe 10% zadania wykonasz przez pozostałe 90% czasu.

Przydatna przy ocenie postępów prac.

Na początku prace idą całkiem żwawo (ogólny zarys zadania), ale dopiero w trakcie odkrywane są nie przemyślane wcześniej przypadki brzegowe, przez które zadanie zajmie więcej czasu się spodziewano.

ZEN OF PYTHON

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Q&A

DOBRE PRAKTYKI

CODERS SCHOOL

ZAGADKA

Tomek dostał 3 jabłka, ale 2 zjadł. Ile jabłek ma Tomek?

ODPOWIEDŹ

Nie wiadomo, bo nie wiemy ile miał wcześniej zanim dostał 3 jabłka.

MORAŁ

Zawsze inicjalizuj zmienne.

```
int a;
a += 3;
a -= 2;
std::cout << a << '\n';
```

ZAWSZE INICJALIZUJ ZMIENNE

```
int index = 0;                  // ok
bool completed;                 // bad
bool finished = false;          // ok

if (completed) {                // very bad
    // ...
}

struct Node {
    Node* next = nullptr;       // ok
    int value = 0;
}
```

DEFINIUJ WŁASNE TYPY

Kod pełny nazw domenowych (własnych typów) lepiej się czyta. Także osoby nie zajmujące się programowaniem na co dzień potrafią dużo na jego podstawie wydedukować.

Zobacz te 2 funkcje:

```
std::vector<std::pair<uint8_t, uint8_t>> compressGrayscale(std::array<std::arr
std::array<std::array<uint8_t, width>, height> decompressGrayscale(std::vector
```

I zobacz je teraz:

```
CompressedImage compressGrayscale(const Image& bitmap);
Image decompressGrayscale(const CompressedImage& compression);
```

Aby to osiągnąć wystarczy nadać inne nazwy typom w taki sposób:

```
using Image = std::array<std::array<uint8_t, width>, height>;
using CompressedImage = std::vector<std::pair<uint8_t, uint8_t>>;
```

FORWARD DECLARATIONS

Używaj deklaracji zapowiadających tam gdzie to możliwe zamiast `#include`.

Kompilator, aby używać jakiegoś obiektu potrzebuje informacji o jego rozmiarze.

```
#include "Fruit.hpp"

int main() {
    Fruit apple;
    // ...
}
```

Aby można było utworzyć zmienną lokalną potrzebuje więc znać jej rozmiar. Musi więc mieć odpowiedni nagłówek, w którym ta klasa jest zdefiniowana, aby znać rozmiar takiej zmiennej na stosie.

ZAGADKA

- Jaki rozmiar ma wskaźnik na `int`?
- Jaki rozmiar ma wskaźnik na `Fruit`?

ODPOWIEDŹ

Rozmiar wskaźnika nie zależy od typu na który wskazuje. Zazwyczaj ma on 4 lub 8 bajtów (odpowiednio dla procesorów 32 i 64 bitowych).

NIEZNANY ROZMIAR OBIEKTU

Każdy wskaźnik oraz referencja mają ten sam rozmiar. Dopóki się nimi posługujemy i nie odwołujemy się do żadnych metod lub pól klasy, to informacja o jej rozmiarze lub zawartości nie jest nam potrzebna.

```
class Fruit;     // class forward declaration

void pass(Fruit* fruit) {
    if (!fruit) {
        return;
    }
    // do sth;
}
```

W takim przypadku wystarczy, że powiemy kompilatorowi, że typ wskaźnika jest naszą klasą za pomocą deklaracji zapowiadającej (forward declaration). Nie trzeba wtedy stosować `#include` i przyspieszamy dzięki temu kompilację.

Jeśli jednak odwołamy się do jakiegoś pola lub metody tego obiektu to kompilator powie nam, że ma niepełne informacje.

```
class Fruit;

unsigned getFruitPrice(Fruit* fruit) {
    if (!fruit) {
        return 0u;
    }
    return fruit->getPrice();  // compilation error
}
```

W takim przypadku potrzeba dołączyć właściwy nagłówek.

UNIKAMY NIEJAWNYCH KONWERSJI

```
class Apple {
    int weight_;
public:
    Apple(int weight);  // possible conversion from int to Apple
};

void takeApple(Apple a);

int main() {
    takeApple(150);     // int converted implicitly into Apple
}
```

ZAPOBIEGANIE NIEJAWNYM KONWERSJOM

Słowo kluczowe `explicit` zabrania niejawnych konwersji. Stosuje się je przy konstruktorach, które mogą przypadkiem spowodować konwersję z typu argumentu który przyjmują.

```
class Apple {
    int weight_;
public:
    explicit Apple(int weight);
};

void takeApple(Apple a);

int main() {
    takeApple(150);     // compilation error, expected Apple, got int
}
```

`explicit`

Słowo kluczowe `explicit` stosujemy przy:

- konstruktorach jednoargumentowych
  - `explicit Apple(int weight);`
- konstruktorach wieloargumentowych, z jednym nie-domyślnym argumentem
  - `explicit Apple(int weight, int size = 1);`
- operatorach konwersji
  - `explicit operator int() const { return weight_; }`

CZYTANIE I STOSOWANIE CPPCOREGUIDELINES

F: FUNCTIONS

- F.1: "Package" meaningful operations as carefully named functions
- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- F.4: If a function may have to be evaluated at compile time, declare it constexpr
- F.5: If a function is very small and time-critical, declare it inline
- F.6: If your function may not throw, declare it noexcept
- F.9: Unused parameters should be unnamed
- F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to const
- F.17: For "in-out" parameters, pass by reference to non-const
- F.20: For "out" output values, prefer return values to output parameters
- F.21: To return multiple "out" values, prefer returning a struct or tuple
- F.60: Prefer T* over T& when "no argument" is a valid option
- F.26: Use a unique_ptr to transfer ownership where a pointer is needed
- F.27: Use a shared_ptr to share ownership

C: CLASSES AND CLASS HIERARCHIES

- C.1: Organize related data into structures (structs or classes)
- C.2: Use class if the class has an invariant; use struct if the data members can vary independently
- C.7: Don't define a class or enum and declare a variable of its type in the same statement
- C.8: Use class rather than struct if any member is non-public
- C.9: Minimize exposure of members

ENUM: ENUMERATIONS

- Enum.1: Prefer enumerations over macros
- Enum.2: Use enumerations to represent sets of related named constants
- Enum.3: Prefer enum classes over "plain" enums
- Enum.4: Define operations on enumerations for safe and simple use
- Enum.5: Don't use ALL_CAPS for enumerators
- Enum.6: Avoid unnamed enumerations
- Enum.7: Specify the underlying type of an enumeration only when necessary
- Enum.8: Specify enumerator values only when necessary

Q&A

PODSTAWY C++

PODSUMOWANIE

CODERS SCHOOL

CO PAMIĘTASZ Z DZISIAJ?

NAPISZ NA CZACIE JAK NAJWIĘCEJ HASEŁ

1. Współpraca zespołowa
2. Konwencje
3. Cpp Core Guidelines
4. Dobre praktyki

PRACA DOMOWA

PRE-WORK

- Przypomnij sobie jak zbudować projekt w trybie Debug
- Obejrzyj to wideo o debugowaniu
- Upewnij się, że masz zainstalowany debugger `gdb`
- Pobaw się debuggerem gdb lub onlinegdb.com

POST-WORK

- Zastosuj poznaną wiedzę we wszystkich dotychczasowych projektach i pracach domowych. Nie musisz zgłaszać ponownie PR, ale poproś inne osoby o review, jeśli masz jakieś wątpliwości.
- Od teraz robiąc review innym osobom zwracaj uwagę na przedstawione dobre praktyki :)
- Z nudów powróć do zadań na Firecode.io :)
- Przejrzyj nagłówki z CppCoreGuidelines, a w wolnych chwilach czytaj ile się da :)

CODERS SCHOOL