

Podstawy OOP w C++ - poziom 5-6

Plan przygotowania na 9 tygodni

Cel materiału

Przygotowanie do pierwszych 2-3 miesięcy nauki na poziomie ocen 5-6. Każdy tydzień zawiera lekcję, ćwiczenie, zadanie rozszerzające i odpowiedź.

Podstawa zakresu

Dokument źródłowy nie zawiera osobnej tabeli OOP. Ten zakres jest świadomie oznaczonym rozszerzeniem: zachowuje standard samodzielności, poprawności, doboru struktur i analizy wymagany w dokumencie dla ocen 5-6.

Jak pracować

- 2 spotkania po 45-60 minut w tygodniu
- najpierw rozwiąż zadanie bez odpowiedzi, potem popraw własne rozwiązanie
- prowadź zeszyt prób śladowych, błędów i wniosków
- po każdym module zalicz krótki quiz i zapisz wynik

Rozdzielenie wymagań: ocena 5 i 6

Ocena 5 - samodzielne i poprawne zastosowanie	Ocena 6 - dobór, uzasadnienie i rozszerzenie
• samodzielne projektowanie klas z polami prywatnymi i publicznym interfejsem	• uzasadnianie odpowiedzialności klas i relacji między nimi
• konstruktory, metody, const, kompozycja i kolekcje obiektów	• świadome zarządzanie cyklem życia zgodnie z RAII i użycie smart pointers
• proste dziedziczenie i polimorfizm, podział programu na pliki	• interfejsy abstrakcyjne, wirtualny destruktor i polimorfizm bez wycieków
• testowanie niezmienników i obsługa sytuacji błędnych	• porównanie kompozycji z dziedziczeniem oraz ocena kosztu operacji
	• samodzielny projekt wieloplikowy z testami i analizą decyzji projektowych

Mapa pracy

Tydzień 1: Klasa, obiekt i odpowiedzialność [zakres 5]

Tydzień 2: Hermetyzacja i niezmienniki [zakres 5-6]

Tydzień 3: Konstruktory, destruktory i RAII [zakres 5-6]

Tydzień 4: Metody, const, static i przeciążanie [zakres 5]

Tydzień 5: Kompozycja i relacje między klasami [zakres 5-6]

Tydzień 6: Dziedziczenie i przesłanianie [zakres 5]

Tydzień 7: Polimorfizm i klasy abstrakcyjne [zakres 5-6]

Tydzień 8: Kolekcje obiektów, algorytmy STL i koszt [zakres 5-6]

Tydzień 9: Projekt wieloplikowy i testowanie [zakres 5-6]

Standard języka i sposób rozumienia OOP

Przyjęta wersja języka

Materiał przyjmuje C++23, czyli aktualnie opublikowany standard ISO/IEC 14882:2024. Przykłady korzystają z praktyk Modern C++ obecnych od C++11 i rozwijanych w C++17/20/23: inicjalizacji klamrowej, RAII, `std::string`, `std::vector`, `std::unique_ptr`, `make_unique`, `override`, `lambda` i algorytmów STL. Powstający C++26 nie jest traktowany jako wymagany standard kursu.

Ważne rozróżnienia

- Norma ISO definiuje język i bibliotekę standardową. C++ Core Guidelines opisują zalecane praktyki korzystania z języka; nie są częścią normy.
- Obiekt w C++ to nie tylko instancja klasy. Także `int`, tablica i wskaźnik są obiektami w znaczeniu modelu języka.
- OOP nie oznacza umieszczania całego kodu w klasach. C++ łączy programowanie obiektowe, proceduralne, generyczne i funkcyjne.
- Nowoczesny C++ preferuje obiekty automatyczne, kontenery biblioteki standardowej, RAII i jawny model własności zamiast ręcznych `new/delete`.

Źródła terminologii

- ISO/IEC 14882:2024 - Programming languages - C++ (aktualny opublikowany standard C++23)
- Standard C++ Foundation: The Standard - <https://isocpp.org/std/the-standard>
- C++ Core Guidelines, B. Stroustrup i H. Sutter - <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- WG21, finalny projekt C++23 N4950 - <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf>

Słownik pojęć Modern C++

Definicje są szersze niż skrótowe określenia szkolne. Przy każdym hasle znajduje się krótki przykład oraz uwaga praktyczna.

Obiekt

Obszar pamięci posiadający typ, czas życia i reprezentujący wartość. Obiektem jest zarówno instancja klasy Konto, jak i zmienna int, tablica czy wskaźnik.

```
Konto k; int licznik = 0;
```

Uwaga: W języku potocznym OOP słowo obiekt często zawęża się do instancji klasy, ale standard C++ używa szerszego znaczenia.

Klasa

Typ definiowany przez użytkownika, który może grupować składowe danych, funkcje składowe, typy zagnieżdżone i reguły dostępu. Definicja klasy opisuje reprezentację oraz interfejs obiektów tego typu.

```
class Konto { double saldo_; public: double saldo() const; };
```

Uwaga: Sama klasa nie zajmuje pamięci na pola konkretnego obiektu; pamięć pojawia się przy utworzeniu obiektu.

Instancja

Konkretny obiekt utworzony zgodnie z definicją danego typu klasy.

```
Konto firmowe;
```

Uwaga: firmowe jest instancją klasy Konto.

Stan obiektu

Zbiór wartości przechowywanych przez obiekt w danym momencie, zwykle reprezentowany przez jego pola.

```
double saldo_ = 0.0;
```

Uwaga: Stan powinien być zmieniany wyłącznie przez operacje zachowujące reguły klasy.

Zachowanie

Operacje udostępniane przez typ, zwykle jako funkcje składowe. Zachowanie mówi, co można zrobić z obiektem.

```
bool wypłac(double kwota);
```

Uwaga: Dobra metoda modeluje operację domenową, a nie tylko techniczny setter pola.

Interfejs klasy

Publiczny kontrakt typu: zestaw dostępnych operacji, ich parametrów, wyników, warunków wstępnych i gwarantowanych efektów.

```
double saldo() const noexcept;
```

Uwaga: Użytkownik klasy powinien móc korzystać z niej bez znajomości prywatnej reprezentacji.

Hermetyzacja

Łączenie danych z operacjami i kontrolowanie dostępu do reprezentacji. Celem jest ochrona reguł typu i zmniejszenie zależności między częściami programu.

```
private: double saldo_;
```

Uwaga: Hermetyzacja nie oznacza automatycznego dodania settera do każdego pola.

Niezmiennik klasy

Warunek dotyczący stanu, który musi być prawdziwy po zakończeniu konstruktora i po każdej publicznej operacji obiektu.

```
// saldo_ >= 0
```

Uwaga: Konstruktor i metody modyfikujące wspólnie odpowiadają za zachowanie niezmiennika.

Konstruktor

Specjalna funkcja składowa rozpoczynająca czas życia obiektu klasy i ustanawiająca jego poprawny stan.

```
Konto(double saldo) : saldo_(sprawdz(saldo)) {}
```

Uwaga: Pola są inicjalizowane przed wejściem do ciała konstruktora.

Destruktor

Specjalna funkcja składowa wykonywana przy końcu czasu życia obiektu. Kończy życie składowych w odwrotnej kolejności ich konstrukcji.

```
~Figura() = default;
```

Uwaga: Nie należy ręcznie czyścić stringów i wektorów - ich destruktory robią to poprawnie.

Czas życia

Okres od rozpoczęcia istnienia obiektu po poprawnej inicjalizacji do jego zniszczenia. Nie jest tym samym co zakres widoczności nazwy.

```
{ Konto k; } // tutaj kończy się życie k
```

Uwaga: Wskaźnik do obiektu po końcu jego życia staje się wiszący.

RAII

Technika Resource Acquisition Is Initialization: własność zasobu jest związana z czasem życia obiektu, a zwolnienie następuje automatycznie w destruktorze.

```
std::lock_guard lock(mutex);
```

Uwaga: RAII dotyczy pamięci, plików, blokad, gniazd i każdego zasobu wymagającego pary pozyskaj/zwolnij.

Własność zasobu

Odpowiedzialność za zakończenie czasu życia zasobu lub jego zwolnienie. W programie powinno być jasne, który obiekt jest właścicielem.

```
std::unique_ptr<Foo> p = std::make_unique<Foo>();
```

Uwaga: Surowy wskaźnik T* zwykle traktujemy jako nieposiadający obserwatora, nie jako właściciela.

Reguła zera

Jeżeli składowe klasy same poprawnie zarządzają zasobami, klasa nie powinna definiować własnego destruktora ani operacji kopiowania i przenoszenia.

```
class Raport { std::string nazwa_; std::vector<int> dane_; };
```

Uwaga: To preferowana reguła w kodzie aplikacyjnym.

Kopiowanie

Utworzenie nowego, niezależnego obiektu o tej samej wartości albo przypisanie wartości istniejącemu obiektowi.

```
Raport b = a;
```

Uwaga: Standardowe kontenery wykonują głębokie kopiowanie swoich elementów.

Przenoszenie

Przekazanie zasobów z jednego obiektu do drugiego bez kosztownego kopiowania. Obiekt źródłowy pozostaje poprawny, ale jego wartość jest nieokreślona w dozwolonym zakresie typu.

```
Raport b = std::move(a);
```

Uwaga: Po przeniesieniu wolno obiekt zniszczyć lub przypisać mu nową wartość; nie zakładaj jego wcześniejszej zawartości.

Kompozycja

Relacja 'ma', w której obiekt zawiera inne obiekty jako składowe i zwykle kontroluje ich czas życia.

```
class Samochod { Silnik silnik_; };
```

Uwaga: Kompozycja jest preferowana, gdy nie zachodzi prawdziwa relacja podstawienia 'jest'.

Agregacja

Słabsza relacja całość-część, w której część może istnieć niezależnie i nie jest własnością obiektu agregującego.

```
class Druzyna { std::vector<std::reference_wrapper<Gracz>> gracze_; };
```

Uwaga: Trzeba zapewnić, że obserwowane obiekty żyją wystarczająco długo.

Dziedziczenie publiczne

Mechanizm tworzenia typu pochodnego, który zachowuje publiczny kontrakt bazy i powinien być używalny wszędzie tam, gdzie oczekiwana jest baza.

```
class Koło final : public Figura { ... };
```

Uwaga: Dziedziczenie nie jest tylko sposobem współdzielenia kodu.

Polimorfizm dynamiczny

Wybór implementacji funkcji virtual w czasie wykonania na podstawie dynamicznego typu obiektu.

```
Figura& f = koło; f.rysuj();
```

Uwaga: Działa przez referencję lub wskaźnik; przekazanie przez wartość może spowodować slicing.

Typ statyczny i dynamiczny

Typ statyczny wynika z deklaracji i jest znany kompilatorowi; typ dynamiczny opisuje rzeczywisty najbardziej pochodny obiekt używany przez wyrażenie polimorficzne.

```
Figura& f = koło; // statyczny Figura, dynamiczny Koło
```

Uwaga: virtual wybiera funkcję według typu dynamicznego.

Funkcja wirtualna

Niestatyczna funkcja składowa zadeklarowana virtual, której końcowe przesłonięcie jest wybierane dynamicznie.

```
virtual double pole() const = 0;
```

Uwaga: Dodawaj override w klasie pochodnej, aby kompilator sprawdził sygnaturę.

Klasa abstrakcyjna

Klasa mająca co najmniej jedną nieprzesłoniętą funkcję czysto wirtualną; nie można tworzyć jej bezpośrednich obiektów.

```
class Figura { virtual double pole() const = 0; };
```

Uwaga: Często reprezentuje interfejs dla rodziny typów.

Wirtualny destruktor

Destruktor umożliwiający poprawne zniszczenie kompletnego obiektu pochodnego przez wskaźnik lub referencję do klasy bazowej.

```
virtual ~Figura() = default;
```

Uwaga: Publiczny destruktor polimorficznej bazy powinien być wirtualny.

Slicing

Utrata części pochodnej podczas skopiowania obiektu klasy pochodnej do obiektu klasy bazowej przez wartość.

```
Figura f = koło; // zła idea dla hierarchii polimorficznej
```

Uwaga: Polimorficzne obiekty przekazuj przez referencję lub wskaźnik.

Przeciążanie

Zdefiniowanie kilku funkcji o tej samej nazwie, ale różnych listach parametrów, rozstrzyganych w czasie kompilacji.

```
drukuj(int); drukuj(std::string_view);
```

Uwaga: Nie jest tym samym co przesłanianie funkcji virtual.

Przesłanianie

Dostarczenie w klasie pochodnej funkcji odpowiadającej funkcji virtual z bazy, tak aby uczestniczyła w dynamicznym wyborze.

```
double pole() const override;
```

Uwaga: override nie tworzy nowej funkcji wirtualnej - sprawdza poprawność przesłonięcia.

std::unique_ptr

Inteligentny wskaźnik reprezentujący wyłączną własność dynamicznego obiektu; jest przenoszalny, ale niekopiowalny.

```
auto p = std::make_unique<Kolo>(2.0);
```

Uwaga: To domyślny wybór, gdy naprawdę potrzebna jest dynamiczna alokacja i pojedynczy właściciel.

std::shared_ptr

Inteligentny wskaźnik współdzielący własność przez licznik odwołań; zasób jest niszczone po zniknięciu ostatniego właściciela.

```
auto p = std::make_shared<Wezel>();
```

Uwaga: Stosuj tylko przy rzeczywistej współwłasności. Cykle wymagają std::weak_ptr.

SOLID

Zbiór pięciu zasad projektowych dla kodu obiektowego. W tym kursie najważniejsze są: jedna odpowiedzialność, otwartość na rozszerzenia i poprawne podstawienie typów pochodnych.

```
class Raport { /* jedna spójna odpowiedzialność */ };
```

Uwaga: To wskazówki projektowe, nie część normy ISO C++. Nie należy stosować ich mechanicznie.

Tydzień 1. Klasa, obiekt i odpowiedzialność

Poziom

Zakres docelowy: ocena 5

Po tym module

- odróżnisz klasę od obiektu
- zaprojektujesz stan i zachowanie
- utworzysz obiekt na stosie

Definicje kluczowych pojęć

Pojęcie	Definicja
Abstrakcja	Modelowanie tylko tych cech obiektu, które są potrzebne w danym problemie. Abstrakcja określa co typ robi, a nie każdy szczegół jego reprezentacji.
Funkcja składowa	Funkcja zadeklarowana w klasie. Niestatyczna funkcja składowa działa dla konkretnego obiektu i ma dostęp do jego składowych przez niejawną <code>this</code> .
<code>this</code>	Wskaźnik dostępny w niestatycznej funkcji składowej, wskazujący obiekt, dla którego wywołano metodę.

Lekcja

Klasa jest definicją typu, a obiekt konkretną instancją. Pola opisują stan, metody - operacje dozwolone na tym stanie. Dobra klasa reprezentuje jedno spójne pojęcie.

Zanim napiszesz kod, zapisz odpowiedzialność klasy, jej niezmienniki i przykładowe użycie. Obiekt utworzony lokalnie ma automatyczny czas życia i jest niszczone po wyjściu z zakresu.

W C++ definicję klasy kończy średnik. Domyślny dostęp w `class` jest prywatny, a w `struct` publiczny.

```
class Punkt {
public:
    double x = 0;
    double y = 0;
    double odlegloscOdZera() const;
};

Punkt p{3, 4};
```

Rozwinięcie i praktyka Modern C++

C++ jest językiem wieloparadygmatowym. OOP jest jednym z dostępnych stylów, obok programowania proceduralnego, generycznego i funkcyjnego. Nie każda funkcja musi być metodą, a nie każdy problem wymaga hierarchii klas.

Projektowanie zaczynamy od odpowiedzialności: 'Punkt przechowuje współrzędne i wykonuje operacje geometryczne'. Nazwa klasy powinna określać pojęcie, a publiczne metody - sensowne operacje na nim.

Współczesny C++ preferuje typy o wartościowej semantyce: obiekt można bezpiecznie utworzyć lokalnie, przekazać i zwrócić bez ręcznego zarządzania pamięcią. Dynamiczna alokacja nie jest warunkiem użycia OOP.

Przykład: typ wartościowy reprezentujący punkt

```
class Punkt {
    double x_{};
    double y_{};
public:
    Punkt(double x, double y) : x_(x), y_(y) {}
    [[nodiscard]] double x() const noexcept { return x_; }
    [[nodiscard]] double y() const noexcept { return y_; }
    void przesun(double dx, double dy) noexcept {
        x_ += dx; y_ += dy;
    }
};

Punkt p{3.0, 4.0}; // obiekt automatyczny, bez new/delete
```

Typowa pułapka

Błąd projektowy: klasa MenedzerProgramu z dziesiątkami niepowiązanych pól i metod. Naprawa: rozdzielenie odpowiedzialności na małe typy i zwykłe funkcje.

Ćwiczenie

Zaprojektuj klasę Prostokat: wymiary oraz metody pole() i obwod(). Zapisz przykładowe użycie.

Zadanie na ocenę 6

Wyjaśnij, dlaczego klasa 'Program' zawierająca wszystkie dane i funkcje zwykle narusza zasadę jednej odpowiedzialności.

Sprawdź się

Klasa to...

- A. definicja typu
- B. zawsze jeden obiekt
- C. wyłącznie funkcja
- D. plik wykonywalny

Odpowiedzi - tydzień 1

Odpowiedź do ćwiczenia

Pola a,b; metody const zwracające $a*b$ oraz $2*(a+b)$. Na tym etapie pola mogą być publiczne, ale w następnym module zostaną ukryte.

Quiz

Poprawna odpowiedź: A. definicja typu. Obiekty są instancjami klasy.

Kryterium zaliczenia

Potrafisz odtworzyć rozwiązanie bez zaglądania do odpowiedzi, wyjaśnić każdy krok i podać co najmniej jeden test graniczny.

Tydzień 2. Hermetyzacja i niezmienniki

Poziom

Zakres docelowy: ocena 5-6

Po tym module

- ukryjesz reprezentację
- zabezpieczysz poprawny stan
- zaprojektujesz mały interfejs

Definicje kluczowych pojęć

Pojęcie	Definicja
private/public/protected	Specyfikatory dostępu. private ogranicza użycie do klasy i jej zaprzyjaźnionych elementów; public tworzy interfejs; protected udostępnia składowe także klasom pochodnym.
Warunek wstępny	Założenie, które musi być spełnione przed wywołaniem operacji, np. kwota > 0.
Warunek końcowy	Gwarancja, która ma być prawdziwa po poprawnym zakończeniu operacji.

Lekcja

Hermetyzacja ogranicza bezpośredni dostęp do pól. Publiczne metody pilnują reguł, dzięki czemu obiekt nie może łatwo znaleźć się w stanie niedozwolonym.

Niezmiennik to warunek prawdziwy po konstrukcji i po każdej publicznej operacji, np. saldo nie może spaść poniżej limitu. Setter nie jest obowiązkowy; udostępniamy tylko operacje potrzebne modelowi.

Metody tylko odczytujące stan oznaczamy const. Kompilator kontroluje wtedy, że nie zmieniają pól obiektu.

```
class Konto {
    double saldo_ = 0;
public:
    bool wplac(double kwota) {
        if (kwota <= 0) return false;
        saldo_ += kwota; return true;
    }
    double saldo() const { return saldo_; }
};
```

Rozwinięcie i praktyka Modern C++

Ukrycie pól pozwala zmienić reprezentację bez zmiany kodu użytkownika. Konto może dziś przechowywać grosze w int64_t, a jutro specjalny typ Money - o ile publiczny kontrakt pozostanie ten sam.

Getter ma sens, gdy odczyt jest częścią modelu. Setter setSaldo omija reguły biznesowe; operacje wplac i wypłac opisują prawdziwe zachowania oraz mogą odmówić zmiany.

protected często zwiększa sprzężenie z klasami pochodnymi. W interfejsach bazowych preferuj prywatną reprezentację i chronione/publiczne operacje, które zachowują niezmienniki.

Przykład: hermetyzacja wartości pieniężnej

```
class Konto {
    std::int64_t grosze_{};
public:
    explicit Konto(std::int64_t start = 0) {
        if (start < 0) throw std::invalid_argument{"saldo"};
        grosze_ = start;
    }
    [[nodiscard]] std::int64_t saldo() const noexcept { return grosze_; }
    bool wypłac(std::int64_t kwota) noexcept {
        if (kwota <= 0 || kwota > grosze_) return false;
        grosze_ -= kwota; return true;
    }
};
```

Typowa pułapka

Antywzorzec: publiczne pole saldo lub setSaldo pozwalające ustawić -100. Publiczny interfejs powinien uniemożliwiać złamanie niezmiennika.

Ćwiczenie

Dodaj do Konta wypłac(kwota), która nie dopuszcza kwoty <=0 ani salda ujemnego.

Zadanie na ocenę 6

Porównaj setSaldo(double) z operacjami wpłac/wypłac. Który interfejs lepiej modeluje domenę i dlaczego?

Sprawdź się

Hermetyzacja służy m.in. do...

- A. ochrony niezmienników
- B. przyspieszenia każdej pętli
- C. usuwania konstruktorów
- D. zamiany klasy w funkcję

Odpowiedzi - tydzień 2

Odpowiedź do ćwiczenia

Jeśli kwota $\leq$ 0 lub kwota $>$ saldo_, zwróć false; inaczej saldo_ $-=$ kwota i zwróć true. Niezmiennik saldo_ $\geq$ 0 pozostaje zachowany.

Quiz

Poprawna odpowiedź: A. ochrony niezmienników. Kontrolowany interfejs chroni poprawny stan.

Kryterium zaliczenia

Potrafisz odtworzyć rozwiązanie bez zaglądania do odpowiedzi, wyjaśnić każdy krok i podać co najmniej jeden test graniczny.

Tydzień 3. Konstruktory, destruktory i RAII

Poziom

Zakres docelowy: ocena 5-6

Po tym module

- zainicjalizujesz pola listą inicjalizacyjną
- wyjaśnisz czas życia
- poznasz zasadę RAII

Definicje kluczowych pojęć

Pojęcie	Definicja
Lista inicjalizacyjna	Część konstruktora po dwukropku, która bezpośrednio inicjalizuje klasy bazowe i pola przed wykonaniem ciała konstruktora.
Zasób	Coś, co musi zostać później zwolnione: pamięć, plik, blokada, gniazdo sieciowe lub uchwyt systemowy.
Bezpieczeństwo wyjątków	Gwarancja, że błąd zgłoszony wyjątkiem nie pozostawi wycieków ani uszkodzonego stanu.

Lekcja

Konstruktor tworzy poprawny obiekt. Pola inicjalizujemy na liście inicjalizacyjnej, ponieważ powstają przed wykonaniem ciała konstruktora.

Destruktor wykonuje się automatycznie przy końcu czasu życia. RAII wiąże zasób z obiektem: konstruktor pozyskuje zasób, destruktorem go zwalnia. Standardowe typy, np. `vector` i `string`, realizują to za nas.

Dla klas zarządzających surowym zasobem trzeba rozumieć regułę 0/3/5. Na poziomie podstawowym preferuj regułę zera: używaj gotowych typów zarządzających pamięcią.

```
class Uczeń {
    std::string imie_;
    int rocznik_;
public:
    Uczeń(std::string imie, int rocznik)
        : imie_(std::move(imie)), rocznik_(rocznik) {}
};
```

Rozwinięcie i praktyka Modern C++

Pola są inicjalizowane w kolejności deklaracji w klasie, nie w kolejności zapisanej na liście konstruktora. Dlatego deklaracje i lista powinny mieć tę samą kolejność.

RAII zapewnia deterministyczne zwalnianie zasobów również przy `return` i wyjątku. `std::vector`, `std::string`, `fstream`, `unique_ptr` i `lock_guard` są uchwytami RAII.

Reguła zera mówi: jeśli wszystkie pola poprawnie zarządzają własnym czasem życia, pozwól kompilatorowi wygenerować destruktora, kopiowanie i przenoszenie. Ręczny `new/delete` jest wyjątkiem, nie podstawową techniką.

Przykład: plik jako zasób RAI

```
void zapisz(const std::filesystem::path& sciezka,
            std::string_view tekst) {
    std::ofstream plik{sciezka};
    if (!plik) throw std::runtime_error{"Nie można otworzyć"};
    plik << tekst;
} // destruktor ofstream zamyka plik także przy wyjątku
```

Typowa pułapka

Błąd: new w konstruktorze i delete w destruktorze bez poprawnego kopiowania. Skutek: podwójne delete lub wyciek. Preferuj vector/string/unique_ptr i regułę zera.

Ćwiczenie

Napisz konstruktor Prostokata odrzucający niedodatnie wymiary przez std::invalid_argument.

Zadanie na ocenę 6

Wyjaśnij, dlaczego std::vector nie wymaga ręcznego delete i jak RAI zabezpiecza wyjątki.

Sprawdź się

RAI wiąże zasób z...

- A. czasem życia obiektu
- B. pętlą for
- C. makrem
- D. wyłącznie dziedziczeniem

Odpowiedzi - tydzień 3

Odpowiedź do ćwiczenia

Na liście ustaw `a_b_`, a w ciele sprawdź warunek; lepiej najpierw walidować funkcją pomocniczą. W prostym wariancie: `if(a<=0||b<=0) throw std::invalid_argument(...)`.

Quiz

Poprawna odpowiedź: A. czasem życia obiektu. Destruktor automatycznie zwalnia posiadany zasób.

Kryterium zaliczenia

Potrafisz odtworzyć rozwiązanie bez zaglądania do odpowiedzi, wyjaśnić każdy krok i podać co najmniej jeden test graniczny.

Tydzień 4. Metody, const, static i przeciążanie

Poziom

Zakres docelowy: ocena 5

Po tym module

- zastosujesz metody const
- rozróżnisz element obiektu i klasy
- przeciążysz operator w uzasadnionym miejscu

Definicje kluczowych pojęć

Pojęcie	Definicja
const-correctness	Konsekwentne oznaczanie obiektów, parametrów i metod, które nie powinny być modyfikowane.
Przeciążanie funkcji	Kilka funkcji o tej samej nazwie różniących się parametrami; wybór następuje w czasie kompilacji.
Operator przeciążony	Funkcja nadająca operatorowi znaczenie dla typu użytkownika, zachowujące oczekiwaną semantykę operatora.

Lekcja

Metoda const nie zmienia obserwowalnego stanu i może być wywołana dla obiektu const. Parametry złożone przekazujemy często jako const referencję, aby uniknąć kopii.

Pole static jest wspólne dla wszystkich obiektów klasy. Funkcja static nie ma wskaźnika this i może korzystać tylko z elementów statycznych lub przekazanych danych.

Przeciążenie powinno zachowywać intuicję operatora. Dla Wektora2D operator+ jest naturalny; dla Konta operator+ mógłby być mylący.

```
class Wektor2D {
public:
    double x, y;
    double dlugosc() const;
};
Wektor2D operator+(const Wektor2D& a, const Wektor2D& b) {
    return {a.x+b.x, a.y+b.y};
}
```

Rozwinięcie i praktyka Modern C++

Metoda const może być wywołana dla obiektu const, a w jej ciele this ma typ wskaźnika do const. To kontrola kompilatora, nie jedynie komentarz.

Małe typy, takie jak int czy wskaźnik, przekazujemy przez wartość. Większe obiekty tylko odczytywane - przez const T&. Jeśli funkcja przejmuje własność, często przyjmuje wartość lub unique_ptr.

Atrybut [[nodiscard]] warto dodać do funkcji, których wynik nie powinien być ignorowany. noexcept opisuje obietnicę braku wyjątków i może pomóc kontenerom bezpiecznie przenosić obiekty.

Przykład: porównywalny typ wartościowy

```
class IdUcznia {
    int value_;
public:
    explicit IdUcznia(int v) : value_(v) {
        if (v <= 0) throw std::invalid_argument{"id"};
    }
    [[nodiscard]] int value() const noexcept { return value_; }
    friend bool operator==(const IdUcznia&, const IdUcznia&) = default;
};
```

Typowa pułapka

Nie przeciążaj operatora w sposób zaskakujący. operator+ powinien tworzyć sumę, a nie np. zapisywać obiekt do pliku.

Ćwiczenie

Dodaj operator== dla Punktu oraz metodę przesun(dx,dy). Która metoda powinna być const?

Zadanie na ocenę 6

Omów koszt przekazywania std::string przez wartość, const& i && w konstruktorze.

Sprawdź się

Metoda const...

- A. nie zmienia obserwowalnego stanu
- B. jest zawsze statyczna
- C. nie może nic zwrócić
- D. działa tylko dla int

Odpowiedzi - tydzień 4

Odpowiedź do ćwiczenia

operator== nie zmienia operandów, więc przyjmuje const referencje. przesun zmienia obiekt i nie może być const.

Quiz

Poprawna odpowiedź: A. nie zmienia obserwowalnego stanu. const jest częścią kontraktu metody.

Kryterium zaliczenia

Potrafisz odtworzyć rozwiązanie bez zaglądania do odpowiedzi, wyjaśnić każdy krok i podać co najmniej jeden test graniczny.

Tydzień 5. Kompozycja i relacje między klasami

Poziom

Zakres docelowy: ocena 5-6

Po tym module

- zbudujesz klasę z innych obiektów
- odróżnisz kompozycję od agregacji
- unikniesz nadmiernego dziedziczenia

Definicje kluczowych pojęć

Pojęcie	Definicja
Relacja ma	Obiekt korzysta z innego obiektu jako składowej lub współpracownika.
Właściciel	Obiekt odpowiedzialny za zwolnienie zasobu lub zakończenie czasu życia posiadanego obiektu.
Obserwator	Wskaźnik, referencja lub widok, który nie posiada obserwowanego obiektu i nie przedłuża jego czasu życia.

Lekcja

Kompozycja oznacza silne 'ma': Samochod ma Silnik, którego czas życia należy do samochodu.

Agregacja to słabsza relacja, gdy obiekt jedynie odwołuje się do elementu istniejącego gdzie indziej.

Kompozycja jest zwykle prostsza niż dziedziczenie, bo nie tworzy zależności 'jest'. Delegujemy pracę do pól zamiast rozszerzać hierarchię.

Należy jasno ustalić własność. Zwykłe pole i `unique_ptr` oznaczają posiadanie; referencja lub obserwujący wskaźnik nie powinny przedłużać czasu życia.

```
class Zamowienie {
    Klient klient_;
    std::vector<Pozycja> pozycje_;
public:
    double wartosc() const;
};
```

Rozwinięcie i praktyka Modern C++

Zwykłe pole `T` oznacza silną kompozycję i najprostszy model własności. `optional<T>` oznacza opcjonalną część, `unique_ptr<T>` - dynamiczną wyłączną własność, a `reference_wrapper<T>` - nieposiadające odwołanie.

Kompozycja redukuje zależność od hierarchii. Obiekt może delegować zadanie do strategii przechowywanej jako pole, a strategię można później wymienić.

Wskaźnik obserwujący musi mieć zapewniony dłuższy czas życia celu. Jeśli nie można tego zagwarantować, model własności trzeba zmienić, zamiast liczyć na ostrożność programisty.

Przykład: zamówienie składające się z pozycji

```
class Zamowienie {
    Klient klient_;
    std::vector<Pozycja> pozycje_;
public:
    void dodaj(Pozycja p) { pozycje_.push_back(std::move(p)); }
    [[nodiscard]] Kwota wartosc() const {
        return std::accumulate(pozycje_.begin(), pozycje_.end(),
                                Kwota{}, [(Kwota s, const Pozycja& p){ return s + p.wartosc(); }]);
    }
};
```

Typowa pułapka

Jeżeli obiekt przechowuje surowy wskaźnik, dokumentuj brak własności i zabezpiecz czas życia. Nie wywołuj delete na obiekcie, którego nie posiadasz.

Ćwiczenie

Zaprojektuj Bibliotekę zawierającą vector<Ksiazka> i operacje dodaj, znajdzPoTytule, wypiszDostepne.

Zadanie na ocenę 6

Wskaż właściciela w relacji Drużyna-Gracz, jeśli gracz może zmieniać drużynę. Uzasadnij model.

Sprawdź się

Kompozycja modeluje relację...

- A. ma
- B. jest
- C. kompiluje
- D. dziedziczy zawsze

Odpowiedzi - tydzień 5

Odpowiedź do ćwiczenia

Biblioteka posiada książki, więc kompozycja z vector jest naturalna. Wyszukiwanie liniowe ma $O(n)$; dla częstych wyszukiwań można rozważyć mapę indeksującą tytuł.

Quiz

Poprawna odpowiedź: A. ma. Obiekt składa się z innych obiektów.

Kryterium zaliczenia

Potrafisz odtworzyć rozwiązanie bez zaglądania do odpowiedzi, wyjaśnić każdy krok i podać co najmniej jeden test graniczny.

Tydzień 6. Dziedziczenie i przesłanianie

Poziom

Zakres docelowy: ocena 5

Po tym module

- zapiszesz klasę bazową i pochodną
- użyjesz override
- rozpoznasz poprawną relację 'jest'

Definicje kluczowych pojęć

Pojęcie	Definicja
Klasa bazowa	Typ, którego interfejs lub implementacja są dziedziczone przez klasę pochodną.
Klasa pochodna	Typ zdefiniowany z bazą, który rozszerza lub specjalizuje jej zachowanie.
Zasada podstawienia	Jeśli D publicznie dziedziczy po B, obiekt D powinien spełniać kontrakt B i być używalny w miejscu oczekującym B.

Lekcja

Dziedziczenie modeluje relację 'jest', np. Kolo jest Figurą. Klasa pochodna dziedziczy publiczny interfejs i może przesłonić metody wirtualne.

Słowo `override` prosi kompilator o sprawdzenie zgodności sygnatury. Chroni przed przypadkowym utworzeniem innej metody zamiast przesłonięcia.

Nie używaj dziedziczenia wyłącznie do ponownego wykorzystania kilku linii kodu. Najpierw sprawdź, czy każdy obiekt pochodny naprawdę może zastąpić bazowy bez łamania kontraktu.

```
class Figura {
public:
    virtual double pole() const = 0;
    virtual ~Figura() = default;
};
class Kolo : public Figura {
    double r_;
public:
    double pole() const override;
};
```

Rozwinięcie i praktyka Modern C++

Publiczne dziedziczenie jest deklaracją semantyczną, nie tylko mechanizmem technicznym. Kolo jest Figurą, ale Silnik nie jest Samochodem - samochód ma silnik.

Konstruktor klasy pochodnej najpierw konstruuje bazę, potem pola pochodne. Destrakcja zachodzi odwrotnie. Prywatne pola bazy nie są bezpośrednio dostępne w klasie pochodnej.

`final` na klasie zakazuje dalszego dziedziczenia; `final` na funkcji zakazuje kolejnego przesłonięcia. Używaj, gdy dalsze rozszerzanie naruszałoby kontrakt lub nie ma sensu projektowego.

Przykład: hierarchia z poprawnym kontraktem

```
class Figura {
public:
    [[nodiscard]] virtual double pole() const noexcept = 0;
    virtual ~Figura() = default;
};
class Koło final : public Figura {
    double r_;
public:
    explicit Koło(double r) : r_(r) {
        if (r <= 0) throw std::invalid_argument{"promien"};
    }
    double pole() const noexcept override { return std::numbers::pi*r_*r_; }
};
```

Typowa pułapka

Klasyczny problem: Kwadrat dziedziczy po Prostokacie z niezależnymi setSzerokosc/setWysokosc. Kwadrat nie może zachować kontraktu obu setterów, więc podstawienie jest wadliwe.

Ćwiczenie

Dodaj Prostokąt jako Figurę i zapisz implementację pole().

Zadanie na ocenę 6

Wyjaśnij zasadę podstawienia Liskov na kontrprzykładzie Kwadrat dziedziczący po Prostokacie z niezależnymi setterami boków.

Sprawdź się

override pomaga wykryć...

- A. niezgodną sygnaturę przesłanianej metody
- B. brak średnika w dowolnym pliku
- C. wyciek sieci
- D. złą nazwę zmiennej lokalnej

Odpowiedzi - tydzień 6

Odpowiedź do ćwiczenia

```
class Prostokat : public Figura { double a_,b_; public: ... double pole() const override { return a_*b_; } };
```

Quiz

Poprawna odpowiedź: A. niezgodną sygnaturę przesłanianej metody. Kompilator sprawdza, czy metoda rzeczywiście przesłania wirtualną.

Kryterium zaliczenia

Potrafisz odtworzyć rozwiązanie bez zaglądania do odpowiedzi, wyjaśnić każdy krok i podać co najmniej jeden test graniczny.

Tydzień 7. Polimorfizm i klasy abstrakcyjne

Poziom

Zakres docelowy: ocena 5-6

Po tym module

- wywołasz metodę przez interfejs bazowy
- zastosujesz wirtualny destruktor
- użyjesz `unique_ptr`

Definicje kluczowych pojęć

Pojęcie	Definicja
Dynamiczna dyspozycja	Mechanizm wyboru końcowego przesłonięcia funkcji virtual na podstawie typu dynamicznego obiektu.
Interfejs abstrakcyjny	Klasa bazowa skupiona na deklaracji operacji, zwykle bez stanu i z metodami czysto wirtualnymi.
Inteligentny wskaźnik	Obiekt zachowujący się jak wskaźnik i automatycznie realizujący określoną politykę własności.

Lekcja

Polimorfizm pozwala traktować różne obiekty przez wspólny interfejs. Wywołanie metody virtual wybiera implementację zgodną z rzeczywistym typem obiektu.

Klasa z metodą czysto wirtualną jest abstrakcyjna i nie można tworzyć jej obiektów. Dstruktor bazy używanej polimorficznie musi być wirtualny.

Kolekcja `std::unique_ptr<Figura>` przechowuje obiekty różnych klas bez kopiowania i jasno wyraża własność. `make_unique` tworzy zasób bezpiecznie.

```
std::vector<std::unique_ptr<Figura>> figury;
figury.push_back(std::make_unique<Kolo>(2));
figury.push_back(std::make_unique<Prostokat>(3,4));
for (const auto& f : figury)
    std::cout << f->pole() << '\n';
```

Rozwinięcie i praktyka Modern C++

Polimorfizm wymaga pracy przez referencję lub wskaźnik do bazy. Przekazanie po wartości tworzy osobny obiekt bazowy i usuwa część pochodną - to slicing.

Właściciel kolekcji heterogenicznej zwykle przechowuje `unique_ptr<Base>`. `shared_ptr` wybieramy tylko wtedy, gdy czas życia naprawdę ma wielu równorzędnych właścicieli.

Nie wywołuj funkcji virtual z konstruktora lub destruktor, oczekując zachowania klasy pochodnej. W tych fazach dynamiczna część pochodna nie jest jeszcze gotowa albo już została zniszczona.

Przykład: polimorficzna kolekcja bez ręcznego delete

```
std::vector<std::unique_ptr<Figura>> figury;
figury.push_back(std::make_unique<Kolo>(2.0));
figury.push_back(std::make_unique<Prostokat>(3.0, 4.0));

double suma = 0;
for (const auto& f : figury) suma += f->pole();
// vector i unique_ptr automatycznie niszczą kompletne obiekty
```

Typowa pułapka

Baza z publicznym niewirtualnym destruktorom używana przez `unique_ptr<Base>` prowadzi do nieokreślonego zachowania przy niszczeniu obiektu pochodnego.

Ćwiczenie

Utwórz interfejs `Platnosc` z `wykonaj(kwota)` i dwie implementacje: `Karta` i `Przelew`. Przetwórz je przez wspólną funkcję.

Zadanie na ocenę 6

Wyjaśnij różnicę między `unique_ptr` i `shared_ptr` oraz dlaczego `shared_ptr` nie powinien być wyborem domyślnym.

Sprawdź się

Wirtualny destruktor bazy jest potrzebny, gdy...

- A. usuwamy obiekt pochodny przez wskaźnik bazowy
- B. używamy wyłącznie `int`
- C. nie ma dziedziczenia
- D. klasa ma tylko `static`

Odpowiedzi - tydzień 7

Odpowiedź do ćwiczenia

Platnosc ma virtual bool wykonaj(double)=0 i virtual ~Platnosc()=default. Funkcja przyjmuje Platnosc& lub kolekcję unique_ptr<Platnosc>.

Quiz

Poprawna odpowiedź: A. usuwamy obiekt pochodny przez wskaźnik bazowy. Zapewnia wykonanie pełnego łańcucha destruktorów.

Kryterium zaliczenia

Potrafisz odtworzyć rozwiązanie bez zaglądania do odpowiedzi, wyjaśnić każdy krok i podać co najmniej jeden test graniczny.

Tydzień 8. Kolekcje obiektów, algorytmy STL i koszt

Poziom

Zakres docelowy: ocena 5-6

Po tym module

- przechowasz obiekty w vector
- zastosujesz sort i find_if
- ocenisz kopiowanie i złożoność

Definicje kluczowych pojęć

Pojęcie	Definicja
Semantyka wartości	Obiekt zachowuje się jak niezależna wartość: może być kopiowany, porównywany i przechowywany bez ukrytej współwłasności.
Iterator	Obiekt wskazujący pozycję w sekwencji i umożliwiający algorytmom przechodzenie po elementach bez znajomości kontenera.
Lambda	Anonimowy obiekt funkcyjny tworzony składnią [], który może przechwytywać wartości z otaczającego zakresu.

Lekcja

`std::vector<T>` przechowuje obiekty ciągle w pamięci. Dodanie na końcu ma amortyzowane $O(1)$, dostęp po indeksie $O(1)$, a wyszukiwanie liniowe $O(n)$.

Algorytmy STL oddzielają operację od kontenera. Funkcja `sort` wymaga porządku, a `find_if` predykatu. Lambdy pozwalają krótko zapisać kryterium.

Duże obiekty przekazuj przez `const` referencję. Przy sortowaniu mogą wystąpić przeniesienia; dobrze zaprojektowane typy korzystają z semantyki przenoszenia zapewnianej przez pola standardowe.

```
std::sort(uczniowie.begin(), uczniowie.end(),
    [](const Uczeń& a, const Uczeń& b) {
        return a.srednia() > b.srednia();
    });
```

Rozwinięcie i praktyka Modern C++

`vector<T>` daje dobrą lokalność pamięci i jest domyślnym kontenerem sekwencyjnym. Rezerwuj pojemność, gdy znasz przybliżony rozmiar i chcesz ograniczyć realokację.

Algorytmy STL opisują zamiar czytelniej niż ręczne pętle: `sort`, `find_if`, `transform`, `count_if`. W C++20 `ranges` często upraszczają przekazywanie całego zakresu.

Złożoność należy wiązać z publiczną operacją. Wyszukanie w `vector` jest $O(n)$, w `unordered_map` średnio $O(1)$, ale mapa zużywa więcej pamięci i nie zachowuje zwartego układu.

Przykład: sortowanie i wyszukiwanie obiektów

```
std::ranges::sort(uczniowie, std::greater{}, &Uczen::srednia);
auto it = std::ranges::find_if(uczniowie, [](const Uczen& u) {
    return u.srednia() >= 4.75;
});
if (it != uczniowie.end()) std::cout << it->imie();
```

Typowa pułapka

Nie przechowuj polimorficznych obiektów jako `vector<Base>` - nastąpi slicing. Użyj `vector<unique_ptr<Base>>` albo zamkniętego wariantu `std::variant`.

Ćwiczenie

Wyszukaj pierwszą książkę autora 'Lem' przez `find_if`. Podaj złożoność.

Zadanie na ocenę 6

Porównaj `vector<Figura>`, `vector<unique_ptr<Figura>>` i `vector<variant<Kolo,Prostokat>>` pod kątem polimorfizmu, pamięci i lokalności.

Sprawdź się

Dostęp `vector[i]` ma złożoność...

- A. $O(1)$
- B. $O(n)$
- C. $O(\log n)$
- D. $O(n^2)$

Odpowiedzi - tydzień 8

Odpowiedź do ćwiczenia

Predykat porównuje autor(); find_if przechodzi najwyżej n elementów, więc $O(n)$ czasu i $O(1)$ dodatkowej pamięci.

Quiz

Poprawna odpowiedź: A. $O(1)$. Wektor zapewnia dostęp swobodny.

Kryterium zaliczenia

Potrafisz odtworzyć rozwiązanie bez zaglądania do odpowiedzi, wyjaśnić każdy krok i podać co najmniej jeden test graniczny.

Tydzień 9. Projekt wieloplikowy i testowanie

Poziom

Zakres docelowy: ocena 5-6

Po tym module

- podzielimy deklaracje i definicje
- przetestujemy zachowanie klas
- uzasadnimy decyzje projektowe

Definicje kluczowych pojęć

Pojęcie	Definicja
Jednostka translacji	Plik źródłowy po przetworzeniu dyrektyw <code>#include</code> ; kompilator przetwarza każdą jednostkę osobno.
ODR	One Definition Rule: reguły określające, ile definicji elementu programu może istnieć i kiedy definicje w różnych jednostkach muszą być identyczne.
Test jednostkowy	Automatyczny test małej jednostki zachowania, wykonywany niezależnie i dający jednoznaczny wynik.

Lekcja

Deklarację klasy umieszczamy zwykle w `.hpp`, a definicje metod w `.cpp`. Nagłówek powinien być samowystarczalny i chroniony przez `#pragma once` lub `include guards`.

Testujemy publiczny kontrakt: konstrukcję, poprawne operacje, granice i błędne dane. Nie testujemy bezpośrednio prywatnych pól, bo są szczegółem implementacji.

Projekt na ocenę 6 powinien zawierać opis odpowiedzialności klas, diagram relacji, uzasadnienie kompozycji/dziedziczenia, analizę kosztów oraz instrukcję kompilacji.

```
// Konto.hpp
#pragma once
class Konto { /* deklaracje */ };

// Konto.cpp
#include "Konto.hpp"
// definicje metod
```

Rozwinięcie i praktyka Modern C++

Nagłówek zawiera deklaracje potrzebne użytkownikowi, a plik `.cpp` szczegóły implementacji. `#pragma once` lub `include guards` zapobiegają wielokrotnemu włączeniu tej samej definicji w jednostce translacji.

Testuj kontrakt: wynik operacji, zachowanie na granicach i reakcję na błędne dane. Prywatna reprezentacja może się zmienić bez zmiany testów.

Współczesny projekt włącza ostrzeżenia kompilatora, formatter, analizę statyczną i testy. Minimalne flagi GCC/Clang: `-std=c++23 -Wall -Wextra -Wpedantic`, a w debugowaniu sanitizery adresów i zachowań niezdefiniowanych.

Przykład: prosty test kontraktu

```
void test_wypłaty() {
    Konto k{1000};
    assert(k.wypłac(400));
    assert(k.saldo() == 600);
    assert(!k.wypłac(700));
    assert(k.saldo() == 600); // nieudana operacja nie zmienia stanu
}
```

Typowa pułapka

Test zależny od prywatnego pola lub kolejności pól utrudnia refaktoryzację. Sprawdzaj obserwowalne zachowanie publicznego interfejsu.

Ćwiczenie

Zaplanuj testy klasy Konto: konstrukcja, wpłata, wypłata, brak środków, kwota zero i ujemna.

Zadanie na ocenę 6

Przygotuj mini-projekt Biblioteka: Książka, Czytelnik, Wypożyczenie, Biblioteka. Uzasadnij własność i czas operacji wyszukiwania.

Sprawdź się

Test klasy powinien opierać się przede wszystkim na...

- A. publicznym kontrakcie
- B. adresach prywatnych pól
- C. kolejności pól w pamięci
- D. nazwie pliku wykonywalnego

Odpowiedzi - tydzień 9

Odpowiedź do ćwiczenia

Po każdej operacji sprawdź wartość zwrótną i saldo. Niepoprawna operacja ma pozostawić stan bez zmian. Dodaj serię operacji, by sprawdzić niezmiennik.

Quiz

Poprawna odpowiedź: A. publicznym kontrakcie. Szczegóły prywatne mogą się zmieniać bez zmiany zachowania.

Kryterium zaliczenia

Potrafisz odtworzyć rozwiązanie bez zaglądania do odpowiedzi, wyjaśnić każdy krok i podać co najmniej jeden test graniczny.

Test końcowy

Najpierw rozwiąż bez zaglądania do klucza. Za każde pytanie: 1 punkt. Następnie wybierz jedno zadanie projektowe.

1. Obiekt jest...

- A. instancją klasy
- B. plikiem nagłówkowym
- C. kompilatorem
- D. pętlą

2. Niezmiennik klasy powinien być prawdziwy...

- A. po każdej publicznej operacji
- B. tylko przed konstruktorem
- C. wyłącznie po błędzie
- D. tylko w debuggerze

3. Lista inicjalizacyjna konstruktora...

- A. inicjalizuje pola przed ciałem
- B. wywołuje main
- C. usuwa obiekt
- D. działa tylko dla static

4. RAII oznacza...

- A. zasób związany z czasem życia obiektu
- B. ręczne delete wszędzie
- C. brak konstruktorów
- D. wyłącznie dziedziczenie

5. Kompozycja odpowiada relacji...

- A. ma
- B. jest
- C. dzieli
- D. kompiluje

6. Dziedziczenie publiczne modeluje zwykle...

- A. jest
- B. ma
- C. zna
- D. tworzy plik

7. Klasa abstrakcyjna...

- A. może mieć metodę czysto wirtualną
- B. nie ma metod
- C. zawsze jest final
- D. musi być struct

8. `unique_ptr` wyraża...

- A. wyłączną własność
- B. współdzieloną własność
- C. brak własności
- D. wartość całkowitą

9. `find_if` na wektorze ma w najgorszym przypadku...

- A. $O(n)$
- B. $O(1)$
- C. $O(\log n)$
- D. $O(n^2)$

10. Metoda oznaczona `const...`

- A. nie zmienia obserwowalnego stanu
- B. jest konstruktorem
- C. jest prywatna
- D. musi zwracać `void`

Zadania projektowe

Projekt 1: System biblioteczny

Zaprojektuj klasy Książka, Czytelnik, Wypożyczenie i Biblioteka. Dodaj walidację, wyszukiwanie, wypożyczenie i zwrot oraz testy.

Projekt 2: Figury

Utwórz abstrakcyjną Figurę, Kolo i Prostokat, kolekcję unique_ptr oraz obliczenie sumy pól.

Projekt 3: Dziennik ocen

Zaprojektuj Ucznia i Przedmiot, oblicz średnie, sortuj uczniów i znajdź najlepszy wynik. Dodaj testy danych pustych.

Rubryka oceny projektu

Kryterium	5	6
Poprawność	działa dla testów typowych i granicznych	uzasadnione niezmienniki i pełna obsługa błędów
Struktura	czytelny podział na funkcje/klasę	uzasadniony dobór struktur i odpowiedzialności
Analiza	czas i pamięć głównych operacji	porównanie alternatyw, wzory i kompromisy
Samodzielność	dokumentacja i czytelny kod	rozszerzenie lub drugi wariant rozwiązania

Klucz odpowiedzi

1. A - instancją klasy. Klasa definiuje typ, obiekt jest jego egzemplarzem.
2. A - po każdej publicznej operacji. Interfejs musi zachować poprawny stan.
3. A - inicjalizuje pola przed ciałem. Pola powstają przed wejściem do ciała.
4. A - zasób związany z czasem życia obiektu. Destruktor zwalnia zasób automatycznie.
5. A - ma. Obiekt zawiera inne obiekty.
6. A - jest. Obiekt pochodny powinien zastępować bazowy.
7. A - może mieć metodę czysto wirtualną. Metoda `=0` czyni klasę abstrakcyjną.
8. A - wyłączną własność. Nie można go kopiować; można przenosić.
9. A - $O(n)$. Może sprawdzić każdy element.
10. A - nie zmienia obserwowalnego stanu. To część kontraktu metody.

Wskazówki do projektów

System biblioteczny
Biblioteka posiada kolekcje książek i wypożyczeń; Czytelnik może być identyfikowany przez ID. Wypożyczenie łączy identyfikatory i daty. Publiczne operacje pilnują dostępności i limitu.
Figury
Figura ma virtual pole() const=0 i wirtualny destruktork. Kolekcja <code>vector<unique_ptr<Figura>></code> daje polimorfizm i jasną własność; suma pól to $O(n)$.
Dziennik ocen
Rozdziel dane ucznia od operacji dziennika. Sortowanie $O(n \log n)$, maksimum $O(n)$. Zdecyduj, czy brak ocen zwraca optional czy zgłasza wyjątek i stosuj konsekwentnie.

Interpretacja wyniku

- 9-10 pkt + dobry projekt: gotowość do pracy na poziomie 5-6
- 7-8 pkt: dobra podstawa; powtórz moduły powiązane z błędami
- 5-6 pkt: wróć do tygodni 1-6 i wykonaj ćwiczenia ponownie
- 0-4 pkt: przejdź materiał od początku w wolniejszym tempie