Programowanie obiektowe

Zajęcia 5. Polimorfizm, klasy abstrakcyjne, rodzaje rzutowań

Zadanie 1. Polimorfizm w C++

1. Utwórz w wybranym miejscu na dysku folder `PolimorfizmCpp`.
2. W VSC otwórz utworzony w poprzednim punkcie folder.
3. Do projektu dodaj plik `main.cpp`.
4. Wykonaj implementację poniższych klas (zadbaj by stan obiektów był zabezpieczony zgodnie z założeniami enkapsulacji i można było się do nich dostać tylko i wyłącznie za pomocą dedykowanych metod; do projektu dodaj odpowiednie pliki `cpp` oraz `h` dla każdej z definiowanych klas.):
   (a) `Osoba`, którą cechują następujące własności:
      - imię i nazwisko
      - data urodzenia
      - miejsce zamieszkania (należy stworzyć dodatkową klasę `Adres`)
      - ustawienie imienia i nazwiska
      - odczytanie imienia i nazwiska
      - ustawienie daty urodzenia ze sprawdzeniem poprawności przekazanej daty

- odczytanie daty urodzenia
- ustawienie miejsca zamieszkania ze sprawdzeniem poprawności przekazanych daych adresowych
- odczytanie miejsca zamieszkania
- metoda o nazwie przedstaw, której zadaniem jest wyświetlenie pełnego opisu osoby

(b) `Student` (dziedziczy po klasie `Osoba`), którego cechują następujące własności:

- lista ocen indeksowana nazwą przedmiotu
- dodanie oceny z danego przedmiotu (liczby z zakresu 1 do 5)
- odczytanie oceny dla wybranego przedmiotu (należy sprawdzić czy student ma ocenę z danego przedmiotu)
- odczytanie listy ocen dla wszystkich przedmiotów
- wykonaj redefinicję metody przedstaw z klasy `Osoba`, tak, aby wyświetlała pełną informację na temat danego studenta.

(c) `Pracownik` (dziedziczy po klasie `Osoba`), którego cechują następujące własności:

- lista przedmiotów prowadzonych przez danego pracownika
- dodanie przedmiotu (o ile już nie istnieje)
- usunięcie przedmiotu (o ile istnieje)
- sprawdzenie czy pracownik prowadzi dany przedmiot
- wykonaj redefinicję metody przedstaw z klasy `Osoba`, tak, aby wyświetlała pełną informację na temat danego pracownika.

5. W pliku `main.cpp` dodaj przykładową implementację funkcji main testującą podstawową funkcjonalność zaimplementowanych klas.

6. Skompiluj i uruchom plik `main.cpp` w trybie debug np. poprzez wciśnięcie przycisku F5 a następnie wybierz odpowiednią dla Twojego systemu i kompilatora konfigurację.

7. Zmodyfikuj użytą konfigurację (zapisaną w pliku `.vscode/tasks.json`) w taki sposób żeby zamiast pojedynczego pliku kompilacji uległy wszystkie pliki cpp zawarte w katalogu, w którym znajduje się aktualnie przez nas modyfikowany plik. Możesz tego dokonać poprzez zamianę użytego w pliku `${file}` na `${fileDirname}/*.cpp` lub nieco ogólniej `${fileDirname}/*${fileExtname}`.

8. Utwórz funkcję przyjmującą obiekt klasy `Osoba` przez referencję oraz przez wskaźnik. Np.:

```
void foo(Osoba &os) {
  os.przedstaw();
}
void bar(Osoba *os) {
  os->przedstaw();
}
```

9. Utwórz obiekt klasy `Osoba`, wywołaj metody ustawiające wartości poszczególnych pól, a następnie przekaż obiekt do utworzonej w punkcie 8 funkcji. Skompiluj program i uruchom go w trybie debug (np. poprzez wciśnięcie F5).

10. Utwórz obiekt klasy `Pracownik`, wywołaj metody ustawiające wartości poszczególnych pól, a następnie przekaż obiekt do utworzonej w punkcie 8 funkcji. Skompiluj program i uruchom go w trybie debug (np. poprzez wciśnięcie F5).

11. Utwórz obiekt klasy `Student`, wywołaj metody ustawiające wartości poszczególnych pól, a następnie przekaż obiekt do utworzonej w punkcie 8 funkcji. Skompiluj program i uruchom go w trybie debug (np. poprzez wciśnięcie F5).

12. Porównaj efekt wywołań w punktach od 9 do 11, zapisz swoje obserwacje i spróbuj je zinterpretować.

13. Zmodyfikuj deklarację funkcji `przedstaw` w klasie `Osoba` poprzez dodanie specyfikatora `virtual`.

14. Ponownie wykonaj kroki od 9 do 11, skompiluj program i uruchom go w trybie debug (np. poprzez wciśnięcie F5), zapisz swoje obserwacje i spróbuj je zinterpretować.

15. Wprowadź następujący fragment kodu do funkcji `main`:

```
Pracownik *p1 = new Pracownik;
Osoba *o1 = static_cast<Osoba *>(p1);
Osoba *o2 = static_cast<Pracownik *>(o1);

Osoba *o3 = new Osoba;
Pracownik *p2 = static_cast<Pracownik *>(o3);
```

    Czy wszystkie z powyższych rzutowań są poprawne? Czy wszystkie spośród powyższych rzutowań są potrzebne? Skompiluj program i uruchom go w trybie debug (np. poprzez wciśnięcie F5), zapisz swoje wnioski.

16. Wprowadź następujący fragment kodu do funkcji `main`:

```
Pracownik *p3 = new Pracownik;
Osoba *o4 = dynamic_cast<Osoba *>(p3);
Pracownik *p4 = dynamic_cast<Osoba *>(o4);
```

Czy wszystkie z powyższych rzutowań są poprawne? Czy wszystkie spośród powyższych rzutowań są potrzebne? Skompiluj program i uruchom go w trybie debug (np. poprzez wciśnięcie F5), zapisz swoje wnioski.

17. Wprowadź następujący fragment kodu do funkcji `main`:

```
Osoba *o5 = new Osoba;
Pracownik *p5 = dynamic_cast<Pracownik *>(o5);
if(p5 == nullptr) std::cout << "Blad rzutowania!" << std::endl;
else p5 -> listaPrzedmiotow();
```

Jak myślisz, dlaczego należy unikać testów `dynamic_cast`! =`nullptr`?

18. Zastąp definicję funkcji `przedstaw` w klasie `Osoba` następującą:

```
virtual void przedstaw() = 0;
```

19. Czy teraz możesz skompilować projekt? Postaraj się ustalić przyczynę problemu, opisz ją a następnie postaraj się zmodyfikować kod tak, żeby projekt znowu się kompilował.

20. Wszystkie pola implementowanych klas zmodyfikuj w taki sposób, żeby możliwy był dla nich lazy loading, tj. niech stanowią wskaźniki na miejsce w pamięci przechowujące wartość przechowywanej w polu właściwości.

21. Utwórz dla nich odpowiednio: destruktory, konstruktory kopiujące, konwertujące, przenoszące oraz domyślny.

22. Skompiluj program i uruchom go w trybie debug (np. poprzez wciśnięcie F5). Czy program działa poprawnie, czy pamięć zaalokowana na obiekty została poprawnie zwolniona? Zapisz swoje obserwacje i spróbuj je zinterpretować.

23. Zmodyfikuj deklarację destruktora w klasie `Osoba` poprzez dodanie specyfikatora `virtual`. Ponownie skompiluj program i uruchom go w trybie debug (np. poprzez wciśnięcie F5). Co uległo zmianie? Zapisz swoje obserwacje i spróbuj je zinterpretować.

24. Utwórz w projekcie abstrakcyjną klasę `FabrykaOsob` (utwórz dla niej plik źródłowy i nagłówkowy). Zaimplementuj w niej metodę utworz(), która tworzy obiekt klasy `Osoba`:

```
virtual Osoba* utworz() = 0;
```

25. Utwórz w projekcie klasę `FabrykaStudentow` (utwórz dla niej plik źródłowy i nagłówkowy) dziedziczącą po klasie `FabrykaOsob`. Zaimplementuj w niej metodę utworz(), która tworzy (produkuje) obiekt klasy `Student`.

26. Utwórz w projekcie klasę `FabrykaPracownikow` (utwórz dla niej plik źródłowy i nagłówkowy) dziedziczącą po klasie `FabrykaOsob`. Zaimplementuj w niej metodę utworz(), która tworzy (produkuje) obiekt klasy `Pracownik`.

27. W pliku `main.cpp` dodaj kod testujący podstawową funkcjonalność zaimplementowanych klas. Skompiluj i uruchom program.

28. Zaimplementuj możliwość tworzenia nowych osób: studentów lub pracowników (w pętli), które będą przechowywane w tablicy. Do tego celu należy stworzyć proste menu, które będzie obsługiwało podawane z wejścia komendy. Użytkownik może podawać następujące komendy słowne: "pracownik" (utwórz obiekt klasy `Pracownik` i dodaj go do tablicy), "student" (utwórz obiekt klasy `Student` i dodaj go do tablicy), "wyswietl" (wyświetlenie zawartości tablicy, gdzie dla każdego obiektu zostanie wywołana jego metoda `przedstaw`), "wyjscie" (zakończenie działania programu). Aby to zrealizować wykonaj następujące kroki:

29. W funkcji `main` wprowadź następującą definicję:

```
std::map<std::string, FabrykaOsob*> m;
```

Kontener typu `map` (tablica asocjacyjna), który umożliwia ustawienie typu dla klucza i wartości przechowywanej, w naszym zadaniu pozwoli powiązać komendy wydawane przez użytkownika ("pracownik" lub "student") z odpowiednią "fabryką" studentów lub pracowników, czyli pozwoli na wywołanie odpowiedniej metody `utworz()` produkującej obiekt typu `Pracownik` lub `Student`.

30. W tym celu należy dodać 2 elementy do obiektu klasy `map` z wykorzystaniem operatora `[ ]` w następujący sposób (wykonywane tylko raz, przed pętlą w ramach której będzie realizowana interakcja z użytkownikiem i do tablicy dodawane będą obiekty klas `Pracownik`, `Student`):

```
m["pracownik"] = new FabrykaPracownikow();
m["student"] = new FabrykaStudentow();
```

Zwróć uwagę na definicję kontenera typu `map` z punktu 29 oraz na powyższe instrukcje. Zapisz swoje wnioski.

31. W celu przechowania utworzonych obiektów klas `Pracownik`, `Student` wykorzystaj np. kontener `vector`:

```
std::vector<Osoba*> tablicaOsob;
```

32. W pętli, za każdym razem gdy użytkownik wyda komendę: "pracownik" (utwórz obiekt klasy `Pracownik`) lub "student" (utwórz obiekt klasy `Student`) wystarczy wykorzystać poniższą instrukcję do stworzenia odpowiedniego obiektu i umieszczenia go w obiekcie klasy `vector`:

```
tablicaOsob.push_back(m[currentCommand]->utworz());
```

Gdzie `currentCommand` to zmienna typu `std::string`, która przechowuje wczytany od użytkownika tekst: "pracownik" lub "student".

33. Skompiluj program i uruchom go w trybie debug (np. poprzez wciśnięcie F5), zapisz swoje obserwacje, wnioski i spróbuj je zinterpretować.