Rozdział 8

Dziedziczenie i klasy pochodne

8.1 Wstęp

Często podczas tworzenia klasy napotykamy na sytuację, w której nowa klasa powiększa możliwości innej klasy, nierzadko precyzując jednocześnie jej funkcjonalność. Dziedziczenie to sposób na ponowne wykorzystanie kodu, w której wykorzystujemy już istniejącą klasę nazywaną, w przypadku dziedziczenia - podstawową. Nowa klasa tworzona jest na podstawie klasy podstawowej pobierając jej atrybuty, ale również dodając dane wejściowe, które konieczne są do zdefiniowania klasy. Klasę która dziedziczy nazywa się klasą pochodną. Klasa pochodna jest rozwinięciem klasy podstawowej. W listingu 8.1 przedstawiono przykład dziedziczenia klas w C++. Klasą bazową w tym przypadku będzie `Zwierze`, natomiast na jej podstawie zbudowaną klasą pochodną będzie `Pies`. W linii 22 widać w jaki sposób można zrealizować dziedziczenie od innej klasy - po nazwie klasy należy postawić dwukropek i napisać nazwę klasy, od której chcemy dziedziczyć (jest tam też słowo kluczowe `public`, ale o tym opowiemy w rozdziale 8.2). Jest to tzw. *lista pochodzenia klasy*. W przypadku klasy pochodnej `Pies` rozszerzane są podstawowe czynności, które wykonuje pies o szczekanie oraz warczenie, co możemy zobaczyć w linii 38. Dodatkowo klasa `Pies` zawiera wszystkie składniki z klasy `Zwierze` o czym możemy się np. przekonać w linii 39, gdzie wywołana jest funkcja `jedz` z klasy podstawowej. Widzimy też, że do składników, które zostały odziedziczone odnosimy się tak samo jakby były zdefiniowane w klasie pochodnej `Pies`.

Należy pamiętać, że dziedziczenie może być wielopokoleniowe, czyli klasa pochodna może stać się klasą podstawową dla innej klasy. Np. `Zwierze`—>`Pies`—>`Jamnik`.

```
1  #include <iostream>
2
3  class Zwierze
4  {
5     public:
6        Zwierze() { }
```

```
7        void jedz( )
8        {
9            for (int i = 0; i < 10; ++i)
10              std::cout << "Om Nom Nom Nom\n";
11       }
12       void pij( )
13       {
14           for (int i = 0; i < 5; ++i)
15              std::cout << "Chlip, chlip\n";
16       }
17       void spij( )
18       {
19           std::cout << "Chrr...\n";
20       }
21 };
22 class Pies : public Zwierze
23 {
24    public:
25      Pies() { }
26      void szczekaj()
27      {
28         std::cout << "Hau! hau!...\n";
29      }
30      void warcz()
31      {
32         std::cout << "Wrrrrrr...\n";
33      }
34 };
35 int main()
36 {
37   Pies Kajtek;
38   Kajtek.szczekaj();
39   Kajtek.jedz();
40 }
```

Listing 8.1: Przykład dziedziczenia klas

Zapamiętaj

Konstruktory i destruktory nie są dziedziczone.

8.2 Dostęp do składników klasy

Przy dziedziczeniu możemy decydować jaki będzie dostęp do składników klasy. Mamy dwa sposoby na określanie tego dostępu:

1. dostęp do składników zdefiniowany wewnątrz klasy
2. dostęp do składników odziedziczonych po klasie podstawowej

W pierwszym przypadku dostęp do składników w klasie ustalany jest w momencie definiowania klasy. Służą do tego znane nam już słowa kluczowe `public` i `private` oraz nowe słowo kluczowe `protected` utworzone na potrzeby dziedziczenia:

- `public` - dostęp do składnika ma każdy
- `private` - dostęp do składnika ma tylko klasa, w której jest zdefiniowany. W praktyce oznacza to, że dostęp do tych składników jest tylko przez funkcje zdefiniowane w klasie. Wynika stąd, że jeśli klasa `B` dziedziczy od `A` i wewnątrz `A` są składniki prywatne (np. `x`) to klasa pochodna `B` nie ma do nich bezpośredniego dostępu (jedynie przez funkcje publiczne z klasy podstawowej `A`), czyli nie można napisać `A.x` .
- `protected` - nie ma dostępu do składnika spoza klasy natomiast składnik jest dostępny wewnątrz klas pochodnych

Zapamiętaj

Klasa dzieli się na interfejs (to co widzi użytkownik klasy) i implementację (to co widzi projektant klasy). Interfejs oznaczamy etykietą `public` natomiast implementację etykietą `private`.

W drugim przypadku o sposobie dostępu do składnika zdefiniowanego wewnątrz klasy bazowej decyduje się na liście pochodzenia klasy. W poprzednim przykładzie (listing 8.1) widzieliśmy, że na liście pochodzenia klasy oprócz nazwy klasy podstawowej było też słowo kluczowe `public`. Inne możliwości jakie mogły się tam pojawić to `private` i `protected`. Poniżej opisane jest co dane słowa kluczowe na liście pochodzenia klasy oznaczają:

- `public` - odziedziczone składniki publiczne są wewnątrz klasy pochodnej `public` natomiast składniki `protected` są `protected`
- `protected` - odziedziczone składniki `public` i `protected` są w klasie pochodnej jako `protected`
- `private` - odziedziczone składniki `public` i `protected` są w klasie pochodnej jako `private`
- jeśli nie ma tam żadnego słowa kluczowego to dla `class` domniemanie jest jako `private`, a dla `struct` jako `public`.

Należy pamiętać, że określanie sposobu dostępu na liście pochodzenia klasy dotyczy tylko nieprywatnych składników (`public` i `protected`). Składniki prywatne z klasy podstawowej są nadal niedostępne (bezpośrednio) w klasie pochodnej.

Zapamiętaj

Dziedziczenie `private` stosujemy, gdy nie chcemy aby było wiadomo, że obiekty klasy pochodnej są szczególnym rodzajem obiektów klasy podstawowej.

8.3 Zasłanianie składników

Może się zdarzyć, że klasa pochodna będzie miała daną składową lub funkcję o identycznej nazwie jak w klasie podstawowej. W tym momencie składnik o tej nazwie w klasie bazowej zostaje zasłonięty przez nazwę składnika z klasy podstawowej. Przykład pokazany jest w listingu 8.2. Widzimy, że w klasie bazowej `Punkt2D` jest funkcja `pokaz`. Taka sama nazwa funkcji pojawia się w klasie pochodnej. W momencie wywołania na obiekcie klasy pochodnej `LepszyPunkt2D` funkcji `pokaz` zostaje wywołana funkcja z klasy pochodnej (linia 18). Jeśli koniecznie chcemy wywołać funkcję z klasy podstawowej musimy posłużyć się operatorem zakresu (linia 19). Lepiej jednak nadawać takie nazwy, które nie istnieją w klasie podstawowej.

```
1  #include <iostream>
2
3  class Punkt2D
4  {
5     private:
6     double x = 0, y = 0;
7   public:
8     void pokaz() {cout << x << ", " << y << endl;}
9  };
10 class LepszyPunkt2D : public Punkt2D
11 {
12    public:
13    void pokaz() {cout << "Pracuje funkcja pokaz z klasy
     LepszyPunkt2D!" << endl;}
14 };
15 int main()
16 {
17   LepszyPunkt2D p;
18   p.pokaz();           // pokaz() z LepszyPunkt2D
19   p.Punkt2D::pokaz(); // pozaz() z Punkt2D
20 }
```

Listing 8.2: Przykład zasłonięcia składnika w klasie pochodnej

8.4 Inicjalizacja w przypadku dziedziczenia

W przypadku dziedziczenia obiekt klasy pochodnej składa się jakby z dwóch części: odziedziczonej i nowej (dodanej w klasie pochodnej). W związku z tym

przy tworzeniu obiektów klasy pochodnej najpierw trzeba zbudować część ódziedziczoną", a następnie nową. Można w tym celu postąpić w sposób jaki znamy do tej pory, a więc napisać odpowiedni konstruktor, który będzie inicjalizował wszystkie składniki z części odziedziczonej jak i ńowej". Jednak jest to nieporęczne. Ponadto, co w przypadku gdy klasa podstawowa to np. 10 klasa w hierarchii? Jak wtedy zainicjalizować te wszystkie składniki? Teoretycznie jest to możliwe, ale czasem po prostu nie mamy dostępu do tych danych.

Postępujemy, więc inaczej wykorzystując konstruktor klasy podstawowej, który umieszczamy na liście inicjalizacyjnej klasy pochodnej. Jego zadaniem jest inicjalizacja części odziedziczonej. Resztę, czyli ńowąćęść inicjalizujemy umieszczając je za konstruktorem z klasy podstawowej (lub w ciele konstruktora klasy pochodnej). Listing 8.3 pokazuje przykład inicjalizacji obiektu klasy pochodnej przy wykorzystaniu konstruktora klasy podstawowej.

```
1  #include <iostream>
2
3  class Figura
4  {
5  private:
6      double pole;
7  public:
8      Figura() : pole(0) {}
9      void pokazPole() {cout << pole << endl;}
10 };
11 class Kwadrat : public Figura
12 {
13 private:
14     double bok;
15 public:
16     Kwadrat() : Figura(), bok(0) {}
17 };
18 int main()
19 {
20     Figura fig;
21     fig.pokazPole();
22 }
```

Listing 8.3: Uruchomienie konstruktora klasy podstawowej na liście inicjalizacyjnej klasy pochodnej

8.5 Dziedziczenie i zawieranie obiektów

Wiemy, że składnikami klasy mogą być też inne klasy. Np. obiekt klasy `Dom` może się składać (zawierać w sobie) takie obiekty (klasy) jak: pokój, szafa, kuchnia,... Jest to tzw. relacja zawierania (ang. *has-a relationship*). Dzięki temu możemy wykorzystywać wielokrotnie klasy, które definiujemy tylko raz. Drugim sposobem użycia klas, które już wcześniej zostały zdefiniowane jest dziedziczenie. Jeśli jakaś klasa jest szczególnym rodzajem jakiejś innej klasy to możemy od niej

dziedziczyć tworząc nową klasę, a więc wykorzystać klasę już zdefiniowaną. Dziedziczenie wprowadza tzw. relację "jest rodzaju"(ang. *is-a relationship*).

Zapamiętaj

Dziedziczenie stosujemy, gdy możemy powiedzieć, że obiekt klasy pochodnej jest szczególnym rodzajem klasy podstawowej.

Zawieranie obiektu klasy B w klasie A stosujemy, gdy możemy powiedzieć, że klasa A składa się między innymi z obiektów klasy B.

8.6 Konwersje standardowe przy dziedziczeniu

Poniższe informacje będą dla nas przydatne przy następnym rozdziale na temat polimorfizmu.

Zapamiętaj

Wskaźnik do obiektu klasy pochodnej może być niejawnie przekształcony na wskaźnik do obiektu klasy podstawowej.

Oznacza to np., że jeśli mamy klasę podstawową `A` oraz funkcję, która jako argumentu spodziewa się wskaźnika do tej klasy podstawowej, to możemy do tej funkcji wysłać także wskaźnik do klasy pochodnej od `A`. Zostanie on wtedy niejawnie przekształcony na wskaźnik do `A` i funkcja zadziała poprawnie.

Podobny mechanizm działa w przypadku referencji.

Zapamiętaj

Referencja obiektu klasy pochodnej może być przypisana do referencji klasy podstawowej.

Konwersje te zostały wprowadzone min. po to by rozszerzyć użyteczność pisanego kodu - możemy wykorzystywać kod, który nie był napisany dla naszej klasy. Warunkiem tego jest, aby nasza klasa dziedziczyła od klasy, dla której ten kod został napisany. Ponadto takie konwersje naśladują też rzeczywistość. Np. na obiekt typu `Pies` mogę powiedzieć też `Zwierze` (czyli mówiąc językiem C++ zachodzi niejawna konwersja referencji do `Psa` na referencję do `Zwierze`).

8.7 Zadania do rozwiązania na zajęciach

Zad. 1. Stwórz klasę `FiguraPlaska` posiadającą prywatne składowe obwód i pole. Napisz klasy pochodne klasy `FiguraPlaska` służące do przechowywania danych różnych konkretnych figur. Klasy pochodne powinny posiadać prywatne

pola służące do przechowywania ich wymiarów. Wprowadź przynajmniej, kwadrat, prostokąt, trójkąt, okrąg i trapez. W funkcji `main` utwórz obiekt każdego rodzaju.

Zad. 2. Do klas z zadania nr 1 dodaj publiczne funkcje składowe:

- `ustawPole` wpisującą daną pole (klasa bazowa)
- `ustawObwod` wpisującą daną obwód (klasa bazowa)
- `ustawWymiary` wpisującą wymiary dla danej figury (klasa bazowa i pochodne)
- `pokaz` wypisującą dane składowe figury (klasa bazowa i pochodne).

W funkcji `main` wywołaj funkcje dla wszystkich obiektów.

Zad. 3. Do zadania nr 2 dodaj dwa konstruktory (domyślny i z odpowiednimi argumentami) oraz destruktor dla wszystkich klas. W konstruktorach i destruktorach dodaj kod wypisujący jaki konstruktor/destruktor aktualnie pracuje. W funkcji `main` utwórz obiekty każdego typu.

Zad. 4. Do zadania nr 2 dodaj klasę `TrojkatRownoboczny` dziedziczącą od klasy `Trojkat`. Zdefiniuj na nowo funkcję `pokaz` tak, aby w niej najpierw była wywołana funkcja `pokaz` z klasy `FiguraPlaska`, a potem nastąpiło wypisanie dodatkowej zmiennej z klasy `TrojkatRownoboczny`. W funkcji `main` utwórz obiekt typu `TrojkatRownoboczny` i wywołaj funkcję `pokaz`.

8.8 Zadania do rozwiązania samodzielnie

Zadania na ocenę 3.0

Zad. 1. Stwórz klasę `Rower` posiadającą prywatne składowe nazwę i kolor. Napisz klasy pochodne klasy `Rower` służące do przechowywania danych różnych konkretnych rowerów. Klasy pochodne powinny posiadać prywatne pola służące do przechowywania liczby przerzutek danego roweru. Wprowadź przynajmniej rower: górski, miejski, trekkingowy. W funkcji `main` utwórz obiekt każdego rodzaju.

Zad. 2. Do zadania nr 1 dodaj publiczne funkcje składowe:

- `ustawKolor` pozwalającą na wpisanie koloru roweru (klasa bazowa)
- `ustawNazwe` pozwalającą na wpisanie nazwy roweru (klasa bazowa)
- `ustawLiczbePrzerzutek` wypisującą liczbę przerzutek dla dla danego roweru (klasa bazowa i pochodne)

- `pokaz` pozwalającą na wypisanie danych składowych danego roweru (klasa bazowa i pochodne)

W funkcji `main` wywołaj funkcje dla wszystkich obiektów.

Zadanie na ocenę 4.0

Zad. 3. Do zadania nr 2 dodaj dwa konstruktory (domyślny i z odpowiednimi argumentami) oraz destruktor dla wszystkich klas. W konstruktorach i destruktorach dodaj kod wypisujący jaki konstruktor/destruktor aktualnie pracuje. W funkcji `main` utwórz obiekty każdego typu.

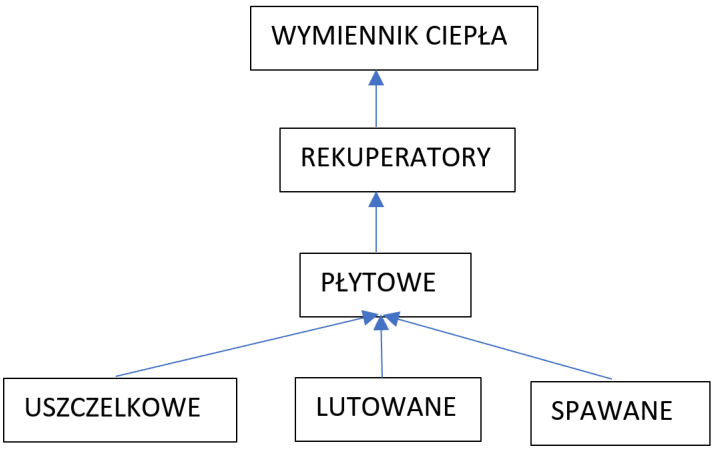
Zadanie na ocenę 5.0

Zad. 4. Do zadania nr 3 dodaj klasę `RowerMiejskiElektryczny` dziedziczącą od klasy `RowerMiejski`. Zdefiniuj na nowo funkcję `pokaz` tak, aby w niej najpierw była wywołana funkcja `pokaz` z klasy `Rower`, a potem nastąpiło wypisanie dodatkowej zmiennej z klasy `RowerMiejskiElektryczny`. W funkcji `main` utwórz obiekt typu `RowerMiejskiElektryczny` i wywołaj funkcję `pokaz`.

Zadania nieobowiązkowe

Zad. 5. Utwórz hierarchię klas przedstawioną na rys. 8.1. Napisz funkcję, `podajMoc`, która jako argument będzie przyjmowała wskaźnik lub referencję do klasy `WymiennikCiepla`. Funkcja ma zwracać wartość mocy dla danego wymiennika ciepła. Wywołaj ją ze wskaźnikiem (lub referencją) do klasy `WymiennikCiepla` oraz `UszczelkowyWymiennikCiepla`. Wartość mocy wymiennika powinna być obliczana ze wzoru (8.1). Zakładamy, że wartości strumienia masy $\dot{m}$, ciepła właściwego czynnika roboczego c_p oraz spadku temperatury w danym wymienniku ciepła są znane i przechowywane w polach `protected` klasy `WymiennikCiepla`.

$$\dot{Q} = \dot{m} c_p \Delta t \tag{8.1}$$



Rys. 8.1 Hierarchia klas dla płytowych wymienników ciepła