17

Funkcje i klasy zaprzyjaźnione

Prywatne elementy członkowskie klasy (np. prywatne zmienne członkowskie) są dostępne w sposób bezpośredni wyłącznie w obrębie klasy, w której zostały zdefiniowane. Natomiast chronione elementy członkowskie są dostępne bezpośrednio zarówno wewnątrz klasy, w której zostały zdefiniowane, jak i w jej klasach pochodnych.

Zgodnie z zasadami implementacji mechanizmu hermetyzacji (ukrywania danych) dostęp do zmiennych prywatnych i chronionych z poziomu otoczenia wymienionych wcześniej klas można zrealizować za pomocą publicznych metod dostępowych — instancyjnych setterów i getterów zdefiniowanych w tych klasach.

Prywatne elementy członkowskie klasy mogą być przetwarzane w jej otoczeniu także za pośrednictwem jej „przyjaciół" (ang. *friends*). To samo dotyczy możliwości przetwarzania chronionych elementów członkowskich klas wchodzących w skład łańcucha dziedziczenia na poziomie otoczenia tej hierarchii klas.

Wspomnianymi powyżej przyjaciółmi danej klasy (lub klas należących do łańcucha dziedziczenia) mogą być tzw. funkcje zaprzyjaźnione oraz klasy zaprzyjaźnione.

17.1. Funkcje zaprzyjaźnione

Funkcje zaprzyjaźnione (ang. *friend functions*) zapewniają dostęp do prywatnych członków klasy, tak samo jak metody (funkcje) członkowskie należące do tej klasy. To samo dotyczy chronionych elementów członkowskich, które z definicji są dostępne w klasie bazowej (czyli tej, w której zostały zadeklarowane) oraz w jej klasach pochodnych.

Funkcję zaprzyjaźnioną deklaruje się za pomocą prototypu w obrębie klasy, której przyjacielem jest ta funkcja. W tym celu wspomnianą deklarację należy poprzedzić słowem kluczowym `friend`. Na przykład wyrażenie: `friend double getPromienFriendKolo(Kolo);` oznacza deklarację funkcji zaprzyjaźnionej o nazwie `getPromienFriendKolo`, która ma jeden para-

metr wejściowy typu `Kolo` i zwraca na zewnątrz wartość typu `double`. Przy tym jest zupełnie obojętne, czy deklaracja funkcji zaprzyjaźnionej znajduje się w sekcji prywatnej (chronionej) klasy, czy też w sekcji publicznej.

Funkcja zaprzyjaźniona danej klasy musi mieć co najmniej jeden parametr/argument typu obiektowego. Powinien on stanowić instancję klasy, której ta funkcja jest przyjacielem. Wynika to bezpośrednio ze sposobu uzyskiwania dostępu do prywatnych (chronionych) zmiennych członkowskich klasy przez tę funkcję. Mianowicie ten dostęp jest realizowany za pośrednictwem obiektu, który jest argumentem wywołania funkcji zaprzyjaźnionej.

Definicja funkcji zaprzyjaźnionej może się znajdować w dowolnym miejscu programu — tak samo jak definicje „zwykłych" funkcji. Bardzo ważne jest to, że dana funkcja zaprzyjaźniona może być „przyjacielem" kilku klas jednocześnie.

Należy podkreślić, że funkcja zaprzyjaźniona nie jest elementem członkowskim klasy (ang. *class member*), w której została zadeklarowana. Tym samym nie może ona być wywoływana za pośrednictwem obiektu będącego instancją tej klasy.

W ogólności funkcje zaprzyjaźnione można definiować jako:

- zwykłe funkcje globalne, które nie należą do żadnej klasy, albo
- funkcje członkowskie (metody) należące do innej klasy.

W praktyce wykorzystanie funkcji zaprzyjaźnionych może wspomagać i rozszerzać implementację mechanizmu hermetyzacji — ukrywania danych.

Poniżej przedstawiono dwa przykłady. W pierwszym z nich (przykład 17.1) wykorzystano funkcje zaprzyjaźnione, które nie należą do żadnej klasy, a w drugim (przykład 17.2) — funkcje zaprzyjaźnione, które są elementami członkowskimi innej klasy

Przykład 17.1

```
#include <iostream>
#define _USE_MATH_DEFINES // w celu użycia stałej M_PI, która nie jest zdefiniowana w standardowym C/C++
#include <cmath>
using namespace std;
// Definicja klasy Kolo:
class Kolo {
    // Deklaracja zmiennej członkowskiej (domyślnie prywatnej):
    double _r;
public:
    // Definicje konstruktorów:
    Kolo() {}
    Kolo(double r) {_r = r;
    }
    // Deklaracje (prototypy) funkcji członkowskich:
    void setPromien(double); // setter
```

```
    double getPromien(); // getter
    double pole();
    double obwod();
    // Prototypy funkcji zaprzyjaźnionych:
    friend void setPromienFriendKolo(Kolo&, double);
    friend double getPromienFriendKolo(Kolo);
    /* UWAGA
     * Funkcja zaprzyjaźniona musi mieć co najmniej jeden parametr typu obiektowego należący do klasy,
     * której jest przyjacielem.
     * Wynika to z faktu, że funkcja zaprzyjaźniona uzyskuje dostęp do prywatnych zmiennych członkowskich klasy
     * za pośrednictwem obiektu będącego jej instancją, który jest argumentem jej wywołania.
     */
};
// Definicje metod członkowskich klasy Kolo:
void Kolo::setPromien(double r)    {
    _r = r;
}
double Kolo::getPromien() {
    return _r;
}
double Kolo::pole() {
    return M_PI * _r * _r;
}
double Kolo::obwod() {
    return 2 * M_PI * _r;
}
// Definicje funkcji zaprzyjaźnionych klasy Kolo:
void setPromienFriendKolo(Kolo &kolo, double r)    {
    kolo._r = r;
}
double getPromienFriendKolo(Kolo kolo) {
    return kolo._r;
}
/* UWAGA
 * Funkcje zaprzyjaźnione zostały zdefiniowane tutaj jako zwykłe funkcje globalne, które nie należą do żadnej klasy.
 * Definicje te mogą się znajdować w dowolnym miejscu programu.
 */

int main() {
    // Utworzenie obiektu kolo1 klasy Kolo:
    Kolo kolo1; // niejawne wywołanie konstruktora domyślnego
    // OBSŁUGA OBIEKTU ZA POMOCĄ METOD INSTANCYJNYCH
    // Nadanie wartości zmiennej prywatnej _r — wykorzystanie settera setPromien():
    kolo1.setPromien(1);
```

```
    // Prezentacja wartości zmiennej _r — wywołanie gettera getPromien():
    cout << "Promień koła: " << kolo1.getPromien() << endl;
    // Wyświetlenie wartości pola i obwodu koła kolo1:
    cout << "Pole wynosi " << kolo1.pole() << endl;
    cout << "Obwod wynosi " << kolo1.obwod() << endl;

    // Utworzenie obiektu kolo2 klasy Kolo:
    Kolo kolo2; // niejawne wywołanie konstruktora domyślnego
    // OBSŁUGA OBIEKTU ZA POMOCĄ FUNKCJI ZAPRZYJAŹNIONYCH
    // Nadanie wartości zmiennej prywatnej _r — wykorzystanie funkcji zaprzyjaźnionej setPromienFriendKolo():
    setPromienFriendKolo(kolo2, 2);
    /* UWAGA
     * Obiekt kolo2 jest argumentem wywołania funkcji zaprzyjaźnionej setPromienFriendKolo().
     * Dostęp do prywatnej zmiennej członkowskiej _r klasy Kolo uzyskuje się za pośrednictwem tego właśnie obiektu.
     */
    // Prezentacja wartości zmiennej _r — wykorzystanie funkcji zaprzyjaźnionej getPromienFriendKolo():
    cout << "Promień koła: " << getPromienFriendKolo(kolo2) << endl;
    /* UWAGA
     * Obiekt kolo2 jest argumentem wywołania funkcji zaprzyjaźnionej getPromienFriendKolo().
     */
    // Wyświetlenie wartości pola i obwodu koła kolo1:
    cout << "Pole wynosi " << kolo2.pole() << endl;
    cout << "Obwod wynosi " << kolo2.obwod() << endl;

    return 0;
}
```

W programie zdefiniowano klasę `Kolo` zawierającą deklarację prywatnej zmiennej członkowskiej `_r`. Dostęp do tej zmiennej spoza klasy `Kolo` (czyli z poziomu otoczenia klasy `Kolo`) można uzyskać za pośrednictwem publicznych metod dostępowych tej klasy — settera `setPromien()` i gettera `getPromien()`.

Ponadto w klasie `Kolo` zadeklarowano przy użyciu słowa kluczowego `friend` dwie funkcje będące przyjaciółmi tej klasy: `setPromienFriendKolo()` i `getPromienFriendKolo()`. Funkcjonalność funkcji zaprzyjaźnionej `setPromienFriendKolo()` odpowiada funkcjonalności settera `setPromien()`, a funkcji `setPromienFriendKolo()` — gettera `getPromien()`. Każda z wymienionych funkcji zaprzyjaźnionych ma parametr należący do typu obiektowego (klasy) `Kolo`.

Należy podkreślić, że zdefiniowane tutaj funkcje zaprzyjaźnione nie należą do żadnej klasy.

Funkcje zaprzyjaźnione `setPromienFriendKolo()` i `getPromienFriendKolo()` są wywoływane w programie w sposób bezpośredni — bez konieczności użycia obiektu będącego instancją klasy `Kolo` jako pośrednika — jak w przypadku settera `setPromien()`i gettera `getPromien()`.

Z drugiej strony w każdym wywołaniu funkcji `setPromienFriendKolo()` i `getPromienFriendKolo()` występuje argument `kolo2`, który jest obiektem będącym instancją klasy `Kolo`. Obiekt

ten odgrywa rolę pośrednika w realizacji dostępu do prywatnej zmiennej członkowskiej `_r` klasy `Kolo` za pomocą funkcji zaprzyjaźnionych.

Omawiane tutaj funkcje zaprzyjaźnione rozszerzają interfejs klasy `Kolo` — stanowią komponent zewnętrzny tego interfejsu. Wspomagają one implementację mechanizmu hermetyzacji — ukrywania danych w klasie `Kolo`.

Pole i obwód są obliczane w programie dwukrotnie: pierwszy raz w odniesieniu do koła reprezentowanego przez obiekt `kolo1`, a drugi — do koła `kolo2`.

Ćwiczenie 17.1

Zmodyfikuj program zawarty w przykładzie 17.1 — zamiast klasy `Kolo` zdefiniuj klasę `Prostokat`. Klasa `Prostokat` powinna zawierać niezbędne deklaracje zmiennych i funkcji członkowskich pozwalających obliczyć pole i obwód prostokąta. Dane (parametry) prostokąta powinny być hermetyzowane (ukryte) przed światem zewnętrznym. Ponadto zdefiniuj zestaw funkcji zaprzyjaźnionych klasy `Prostokat`, których zadaniem jest wspomaganie i rozszerzenie implementacji mechanizmu hermetyzacji danych prostokąta. Zdefiniuj wspomniane funkcje jako globalne — niezależne od klasy `Prostokat`, czyli niebędące jej metodami.

Przykład 17.2

```
#include <iostream>
#define _USE_MATH_DEFINES // w celu użycia stałej M_PI, która nie jest zdefiniowana w standardowym C/C++
#include <cmath>
using namespace std;

// Deklaracja klasy Promien:
class Promien;

// Definicja klasy Kolo:
class Kolo {
public:
    // Deklaracje metod publicznych:
    double pole(Promien pPromien);
    double obwod(Promien pPromien);
    /* UWAGA
     * Metody pole() i obwod() stanowią funkcje zaprzyjaźnione klasy Promien.
     */
};

// Definicja klasy Promien:
class Promien {
private:
    double _r; // deklaracja prywatnej zmiennej członkowskiej
public:
```

```
    // Definicje publicznych metod dostępowych do zmiennej prywatnej _r:
    void setPromien(double r) { // setter
        _r = r;
    }
    double getPromien() { // getter
        return _r;
    }

    // Deklaracje (prototypy) funkcji zaprzyjaźnionych:
    friend double Kolo::pole(Promien);
    friend double Kolo::obwod(Promien);
    /* UWAGA
     * Zgodnie z deklaracjami powyżej definicje funkcji pole() i obwod() znajdują się w klasie Kolo.
     */
};
// Definicje metod pole() i obwod() z klasy Kolo:
double Kolo::pole(Promien pPromien) {
    return M_PI * pPromien._r * pPromien._r;
}
double Kolo::obwod(Promien pPromien) {
    return 2 * M_PI * pPromien._r;
}

int main() {
    // Utworzenie obiektu promien będącego instancją klasy Promien:
    Promien promien;
    promien.setPromien(1);

    // Utworzenie obiektu kolo należącego do klasy Kolo:
    Kolo kolo;
    // Obliczenie i wyświetlenie pola i obwodu koła:
    cout << "Pole wynosi " << kolo.pole(promien) << endl;
    cout << "Obwod wynosi " << kolo.obwod(promien) << endl;
    /* UWAGA
     * Powyżej wywołano metody pole() i obwod(), które stanowią zaprzyjaźnione klasy Promien.
     */
    return 0;
}
```

Funkcjonalność programu zawartego w niniejszym przykładzie jest podobna do funkcjonalności programu z przykładu 17.1. Jednakże tutaj obliczane są pole i obwód koła nie dla dwóch wartości promienia, a dla jednej.

Najważniejsza różnica pomiędzy programem zaprezentowanym tutaj a programem z przykładu 17.1 tkwi w ich implementacji. Mianowicie tutaj zdefiniowano dwie klasy: `Promien` i `Kolo`.

Klasa `Promien` zawiera deklarację prywatnej zmiennej członkowskiej `_r` oraz definicje publicznych metod dostępowych: settera `setPromien()` i gettera `getPromien()`. Oprócz tego w klasie `Promien` zamieszczono deklaracje (prototypy) dwóch funkcji zaprzyjaźnionych: `pole()` i `obwod()`. Tym samym wymienione funkcje to funkcje zaprzyjaźnione klasy `Promien`. Należy zwrócić uwagę na to, że funkcje `pole()` i `obwod()` nie są metodami instancyjnymi klasy `Promien`.

W klasie `Kolo` zdefiniowano dwie metody: `pole()` i `obwod()`, których zadaniem jest obliczenie, odpowiednio, pola i obwodu koła. Metody te są elementami członkowskimi klasy `Kolo` i jednocześnie — o czym już wspomniano — funkcjami zaprzyjaźnionymi klasy `Promien`.

Ćwiczenie 17.2

Zmodyfikuj program zawarty w przykładzie 17.2 — zdefiniuj dodatkową klasę `Kula` zawierającą metody `objetosc()` i `polePowierzchni()`, które umożliwią obliczenie, odpowiednio, objętości i pola powierzchni kuli. W definicji klasy `Promien` zadeklaruj funkcje zaprzyjaźnione `objetosc()` i `polePowierzchni()`, które zostały zdefiniowane w klasie `Kula`, jako jej funkcje członkowskie. Wykonaj testy poprawności działania programu dla kuli o zadanym promieniu `1`.

17.2. Klasy zaprzyjaźnione

W ogólności **klasa zaprzyjaźniona** (ang. *friend class*) to klasa, której obiekty (jako jej instancje) mogą uzyskać dostęp do prywatnych i chronionych elementów członkowskich innej klasy — tej, w której została zadeklarowana jako jej „przyjaciel” (ang. *friend*).

Tym samym prywatne i chronione składniki określonej klasy mogą być przetwarzane na poziomie jej otoczenia nie tylko za pomocą publicznych metod członkowskich tej klasy — setterów i getterów, ale również za pośrednictwem metod należących do jej klasy zaprzyjaźnionej.

Bardzo ważną cechą mechanizmu „przyjaźni” pomiędzy klasami jest to, że dana klasa może mieć wiele klas zaprzyjaźnionych.

Wykorzystując w praktyce omówiony powyżej mechanizm przyjaźni pomiędzy klasami, należy pamiętać o kilku ważnych zasadach, mianowicie:

- Przyjaźń nie podlega dziedziczeniu.
- Przyjaźń pomiędzy klasami nie jest automatycznie wzajemna — dwukierunkowa (ang. *mutual*). Oznacza to, że jeżeli klasa A jest przyjacielem klasy B, to odwrotna relacja przyjaźni nie zachodzi w sposób automatyczny.
- Zadeklarowanie zbyt wielu relacji przyjaźni pomiędzy klasami może znacznie skomplikować i zaciemnić zaimplementowany mechanizm hermetyzacji danych w tych klasach.

Przykład 17.3

```
#include <iostream>
#define _USE_MATH_DEFINES // w celu użycia stałej M_PI, która nie jest zdefiniowana w standardowym C/C++
#include <cmath>
using namespace std;

// Definicja klasy Promien:
class Promien {
    // Deklaracja prywatnej zmiennej członkowskiej:
    double _r;
public:
    // Prototypy publicznych funkcji dostępowych:
    void setPromien(double); // setter
    double getPromien(); // getter
    // Deklaracje klas zaprzyjaźnionych:
    friend class Kolo;
    friend class Kula;
    /* UWAGA
     * Powyższe deklaracje z użyciem słowa kluczowego friend skutkują ustaleniem relacji przyjaźni pomiędzy klasą
     * Promien a klasami Kolo i Kula. Klasy Kolo i Kula stanowią klasy zaprzyjaźnione klasy Promien.
     */
};
// Definicje metod klasy Promien:
void Promien::setPromien(double r)    {
    _r = r;
}
double Promien::getPromien() {
    return _r;
}
// Definicja klasy Kolo:
class Kolo {
/* UWAGA
 * Klasa Kolo jest klasą zaprzyjaźnioną klasy Promien.
 */
public:
    double pole(Promien);
    double obwod(Promien);
};
// Definicje metod klasy Kolo:
double Kolo::pole(Promien promien) {
    return M_PI * promien._r * promien._r;
    /* UWAGA
     * W ciele funkcji członkowskiej pole() należącej do klasy Kolo, która jest „przyjacielem" klasy Promien,
     * wykorzystano prywatną zmienną członkowską _r zdefiniowaną w klasie Promien.
     */
}
```

```
double Kolo::obwod(Promien promien) {
    return 2 * M_PI * promien._r;
    /* UWAGA
     * Metoda obwod() z klasy Kolo, będącej „przyjacielem" klasy Promien, wykorzystuje prywatną zmienną
     * członkowską _r zdefiniowaną w klasie Promien.
     */
}
// Definicja klasy Kula, która jest zaprzyjaźniona z klasą Promien:
class Kula {
public:
    double objetosc(Promien);
    double pole(Promien);
};
// Definicje metod klasy Kula:
double Kula::objetosc(Promien promien) {
    return double(4)/double(3) * M_PI * promien._r * promien._r * promien._r;
    /* UWAGA
     * W treści funkcji członkowskiej objetosc() należącej do klasy Kula (która jest „przyjacielem" klasy Promien)
     * wykorzystano prywatną zmienną członkowską _r zdefiniowaną w klasie Promien.
     */
}
double Kula::pole(Promien promien) {
    return 4 * M_PI * promien._r * promien._r;
}

int main() {
    // Utworzenie obiektu promien jako instancji klasy Promien:
    Promien promien;
    // Utworzenie obiektu kolo:
    Kolo kolo; // Obiekt kolo jest instancją klasy Kolo, która jest „przyjacielem" klasy Promien.
    // Ustalenie promienia koła na 1:
    promien.setPromien(1); // wywołanie metody instancyjnej klasy Promien
    // Obliczenie i prezentacja pola i obwodu koła dla zadanego promienia:
    cout << "Pole koła wynosi: " << kolo.pole(promien) << endl;
    cout << "Obwód koła wynosi: " << kolo.obwod(promien) << endl;
    /* UWAGA
     * Za pośrednictwem metod pole() i obwod() obiektu kolo, będącego instancją klasy Kolo (która jest klasą
     * zaprzyjaźnioną klasy Promien), uzyskano dostęp do prywatnej zmiennej członkowskiej _r zdefiniowanej
     * w klasie Promien.
     * Należy zwrócić uwagę, że argument wywołania metod pole() i obwod() jest obiektem promien klasy Promien.
     */

    // Utworzenie obiektu kula:
    Kula kula; // obiekt kula jest instancją klasy Kula, która jest „przyjacielem" klasy Promien
    promien.setPromien(2); // wywołanie metody instancyjnej klasy Promien
    cout << "Objętość kuli wynosi: " << kula.objetosc(promien) << endl;
    cout << "Pole powierzchni kuli wynosi: " << kula.pole(promien) << endl;
```

```
    /* UWAGA
     * Dostęp do prywatnej zmiennej członkowskiej _r zdefiniowanej w klasie Promien uzyskano za pomocą metod
     * objetosc() i pole() obiektu kula, będącego instancją klasy Kula. Przy tym klasa Kula jest klasą
     * zaprzyjaźnioną klasy Promien.
     * Obiektem promien klasy Promien jest argument wywołania metod objetosc() i pole().
     */

    return 0;
}
```

W programie zdefiniowano klasę `Promien`, która zawiera deklarację prywatnej zmiennej członkowskiej `_r`. Dostęp do tej zmiennej z poziomu otoczenia klasy `Promien` można uzyskać za pomocą publicznych metod dostępowych należących do tej klasy: settera `setPromien()` i gettera `getPromien()`.

Oprócz tego definicja klasy `Promien` zawiera deklaracje dwóch klas zaprzyjaźnionych, `Kolo` i `Kula`: `friend class Kolo; friend class Kula;`. Oznacza to, że dostęp do prywatnej zmiennej `_r` należącej do klasy `Promien` można zrealizować również za pośrednictwem publicznych metod należących do wymienionych klas zaprzyjaźnionych, `Kolo` i `Kula`. W szczególności dostęp ten uzyskano za pośrednictwem metod `pole()` i `obwod()` — elementów członkowskich klasy `Kolo`, oraz metod `objetosc()` i `pole()` — elementów członkowskich klasy `Kula`.

Argumentem wywołania każdej z wymienionych metod jest obiekt `promien` należący do klasy `Promien`.

Ćwiczenie 17.3

Zmodyfikuj program zawarty w przykładzie 17.3 — zamiast relacji przyjaźni pomiędzy klasą `Promien` a klasami `Kolo` i `Kula` wykorzystaj mechanizm dziedziczenia. Załóż, że klasy `Kolo` i `Kula` to klasy pochodne klasy `Promien`, która odgrywa rolę klasy bazowej. Uwzględnij hermetyzację — ukrycie danych w klasie `Promien`. Przeprowadź analizę porównawczą wykonanego programu z programem z przykładu 17.3. Omów wyniki tej analizy.

17.3. Pytania i zadania kontrolne

17.3.1. Pytania

1. Podaj definicję funkcji zaprzyjaźnionej. W jakim celu wykorzystuje się w praktyce funkcje zaprzyjaźnione?
2. Co oznacza termin „klasa zaprzyjaźniona”?
3. Czy relacje przyjaźni pomiędzy klasami podlegają dziedziczeniu?
4. Omów wykorzystanie relacji przyjaźni zachodzącej pomiędzy:
   - funkcją a klasą,
   - pomiędzy klasami

   w implementacji mechanizmu hermetyzacji — ukrywania danych.

17.3.2. Zadania

1. Napisz program pozwalający obliczyć objętość, pole powierzchni całkowitej oraz długości wszystkich krawędzi prostopadłościanu. Dane wejściowe mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Wykorzystaj klasę `Prostopadloscian` zawierającą deklaracje i definicje wymaganych komponentów. Uwzględnij hermetyzację — ukrycie danych prostopadłościanu. Dostęp do prywatnych danych prostopadłościanu uzyskaj za pośrednictwem funkcji zaprzyjaźnionych z klasą `Prostopadloscian`. Przy czym wspomniane funkcje zaprzyjaźnione zdefiniuj jako funkcje globalne — niezależne od klasy `Prostopadloscian` (nienależące do żadnej klasy). Wykonaj testy działania programu dla prostopadłościanu o parametrach: długość podstawy `1`, szerokość podstawy `2`, wysokość `3`.
2. Zrób tak jak w zadaniu 1., z tym że funkcje zaprzyjaźnione z klasą `Prostopadloscian` zdefiniuj jako metody instancyjne innej klasy.
3. Napisz program pozwalający przetwarzać dane pracownika i ucznia szkoły. Uwzględnij następujące dane pracownika: imię, nazwisko, stanowisko oraz następujące dane ucznia: imię, nazwisko, klasę. Wykorzystaj zdefiniowane samodzielnie klasy: `Osoba`, `Pracownik` i `Uczen`, które nie będą ze sobą powiązane żadnymi relacjami dziedziczenia. Klasy `Pracownik` i `Uczen` powinny być klasami zaprzyjaźnionymi klasy `Osoba`. Zastosuj mechanizm hermetyzacji — ukrywania danych w poszczególnych klasach. Dostęp do prywatnych danych klasy `Osoba` zrealizuj za pośrednictwem publicznych metod członkowskich należących do klas zaprzyjaźnionych — `Pracownik` i `Uczen`. Wykonaj testy działania programu dla przykładowych danych pracownika i ucznia.
4. Zrób tak jak w zadaniu 3., z tym że zamiast relacji przyjaźni pomiędzy klasą `Osoba` a klasami `Pracownik` i `Uczen` wykorzystaj mechanizm dziedziczenia — dziedziczenie hierarchiczne. Załóż, że klasą bazową jest klasa `Osoba`, a jej klasy pochodne to `Pracownik` i `Uczen`. Dostęp do chronionych danych w poszczególnych klasach zrealizuj za pomocą funkcji zaprzyjaźnionych tych klas.