Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

III. Dziedziczenie i polimorfizm. Implementacja paradygmatów w języku C++

1. Dziedziczenie - jego schematy i tryby

1.1. Relacje zachodzące pomiędzy obiektami występującymi w otaczającej nas rzeczywistości w wielu wypadkach przybierają postać "specjalizacji". Przykładowo, taki łańcuch relacyjny tworzą:

pojazd_silnikowy → samochód → samochód_osobowy →
Toyota → Toyota_Auris_Hybrid

Implementację tych powiązań w językach obiektowych umożliwia paradygmat **dziedziczenia**. Dziedziczenie to relacja zachodząca pomiędzy zdefiniowaną pierwotnie **klasą bazową** (i jej obiektami) oraz stanowiącą jej uszczegółowienie **klasą pochodną** (i jej obiektami).

1.2. Klasa pochodna dziedziczy składowe (dane, struktury danych i metody) po swej klasie bazowej i, dodatkowo, definiuje własne składowe. Dziedziczenie odbywa się według ściśle określonych reguł, wymuszonych tzw. **trybem dziedziczenia**. Nagłówek klasy *A* pochodnej względem klasy bazowej *B* ma postać:

`class` *id-klasy-A* `:` *tryb-dziedziczenia id-klasy-B*

gdzie *tryb-dziedziczenia,* zadawany za pomocą słów kluczowych `public`, `protected` i `private` oznacza, odpowiednio:

- `public`: przejęcie składowych ze wszystkich sekcji (`public`, `protected`, `private`) z zachowaniem dotychczasowych praw dostępu,
- `protected`: przejęcie składowych z sekcji `protected` i `private` z zachowaniem dotychczasowych praw dostępu, oraz składowych `public` ze zmianą dotychczasowego dostępu na `protected`,
- `private`: przejęcie składowych z sekcji `private` z zachowaniem dotychczasowych praw dostępu, oraz składowych `public` i `protected` ze zmianą dotychczasowego dostępu na `private`.

1.3. Rozważmy przykład. Przyjąwszy definicję typu wyliczeniowego naped:

```
typedef enum naped {przednie, tylne, prz_tyl};
```

możemy zdefiniować następującą klasę Samochod1:

```
class Samochod1
{public:
   int poj_silnika;
   naped naped_kola;
 protected:
   int max_predkosc;
   string numer_rejestr = "";
 private:
   char *numer_silnika = nullptr;
 public:
   Samochod1(int psil, naped kola, int pred)
   {poj_silnika = psil;
    naped_kola = kola;
    max_predkosc = pred;}
   ~Samochod1()
   {delete [] numer_silnika;}
   void Zmien_rejestracje(string nowy_numer)
   {numer_rejestr = nowy_numer;}
 //...definicje pozostalych metod składowych
};
```

Klasa bazowa Samochod1 może stanowić punkt wyjścia do zdefiniowania klasy pochodnej Samochod_osob:

```
class Samochod_osob : public Samochod1
{public:
   int liczba_osob;
 protected:
   int liczba_poduszek;
 public:
   Samochod_osob
          (int psil, naped kola, int pred,
           int osoby, int poduszki):
           Samochod1(psil,kola,pred),
           liczba_osob(osoby),
           liczba_poduszek(poduszki) {};
   int Sprawdz_poduszki()
   {return liczba_poduszek;}
 //...definicje pozostalych pol i metod składowych
};
```

W powyższej klasie pochodnej mamy dostępne, między innymi, następujące pola składowe:

public: poj_silnika, naped_kola i liczba_osob,
protected: max_predkosc, numer_rejestr i liczba_poduszek
private: numer_silnika

i następujące funkcje składowe:

```
public:
void Zmien_rejestracje(char *nowy_numer)
int Sprawdz_poduszki()
```

Graficzną ilustrację budowy **obiektu** klasy Samochod_osob stanowi następujący rysunek:

1.4. Kolejną w łańcuchu dziedziczenia klasę `Toyota` można zdefiniować jak następuje:

```
class Toyota : public Samochod_osob
{protected:
   string model_Toyota;
 public:
   Toyota(int psil, naped kola, int pred,
         int osoby, int poduszki):
         Samochod_osob
               (psil,kola,pred,osoby,poduszki)
   {}
   void ustal_mod_Toyota(string model)
   {model_Toyota = model;}
 //...definicje pozostalych pol i metod składowych
};
```

Każdy obiekt tej klasy zawiera osiem pól składowych:

poj_silnika
naped_kola
max_predkosc
numer_rejestr
numer_silnika
liczba_osob
liczba_poduszek
model_Toyota

1.5. Prezentowane dotąd dziedziczenie ma charakter **pojedynczy**: ustala relację pomiędzy pewną (jedną!) klasą bazową i stanowiącą jej uszczegółowienie klasą pochodną. Tymczasem, w niektórych językach obiektowych dopuszcza się możliwość dziedziczenia nie z jednej, lecz z kilku (dwóch lub więcej) klas bazowych. Takie dziedziczenie, zwane **wielokrotnym**, ustala związek między zbiorem klas bazowych i ich klasą pochodną.

1.5.1. Przykładowo, rozważana klasa `Samochod1` może pozostawać w relacji dziedziczenia wielokrotnego z klasami bazowymi

Pojazd_Silnikowy oraz Srodek_Transportu. Załóżmy następujące definicje klas bazowych:

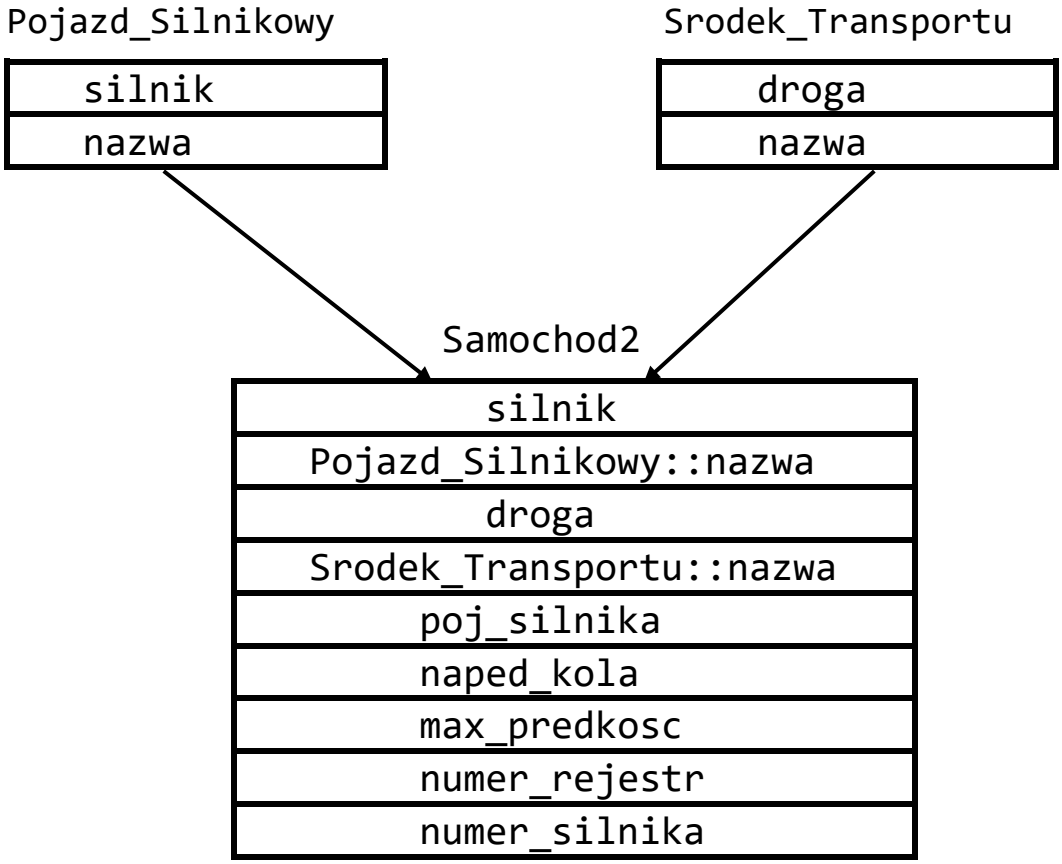
```
typedef enum rodz_silnika
{elektryczny,spalinowy,gazowy,
                  odrzutowy,parowy,wodny,wiatrowy};
typedef enum droga_transportu
{wodna,powietrzna,uliczna,kolejowa};

class Pojazd_Silnikowy
{public:
   rodz_silnika silnik;
   string nazwa;
   Pojazd_Silnikowy()
   {nazwa = "";}
 //definicje pozostalych pol oraz metod
 //skladowych klasy
}
class Srodek_Transportu
{public:
   droga_transportu droga;
   string nazwa;
   Srodek_Transportu()
   {nazwa = "";}
 //definicje pozostalych pol oraz metod
 //skladowych klasy
};
```

1.5.2. W tej nowej sytuacji, klasę Samochod1 zastąpimy poniższą klasą Samochod2:

```
class Samochod2 : public Pojazd_Silnikowy,
                  public Srodek_Transportu
{public:
   int poj_silnika;
   naped naped_kola;
 protected:
   int max_predkosc;
   string numer_rejestr = "";
 private:
   char *numer_silnika = nullptr;
 public:
   Samochod2(int psil, naped kola, int pred,
             rodz_silnika sil=elektryczny,
             droga_transportu drg=uliczna):
             poj_silnika(psil),
             naped_kola(kola),
             max_predkosc(pred)
   {silnik = sil;
    droga = drg;
   }
 //definicje pozostalych metod skladowych klasy}
};
```

Graficzną ilustrację zależności pomiędzy klasami `Pojazd_Silnikowy`, `Srodek_Transportu` i `Samochod2` obrazuje następujący rysunek:



1.5.3. Jeśli w zbiorze klas mamy do czynienia jedynie z dziedziczeniem pojedynczym, to dla zobrazowania hierarchii tych klas wystarczy posłużyć się strukturą drzewiastą (drzewo, las drzew). Jeśli natomiast występują w nim relacje dziedziczenia wielokrotnego, to dla ich zobrazowania konieczna jest struktura grafowa. Dowolną ścieżkę prowadzącą (w drzewie lub grafie) od klasy bazowej do klasy pochodnej nazywa się **ścieżką dziedziczenia**.

1.5.4. Jeśli klasa pochodna dziedziczy z dwóch różnych klas bazowych pola/metody o tym samym identyfikatorze, to w celu ich wskazania należy w klasie pochodnej posłużyć się operatorem zakresu `::` . Argumentami tego operatora są identyfikator klasy bazowej oraz identyfikator jej składowej. W takiej sytuacji nie ma możliwości klasycznego przeciążania dziedziczonych funkcji składowych.

1.5.5. Przykładowo, w zdefiniowanej wyżej klasie `Samochod2` może wystąpić metoda:

```
public:
  void Ustal_Nazwy(string naz1, string naz2)
  {Srodek_Transportu::nazwa = naz1;
   Pojazd_Silnikowy::nazwa = naz2;}
```

1.5.6. W wypadku dziedziczenia wielokrotnego może zdarzyć się sytuacja, w której pewna klasa wystąpi na dwóch różnych ścieżkach dziedziczenia prowadzących do tej samej klasy pochodnej. By zapobiec wielokrotnemu wprowadzaniu do klasy pochodnej tych samych składowych metod, należy rozważanej klasie bazowej nadać charakter abstrakcyjny.

2. Tworzenie, przetwarzanie i likwidacja obiektów klas pochodnych

2.1. W języku C++ obiekty klas pochodnych tworzy się, podobnie jak obiekty klas bazowych, w sposób statyczny lub dynamiczny. Do obiektu utworzonego statycznie można się odwoływać przez identyfikator lub referencję, ewentualnie – za pomocą wskaźnika, po wcześniejszym jego utworzeniu przy użyciu operatora adresu &. Do obiektu utworzonego dynamicznie można się odwoływać za pomocą wskaźnika, lub bezpośrednio – po uprzednim zastosowaniu operatora wyłuskania *, np.

```
//definicja klasy Samochod2 z jej konstruktorem
//Samochod2( int psil, naped kola, int pred,
//           rodz_silnika sil=elektryczny,
//           droga_transportu drg=uliczna):
//           poj_silnika(psil),
//           naped_kola(kola),
//           max_predkosc(pred)
//{silnik = sil;
// droga = drg;
//}
```

```
Samochod2 sam1(1500,przednie,190);
Samochod2 sam2 = Samochod2(2000,tylne,220);
Samochod2 *wsk_sam3 = &sam1;
Samochod2 &sam4 = sam2;
Samochod2 *wsk_sam5 = new
     Samochod2(1100,prz_tyl,200,gazowy,uliczna);
shared_ptr<Samochod2> wsk_sam6 =
     make_shared<Samochod2>(2500,prz_tyl,260,spalinowy)
```

2.2. Utworzenie obiektu klasy pochodnej polega na przydzieleniu obszaru pamięci dla reprezentacji tego obiektu (odziedziczone i własne pola składowe) oraz wywołaniu ciągu konstruktorów: poczynając od najstarszego w hierarchii dziedziczenia konstruktora klasy bazowej i kończąc na ostatnim w tej hierarchii konstruktorze klasy pochodnej.

2.3. W ogólności, w procesie tworzenia obiektu klasy pochodnej kolejne funkcje konstruktorów aktywowane są w sposób automatyczny. W wypadku jednak, **gdy w klasie bazowej są jedynie konstruktory z parametrami (brak konstruktorów domyślnych), należy wywołanie odpowiedniego konstruktora umieścić na liście inicjalizacyjnej**.

2.3.1. Przykładowo, konstruktor zdefiniowany w klasie `Samochod1` jest funkcją z trzema parametrami:

```
Samochod1(int psil, naped kola, int pred)
{poj_silnika = psil;
 naped_kola = kola;
 max_predkosc = pred;}
```

Na liście inicjalizacyjnej konstruktora klasy pochodnej:

```
class Samochod_osob : public Samochod1
```

musi więc wystąpić wywołanie konstruktora `Samochod1`:

```
Samochod_osob
          (int psil, naped kola, int pred,
           int osoby, int poduszki):
           Samochod1(psil,kola,pred),
           liczba_osob(osoby),
           liczba_poduszek(poduszki) {}
```

2.3.2. Przyjmując, że klasy bazowe `Pojazd_Silnikowy` i `Srodek_Transportu` mają konstruktory bezargumentowe, w klasie pochodnej:

```
class Samochod2 : public Pojazd_Silnikowy,
                  public Srodek_Transportu
```

możemy zdefiniować konstruktor poniższej postaci (z ukrytymi wywołaniami konstruktorów bezargumentowych):

```
Samochod2(int psil, naped kola, int pred,
          rodz_silnika sil=elektryczny,
          droga_transportu drg=uliczna) {//...
}
```

2.3.3. W wypadku, gdy **klasa pochodna dziedziczy po klasie bazowej trybem publicznym**, do zmiennej reprezentującej obiekt (referencję do obiektu, wskaźnik do obiektu) klasy bazowej możemy przypisać obiekt (referencję do obiektu, wskaźnik do obiektu) klasy pochodnej. To zastąpienie dokonuje się automatycznie, przy użyciu tzw. konwersji standardowej.

2.3.4. Ze względu na publiczny tryb dziedziczenia klasy `Samochod_osob` po klasie `Samochod1`, zostały spełnione warunki pomyślnej realizacji następujących instrukcji:

```
Samochod1 sam6(900, przednie, 120);
Samochod1 sam7 = sam6;
Samochod1 &sam8 = sam7;
Samochod1 *wsk_sam9 = &sam6;
Samochod_osob sam_os10(1500, tylne, 190, 5, 8);
Samochod_osob sam_os11 = sam_os10;
Samochod_osob &sam_os12 = sam_os11;
samochod_osob *wsk_sam_os13 = &sam_os10;
sam7 = sam_os11;
sam8 = *wsk_sam_os13;
wsk_sam9 = &sam_os10;
```

2.3.5. W celu podstawienia za zmienną/wskaźnik o typie klasy pochodnej – obiektu/wskaźnika do obiektu klasy bazowej, trzeba:

– skorzystać z odpowiedniego operatora konwersji – operatora zdefiniowanego po stronie klasy bazowej przy użyciu słowa kluczowego `operator` i nazwy typu pochodnego. Przykładowo, konwersję z typu `Samochod1` na typ `Samochod_osob` można wykonać przy użyciu operatora w postaci:

```
Samochod1::operator Samochod_osob ()
{return Samochod_osob(poj_silnika,naped_kola,
                      max_predkosc,5,8);}
```

lub

– skorzystać z funkcji `reinterpret_cast`:

```
Samochod1 * wsk_sam = Samochod1(2400,prz,200);
Samochod_osob *wsk_samos =
         reinterpret_cast<Samochod_osob*> (wsk_sam);
```

2.3.6. Likwidacja obiektu klasy pochodnej następuje w momencie zastosowania operatora `delete` do wskaźnika tego obiektu (gdy obiekt był utworzony dynamicznie), lub w momencie osiągnięcia przez sterowanie końca zakresu deklaracji obiektu (gdy obiekt był utworzony statycznie lub przy uzyciu wskaźnika inteligentnego). W obu wymienionych sytuacjach następuje automatyczne wywołanie ciągu destruktorów klasowych: poczynając od ostatniego w hierarchii dziedziczenia destruktora klasy pochodnej i kończąc na pierwszym w tej hierarchii destruktorze klasy bazowej. W obu wypadkach likwidowany jest obszar pamięci zajęty przez składowe obiektu.

3. Metody wirtualne i klasy polimorficzne

3.1. Ze względu na zakres widoczności w obrębie hierarchii klas, wszystkie metody możemy podzielić na:
- metody lokalne, używane tylko w obrębie danej klasy (należy umieścić je w sekcji `private`),
- metody o widoczności pośredniej, używane w obrębie klasy podmiotowej i klas pochodnych (powinny znajdować się w sekcji `protected`),
- metody o widoczności globalnej, mające różne nagłówki i różne znaczenia w klasie podmiotowej i wszystkich klasach pochodnych; dostępne także z zewnątrz obiektów typu klasowego, w którym je zdefiniowano (powinny znajdować się w sekcji `public`),
- metody o widoczności globalnej, „spokrewnione" z innymi metodami o identycznych nagłówkach i zwykle odmiennym znaczeniu, zdefiniowanymi w klasach pochodnych; dostępne także z zewnątrz obiektu typu klasowego, w którym je zdefiniowano (umieszczane koniecznie w sekcji `public`). Metody z takiej spokrewnionej grupy nazywają się **metodami wirtualnymi**, a wybór właściwej dokonuje się w trakcie działania programu.

3.2. Metoda wirtualna z klasy bazowej wraz z **odpowiadającymi** jej metodami z klas pochodnych przynależnych do tej samej ścieżki dziedziczenia tworzą **rodzinę metod wirtualnych**. W programach języka C++, w obrębie rodziny metod wirtualnych, wszystkie metody muszą mieć jednakową nazwę, być jednakowego typu (typ funkcji) i mieć taką samą sygnaturę (liczba i typy parametrów formalnych funkcji).
Definicję (lub deklarację) metody wirtualnej z klasy bazowej należy obowiązkowo poprzedzić słowem kluczowym `virtual`. W celu wyraźnego oznaczenia pozostałych metod z rodziny, można słowem `virtual` poprzedzić także definicje (deklaracje) tych metod w klasach pochodnych.

3.3. Niech `Samochod1`, `Samochod_osob` oraz `Toyota` oznaczają klasy zdefiniowane jak uprzednio. Do każdej z nich wprowadzamy

dodatkowo metodę wirtualną `Podaj_opis`, umożliwiającą pozyskanie informacji na temat wskazanego obiektu klasy. Informacje redagowane przez poszczególne metody różnią się istotnie zakresem i, nieznacznie, formatem zapisu:

```
class Samochod1
{//definicje pol skladowych
 public:
   Samochod1(int psil, naped kola, int pred)
   {poj_silnika = psil;
    naped_kola = kola;
    max_predkosc = pred;}

   ~Samochod1()
   {delete [] numer_silnika;}

   void Zmien_rejestracje(string nowy_numer)
   {numer_rejestr = nowy_numer;}

   virtual void Podaj_opis()
   {cout<<"Pojemnosc silnika: "<<poj_silnika<<endl;
    cout<<"Maksymalna predkosc: "<<max_predkosc<<endl;}
 //...definicje pozostalych metod skladowych
}
```

```
class Samochod_osob : public Samochod1
{//definicje pol skladowych

 public:
   Samochod_osob
          (int psil, naped kola, int pred,
           int osoby, int poduszki):
           Samochod1(psil,kola,pred),
           liczba_osob(osoby),
           liczba_poduszek(poduszki) {}

   int Sprawdz_poduszki()
   {return liczba_poduszek;}

   virtual void Podaj_opis()
   {cout<<"Samochod "<<liczba_osob<<"-osobowy"<<endl;
    cout<<"Pojemnosc silnika: "<<poj_silnika<<endl;
    cout<<"Maksymalna predkosc: "<<max_predkosc<<endl;}

 //...definicje pozostalych metod skladowych
}
```

```
class Toyota : public Samochod_osob
{//definicje pol skladowych
 public:
   Toyota (int psil, naped kola, int pred,
          int osoby, int poduszki):
          Samochod_osob
              (psil,kola,pred,osoby,poduszki)
          {}
   void ustal_mod_Toyota(string model)
   {model_Toyota = model;}

   virtual void Podaj_opis()
   {cout<<"Toyota: "<<model_Toyota<<endl;
    cout<<"Samochod"<<liczba_osob<<"-osobowy"<<endl;
    cout<<"Pojemnosc silnika: "<<poj_silnika<<endl;
    cout<<"Maksymalna predkosc: "<<max_predkosc<<endl;}

 //...definicje pozostalych metod skladowych
}
```

Trzy powyższe metody wirtualne mają jednakową nazwę (`Podaj_opis`), są jednakowego typu (`void`) i mają jednakową, pustą sygnaturę. Tworzą więc rodzinę metod wirtualnych.

3.4. Klasyczna aktywacja metody wirtualnej ma charakter dynamiczny, tzn. wybór właściwej wersji metody ma miejsce na etapie wykonywania programu. Metodę wirtualną można jednak wywołać także w sposób statyczny, wymuszając uruchomienie wskazanej wersji metody. Po zdefiniowaniu następujących obiektów:

```
Samochod1 sam14(1300,przednie,180);
Samochod1 *wsk_sam15;
wsk_sam15 = &sam14;
Samochod1 &wsk_sam16 =
    Samochod_osob(2000,tylne,230,8,8);
Samochod1 *wsk_sam17 =
    new Toyota(1500,tylne,200,5,8);
Samochod_osob sam_os18(1800,przednie,200,6,5);
Samochod_osob *wsk_sam_os19;
wsk_sam_os19 = &sam_os18;
unique_ptr<Samochod_osob> wsk_sam_os20 =
    make_unique<Toyota>(1100,prz_tyl,155,5,3);
Toyota sam_toy21(1500,prz_tyl,220,5,4);
Toyota *wsk_toy22;
wsk_toy22 = &sam_toy21;
Samochod_osob* wsk_sam_os23 =
    reinterpret_cast<Samochod_osob *>(wsk_sam15);
```

możliwe są następujące aktywacje metody `Podaj_opis`:

```
sam14.Podaj_opis();            //Samochod1
wsk_sam15→Podaj_opis();        //Samochod1
wsk_sam16.Podaj_opis();        //Samochod_osob
(*wsk_sam17).Podaj_opis();     //Toyota
sam_os18.Podaj_opis();         //Samochod_osob
wsk_sam_os19→Podaj_opis();     //Samochod_osob
(*wsk_sam_os20).Podaj_opis(); //Toyota
(&sam_toy21)→Podaj_opis();     //Toyota
(*wsk_toy22).Podaj_opis();     //Toyota
wsk_sam16.Samochod1::Podaj_opis();
                               //Samochod1
(*wsk_sam17).Samochod1::Podaj_opis();
                               //Samochód1
sam_os18.Samochod1::Podaj_opis();
                               //Samochod1
(&sam_toy21)→Samochod_osob::Podaj_opis();
                               //Samochod_osob
(*wsk_sam_os23).Podaj_opis(); //Samochod1
```

Zastosowanie funkcji wirtualnych wiąże się z przeniesieniem ciężaru rozpoznania klasy obiektu (w konsekwencji – jej pól i metod składowych) z kompilatora na system wykonawczy.

3.5. Przyjmijmy, że klasy A i B pozostają ze sobą w relacji dziedziczenia. Nierzadko, metoda wirtualna zadeklarowana w klasie pochodnej B stanowi "rozszerzenie" odpowiadającej jej metody wirtualnej z klas bazowej A, tak jak to ma miejsce w przytoczonym przykładzie. W takiej sytuacji treść metody wirtualnej w klasie pochodnej można zapisać jak następuje:

```
virtual void Samochod1::Podaj_opis()
{cout<<"Pojemnosc silnika: "<<poj_silnika<<endl;
 cout<<"Maksymalna predkosc: "<<max_predkosc<<endl;}

virtual void Samochod_osob::Podaj_opis()
{cout<<"Samochod "<<liczba_osob<<"-osobowy"<<endl;
 Samochod1::Podaj_opis();}

virtual void Toyota::Podaj_opis()
{cout<<"Toyota: "<<model_Toyota<<endl;
 Samochod_osob:: Podaj_opis();}
```

Taki sposób definiowania metod w obrębie rodziny metod wirtualnych pozwala na skrócenie kodu programu.

3.6. Wirtualny charakter można nadać też destruktorowi dowolnej klasy bazowej. Wraz z destruktorami klas pochodnych przynależnych do tej samej ścieżki dziedziczenia, utworzy on szczególną rodzinę metod wirtualnych: rodzinę metod o różnych identyfikatorach.

3.7. Klasa, w której zdefiniowano choćby jedną metodę wirtualną nazywa się **klasą polimorficzną**. Mechanizm **polimorfizmu s**tanowi istotne dopełnienie mechanizmu **dziedziczenia**. Przede wszystkim, stwarza **możliwość pisania kodu o charakterze abstrakcyjnym, który w określonych, niedających się wcześniej przewidzieć warunkach uzyska konkretne znaczenie**.

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

Pytania:

```
#include "stdafx.h"
class Figura
{ // ...
  public:
    int x, y;
    Figura(int x, int y)
    {
      this->x = x;
      this->y = y;
    }
    ~Figura() {printf("Destruktor klasy Figura\n");}
    virtual void Opis()
    {
      printf("Figura o srodku w %d %d\n", x, y);
    }
};
class Okrag : public Figura
{ //...
  public:
    float r;
    Okrag(int x, int y, float r) : Figura(x,y)
    {
       this->r = r;
    }
    ~Okrag() {printf("Destruktor klasy Okrag\n");}
    void Opis()
    {
      printf("Okrag o srodku w %d %d i promieniu %f\n",
             x, y, r);
    }
};

int main()
{ Figura *f; Okrag *o;
  f = o = new Okrag(1,2,3);
  Figura fig = *o;
  o->Opis();
  f->Opis();
  f->Figura::Opis();
  fig.Opis();
  return 0;
}
```

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

Jakie komunikaty zostaną wyświetlone w efekcie działania poniższego programu?

```
Okrag o srodku w 1 2 i promieniu 3.000000
Okrag o srodku w 1 2 i promieniu 3.000000
Figura o srodku w 1 2
Figura o srodku w 1 2
Destruktor z klasy Figura
```

```
#include "stdafx.h"
class Figura
{ // ...
  public:
    int x, y;
    Figura(int x, int y)
    {
      this->x = x;
      this->y = y;}
    ~Figura() {printf("Destruktor klasy Figura\n");}
    virtual void Opis()
    {
      printf("Figura o srodku w %d %d\n", x, y);
    }
};
class Okrag : public Figura
{ //...
  public:
    float r;
    Okrag(int x, int y, float r) : Figura(x,y)
    {
       this->r = r;}
    ~Okrag() {printf("Destruktor klasy Okrag\n");}
    void Opis()
    {
      printf("Okrag o srodku w %d %d i promieniu %f\n", x, y, r);
    }
};

int main()
{ Figura *f; Okrag *o;
  f = o = new Okrag(1,2,3);
  Figura fig = *o;
  o->Opis();
  f->Opis();
  f->Figura::Opis();
  fig.Opis();
  delete f;  //przypadek1
  delete o;  //przypadek2
  return 0;
}
```

```
Okrag o srodku w 1 2 i promieniu 3.000000          (1)
Okrag o srodku w 1 2 i promieniu 3.000000
Figura o srodku w 1 2
Figura o srodku w 1 2
Destruktor z klasy Figura
Destruktor z klasy Figura
```

```
Okrag o srodku w 1 2 i promieniu 3.000000          (2)
Okrag o srodku w 1 2 i promieniu 3.000000
Figura o srodku w 1 2
Figura o srodku w 1 2
Destruktor z klasy Okrag
Destruktor z klasy Figura
Destruktor z klasy Figura
```

```
#include "stdafx.h"
class Figura
{ // ...
  public:
    int x, y;
    Figura(int x, int y)
    {
      this->x = x;
      this->y = y;}
    virtual ~Figura() {printf("Destruktor klasy Figura\n");}
    virtual void Opis()
    {
      printf("Figura o srodku w %d %d\n", x, y);
    }
};
class Okrag : public Figura
{ //...
  public:
    float r;
    Okrag(int x, int y, float r) : Figura(x,y)
    {
     this->r = r;
    }
    virtual ~Okrag() {printf("Destruktor klasy Okrag\n");}
    void Opis()
    {
      printf("Okrag o srodku w %d %d i promieniu %f\n", x, y, r);
    }
};

int main()
{ Figura *f; Okrag *o;
  f = o = new Okrag(1,2,3);
  Figura fig = *o;
  o->Opis();
  f->Opis();
  f->Figura::Opis();
  fig.Opis();
  //delete f;  przypadek1
  //delete o;  przypadek2
  return 0;
}
```

```
Okrag o srodku w 1 2 i promieniu 3.000000          (1)
Okrag o srodku w 1 2 i promieniu 3.000000
Figura o srodku w 1 2
Figura o srodku w 1 2
Destruktor z klasy Okrag
Destruktor z klasy Figura
Destruktor z klasy Figura
```

```
Okrag o srodku w 1 2 i promieniu 3.000000          (2)
Okrag o srodku w 1 2 i promieniu 3.000000
Figura o srodku w 1 2
Figura o srodku w 1 2
Destruktor z klasy Okrag
Destruktor z klasy Figura
Destruktor z klasy Figura
```

```
#include "stdafx.h"
class Figura
{ // ...
  public:
    int x, y;
    Figura(int x, int y)
    {
      this->x = x;
      this->y = y;}
    virtual ~Figura() {printf("Destruktor klasy Figura\n");}
    virtual void Opis()
    {
      printf("Figura o srodku w %d %d\n", x, y);
    }
};
class Okrag : public Figura
{ //...
  public:
    float r;
    Okrag(int x, int y, float r) : Figura(x,y)
    {
       this->r = r;
    }
    virtual ~Okrag() {printf("Destruktor klasy Okrag\n");}
    void Opis()
    {
      printf("Okrag o srodku w %d %d i promieniu %f\n", x, y, r);
    }
};

int main()
{ Figura *f;
  f = new Okrag(1,2,3);
  Figura fig1 = *f;
  Okrag fig2 = *f;
  fig1.Opis();
  fig2.Opis();   //
  f->Figura::Opis();
  delete f;
  return 0;}
```

```
Cannot convert from ’Figura *’ to ’Okrag *’
```

```
Figura o srodku w 1 2
Figura o srodku w 1 2
Destruktor z klasy Okrag
Destruktor z klasy Figura
Destruktor z klasy Figura
```

```
#include "stdafx.h"
class Figura
{ // ...
  public:
    int x, y;
    Figura(int x, int y)
    {
      this->x = x;
      this->y = y;}
    virtual ~Figura() {printf("Destruktor klasy Figura\n");}
    virtual void Opis()
    {
      printf("Figura o srodku w %d %d\n", x, y);
    }
};
class Okrag : public Figura
{ //...
  public:
    float r;
    Okrag(int x, int y, float r) : Figura(x,y)
    {
     this->r = r;
    }
    virtual ~Okrag() {printf("Destruktor klasy Okrag\n");}
    void Opis()
    {
      printf("Okrag o srodku w %d %d i promieniu %f\n", x, y, r);
    }
};

int main()
{ Figura *f1;
  f1 = new Okrag(1,2,3);
  Figura &f2 = *f1;
  f1->Opis();
  f2.Opis();
  delete f1;
  return 0;}
```

```
Okrag o srodku w 1 2 i promieniu 3.000000
Okrag o srodku w 1 2 i promieniu 3.000000
Destruktor z klasy Okrag
Destruktor z klasy Figura
```

```
#include "stdafx.h"
class Okrag;
class Figura
{ // ...
  public:
    int x, y;
    Figura(int x, int y)
    {
      this->x = x;
     this->y = y;
    }
    virtual ~Figura() {printf("Destruktor klasy Figura\n");}
    virtual void Opis()
    {
      printf("Figura o srodku w %d %d\n", x, y);
    }
    operator Okrag();
};
class Okrag : public Figura
{ //...
  public:
    float r;
    Okrag(int x, int y, float r) : Figura(x,y)
    {
       this->r = r;
    }
    virtual ~Okrag() {printf("Destruktor klasy Okrag\n");}
    void Opis()
    {
      printf("Okrag o srodku w %d %d i promieniu %f\n", x, y, r);
    }
};
Figura::operator Okrag()
    {
      return Okrag(x, y, 5);
    }

int main()
{ Figura *f;
  f = new Figura(1,2);
  Figura fig1 = *f;
  Okrag fig2 = *f;  //
  fig1.Opis();
  fig2.Opis();
  f->Figura::Opis();
  delete f;
 return 0; }
```

```
Destruktor z klasy Okrag
Destruktor z klasy Figura
Figura o srodku w 1 2
Okrag o srodku w 1 2 i promieniu 5.000000
Figura o srodku w 1 2
Destruktor z klasy Okrag
Destruktor z klasy Figura
Destruktor z klasy Okrag
Destruktor z klasy Figura
Destruktor z klasy Figura
```

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

Przykład systematyki jabłoni domowej

Domena	Eucaryota
Królestwo	Rośliny
Podkrólestwo	Naczyniowe
Nadgromada	Nasienne
Gromada	Okrytonasienne
Klasa	Dwuliścienne
Rząd	Różowce
Rodzina	Różowate
Rodzaj	Jabłoń
Gatunek	Jabłoń domowa

Przykład systematyki konia domowego

Domena	Eucaryota
Królestwo	Zwierzęta
Podkrólestwo	Ssaki
Nadgromada	Ssaki żyworodne
Gromada	Łożyskowce
Klasa	Nieparzystokopytne
Rząd	Konioksztaltne
Rodzina	Koniowate
Rodzaj	Koń
Gatunek	Koń domowy