Podstawy programowania obiektowego w języku C++ (2)

Opracowanie: Andrzej Bożek, Sławomir Samolej,
Politechnika Rzeszowska,
Katedra Informatyki i Automatyki,
Rzeszów, 2009.

III. Szczegółowe aspekty definiowania klas i używania obiektów. Dziedziczenie.

1. Konstruktor

Konstruktor, jako specjalna funkcja składowa klasy, został wstępnie przedstawiony w poprzednim ćwiczeniu. Obecnie będzie on omówiony dokładniej. Przy definiowaniu konstruktorów należy pamiętać o następujących założeniach:

a) nazwa konstruktora jest identyczna z nazwą klasy,
b) konstruktor nie może posiadać żadnej deklaracji typu zwracanego, nawet *void*,
c) postać listy parametrów konstruktora nie jest w żaden sposób ograniczona, można stosować przeciążenia i wartości domniemane – w praktyce, w przypadku konstruktorów stosowane są one najczęściej,
d) konstruktory mogą wywoływać inne metody klasy w celu wykonania czynności inicjalizacyjnych.

Istotne jest poznanie możliwych do uzyskania przypadków tworzenia obiektów, zależnie od rodzajów zdefiniowanych dla klasy konstruktorów. W rozgraniczeniu tych przypadków ważną rolę odgrywa pojęcie konstruktora domyślnego (domniemanego), tj. takiego, który może zostać wywołany z pustą listą argumentów. Wyróżnia się trzy sytuacje:

a) programista nie definiuje żadnego konstruktora – wtedy (i tylko wtedy) kompilator automatycznie i niejawnie generuje konstruktor domniemany, który posiada pustą implementację – nie wykonuje żadnych czynności,
b) programista definiuje przynajmniej jeden konstruktor (lub zbiór konstruktorów przeciążonych), jednak brak jest konstruktora domniemanego,
c) programista definiuje przynajmniej jeden konstruktor (lub zbiór konstruktorów przeciążonych), z których jeden może pełnić funkcję konstruktora domniemanego.

Rozwiązanie a) jest mało praktyczne i rzadko stosowane, gdyż stan początkowy obiektu nie jest inicjalizowany. Przy braku konstruktora domniemanego (przypadek b)) kłopotliwe stają się niektóre warianty użycia obiektu, związane np. z tworzeniem tablic obiektów oraz wykorzystaniem obiektów w hierarchii dziedziczenia. Wariant c) jest rozwiązaniem najpełniejszym i nie przejawia ograniczeń rozwiązań wcześniej podanych. **W przypadku klas opracowywanych w ramach ćwiczeń, zawsze definiować będziemy przynajmniej jeden jawny konstruktor**, niekoniecznie mogący funkcjonować jako domniemany.

Przykład 1.

Poniżej przedstawiono definicje trzech klas, obrazujące omówione wcześniej podstawowe możliwości deklarowania konstruktorów.

```
/*** a) brak jawnej definicji konstruktora ***/
class KlasaA
{
      // pola prywatne ...
public:
      // BRAK DEKLARACJI KONSTRUKTORA
```

```
        // metody klasy ...
};

// można tworzyć obiety tylko bez parametrów inicjalizacyjnych,
// wtedy działa konstruktor domniemany,
// automatycznie dodany przez kompilator:
KlasaA obiekt1;


/*** b) brak konstruktora domniemanego ***/
class KlasaB
{
        // pola prywatne ...
public:
        KlasaB(int a);
        KlasaB(float g, float s, char z);
        // metody klasy ...
};

// można tworzyć obiekty wykorzystujące pierwszy:
KlasaB obiekt1(4);
// lub drugi konstruktor przeciążony:
KlasaB obiekt2(3.2, 1.5, 'x');
// ale brak konstruktora domniemanego:
KlasaB obiekt3;   // BŁĄD!


/*** c) zadeklarowany konstruktor domniemany ***/
class KlasaC
{
        // pola prywatne ...
public:
        KlasaC(int a = 15);
        KlasaC(float g, float s, char z);
        // metody klasy ...
};

// znów możemy tworzyć obiekty jak poprzednio:
KlasaC obiekt1(2);
KlasaC obiekt2(6.7, 3.5, 'r');
// ale konstruktor domniemany istnieje:
KlasaC obiekt3;   // OK!
```

Podsumowując: zawsze, gdy tworzony jest obiekt danej klasy, wywoływana jest jedna implementacja konstruktora, którą kompilator może dopasować do listy parametrów inicjalizacyjnych obiektu, zgodnie z regułami przeciążenia. Jeżeli kompilator nie byłby w stanie dopasować żadnej definicji konstruktora, zgłoszony będzie błąd w czasie kompilacji.

Należy mieć świadomość, że w języku C++ istnieje wiele wariantów tworzenia obiektów, niektóre nie są nawet w sposób jawny uwidocznione w kodzie programu, jednak zawsze następuje wywołanie konstruktora. Lista najważniejszych możliwości jest następująca:

a) obiekty globalne, tj. zdefiniowane na zewnątrz klas i funkcji – tworzone są jednorazowo na początku wykonywania programu,
b) obiekty automatyczne (lokalne), tj. zdefiniowane wewnątrz funkcji i lokalnych bloków kodu – tworzone są od nowa za każdym razem, gdy przebieg programu osiąga instrukcję ich definicji,
c) obiekty statyczne – tworzone są jednorazowo w chwili napotkania ich definicji, pomimo zakresów ważności jak dla zmiennych lokalnych,

d) obiekty chwilowe, tj. będące efektem jawnego wywołania konstruktora poprzez wyrażenie postaci *<nazwaKlasy>([<parametryInicjalizacji>])* – tworzone są w momencie opracowywania instrukcji ich wywołania.

Przykład 2.

Przykład ukazuje sekwencję wywoływania konstruktorów dla prostego programu.

```
#include <iostream>
#include <string>

using namespace std;

class Klasa
{
public:
      Klasa(string s);
};

Klasa::Klasa(string s)
{
      cout << "Oto tworzony jest: " << s << "\n\n";
}

Klasa ob1("Obiekt globalny ob1");

void main()
{
      Klasa ob2("Obiekt lokalny w funkcji main ob2");
      Klasa("Obiekt chwilowy w funkcji main");

      for(int i=0; i<3; ++i)
      {
            Klasa ob3("Obiekt lokalny w petli ob3");
            static Klasa ob4("Obiekt statyczny w petli ob4");
            ob3 = Klasa("Obiekt chwilowy w petli");
      }
      cout << endl;
}
```

Klasa zdefiniowana w programie zawiera jedynie sam konstruktor, który pozwala na wypisanie tekstu podanego jako parametr inicjalizacyjny. Dzięki temu, po uruchomieniu programu, można prześledzić kolejność wykonywania konstruktorów (tworzenia obiektów).

2. Destruktor

Destruktor stanowi specjalną funkcję składową klasy, która wywoływana jest automatycznie tuż przed usunięciem obiektu z pamięci. Jego działanie jest zatem w pewnym sensie przeciwstawne do działania konstruktora, jednak zasady jego deklarowania nie są identyczne. Przy definiowaniu destruktorów należy pamiętać o następujących założeniach:

a) destruktor nie może pobierać żadnych parametrów ani nie może zwracać żadnej wartości (tak jak dla konstruktora, nie można deklarować żadnego typu zwracanego, nawet *void*),
b) w związku z założeniem powyższym, nie można dokonywać przeciążania destruktorów i dla danej klasy poprawna jest tylko jedna, ściśle określona postać jego nagłówka postaci: *~<nazwaKlasy>()*.
c) destruktory mogą wywoływać inne metody klasy i odwoływać się do pól klasy.

Ze względu na zróżnicowany czas życia obiektów, zależny od rodzaju i miejsca definicji, podobnie jak dla konstruktorów, destruktory wywoływane są w różnych sytuacjach. W odróżnieniu od tworzenia, usuwanie obiektów następuje prawie zawsze niejawnie (pomijamy działanie operatorów *new*, *delete*, których nie rozważamy w ćwiczeniu), tj. nie na podstawie instrukcji programu, ale w chwili określonej przez kompilator, wyznaczonej specyfikacją języka. Lista najważniejszych możliwości jest następująca:

a) obiekty globalne, tj. zdefiniowane na zewnątrz klas i funkcji – niszczone są jednorazowo na końcu wykonywania programu,
b) obiekty automatyczne (lokalne), tj. zdefiniowane wewnątrz funkcji i lokalnych bloków kodu – niszczone są za każdym razem, gdy przebieg programu osiąga koniec funkcji lub bloku lokalnego,
c) obiekty statyczne – niszczone są jednorazowo na końcu programu, pomimo zakresów ważności jak dla zmiennych lokalnych,
d) obiekty chwilowe, tj. będące efektem jawnego wywołania konstruktora poprzez wyrażenie postaci *<nazwaKlasy>([<parametryInicjalizacji>])* – niszczone są bezpośrednio po użyciu w wyrażeniu, w którym występują.

Przykład 3.

Przykład ukazuje sekwencję wywoływania destruktorów dla prostego programu (por. przykład 2).

```
#include <iostream>
#include <string>

using namespace std;

class Klasa
{
      string tekst;
public:
      Klasa(string s);
      ~Klasa();
};

Klasa::Klasa(string s)
{
      tekst = s;
}

Klasa::~Klasa()
{
      cout << "Oto usuwany jest: " << tekst << "\n\n";
}

Klasa ob1("Obiekt globalny ob1");

void main()
{
      Klasa ob2("Obiekt lokalny w funkcji main ob2");
      Klasa("Obiekt chwilowy w funkcji main");

      for(int i=0; i<3; ++i)
      {
            Klasa ob3("Obiekt lokalny w petli ob3");
            static Klasa ob4("Obiekt statyczny w petli ob4");
            ob3 = Klasa("Obiekt chwilowy w petli");
      }
```

```
        cout << endl;
    }
```

W powyższym programie konstruktor zapisuje tekst podany przy tworzeniu obiektu do pola prywatnego. Przed usunięciem obiektu destruktor wypisuje ten tekst, co pozwala na określenie tożsamości niszczonego obiektu.

Destruktory wykorzystywane są najczęściej do zwalniania zasobów sprzętowych, które zostały zarezerwowane w konstruktorach: dynamicznie alokowana pamięć, otwarte pliki oraz porty komunikacyjne itp. Definiowanie destruktorów nie jest obowiązkowe. W przypadku braku destruktora, żadne dodatkowe czynności przy usuwaniu obiektu nie są podejmowane. W programach tworzonych w ramach ćwiczenia zwykle nie będzie potrzeby definiowania destruktorów.

3. Metody

Metody są „standardowymi" funkcjami składowymi klasy. Nie obowiązują ich ograniczenia charakterystyczne dla konstruktorów i destruktorów. W składni oraz funkcjonalności bardzo przypominają zwykłe funkcje globalne. Ich zasadnicze cechy są następujące:

a) każda metoda musi posiadać deklarację typu zwracanego oraz listę parametrów wywołania (w szczególnym przypadku pustą),
b) jeżeli typ zwracany metody jest różny od *void*, wyrażenie wywołania metody reprezentuje sobą wartość, którą można wykorzystać, np. w celu przypisania do zmiennej, zgodnie ze składnią *<nazwaZmiennej> = <nazwaObiektu>.<nazwaMetody>(<listaParametrów>)*,
c) do metod stosują się wszystkie założenia dotyczące przeciążania i wykorzystywania wartości domyślnych parametrów,
d) metody mogą być definiowane w dowolnie wybranych sekcjach dostępu (*private*, *protected* lub *public*).

Metody używane są do modyfikowania stanu obiektu poprzez wykonywanie na nim określonych operacji oraz do odczytu i zapisu tego stanu lub jego składników poprzez kod użytkujący obiekt. Postulat hermetyzacji zakłada, że nawet bezpośredni dostęp do pola obiektu (odczyt/zapis), jeśli jest dozwolony, powinien dokonywać się poprzez metody, a nie poprzez upublicznienie pola. Dzięki temu, wszystkie operacje dostępu do obiektu będą jednolite, a nie każdą z nich można sprowadzić do upublicznienia pola (niekiedy wartość pola należy przekształcić, zweryfikować, umożliwić tylko odczyt lub tylko zapis itp.). Metody pozwalające na dostęp do pól lub innych składników stanu obiektu nazywa się zwyczajowo metodami dostępowymi i często używa się dla nich standardowego nazewnictwa: *set<NazwaPola>* dla metod zapisujących oraz *get <NazwaPola>* dla metod odczytujących. Metody publiczne wykorzystywane są do operowania na stanie obiektu poprzez kod zewnętrzny. Nierzadko definiuje się także metody prywatne, które nie są przeznaczone do bezpośredniego wywoływania z zewnątrz, ale realizują wewnętrzne operacje pomocnicze, wywoływane pośrednio poprzez metody publiczne.

Przykład 4.

Poniżej przytoczony jest kod klasy, spełniającej funkcję prostej jednostki obliczeniowej.

```
#include <iostream>
#include <string>

using namespace std;
```

```
class simpleALU
{
	char a, b, r;
	char strToInt(string s);
public:
	simpleALU();
	void setA(char A);
	void setA(string A);
	void setB(char B);
	void setB(string B);
	int getR();
	void add();
	void mul();
};

simpleALU::simpleALU()
{
	a = b = r = 0;
}

char simpleALU::strToInt(string s)
{
	int p = 128, v = 0;
	if(s.length() != 8) return 0;
	for(int i=0; i<8; i++)
	{
		if(s[i] == '1')
			v = v + p;
		else if(s[i] != '0')
			return 0;
		p = p / 2;
	}
	return v;
}

void simpleALU::setA(char A) { a = A; }

void simpleALU::setA(string A) { a = strToInt(A); }

void simpleALU::setB(char B) { b = B; }

void simpleALU::setB(string B) { b = strToInt(B); }

int simpleALU::getR() { return (int)r; }

void simpleALU::add() { r = a + b; }

void simpleALU::mul() { r = a * b; }
```

Pola klasy *a*, *b* i *r* emulują rejestry układu obliczeniowego, wykonującego obliczenia na liczbach całkowitych ośmiobitowych w kodzie U2 – dlatego nadano im typ *char*. Operandy *a* i *b* mogą być tylko zapisywane, służą do tego metody dostępowe *setA* oraz *setB*. Rejestr wyniku *r* (*result*) może być tylko odczytywany, z wykorzystaniem metody dostępowej *getR*. Metody *setA* i *setB* zostały przeciążone tak, że mogą przyjmować argumenty typu całkowitego oraz łańcuch znaków (*string*). W przypadku wartości całkowitej jest ona bezpośrednio przypisywana do pola. W przypadku łańcucha znaków, metody oczekują, że będzie on zawierał ośmioznakową sekwencję zerojedynkową, która interpretowana jest jako zapis operandu w kodzie binarnym i przetwarzana na wartość bezpośrednią za pomocą

metody prywatnej *strToInt*. Nowy wynik pojawia się w rejestrze *r* po wywołaniu jednej z metod realizujących obliczenie. W przykładowej klasie zaimplementowano dwie takie metody: *add* (dodaj) oraz *mul* (pomnóż).

4. Pola statyczne

Każdy obiekt danej klasy posiada niezależne kopie wartości pól. Dzięki temu, stany poszczególnych obiektów nie są ze sobą powiązane – obiekty mogą odzwierciedlać dane personalne różnych osób, położenia przestrzenne różnych punktów itp., zmiana wartości w jednym z nich nie wpływa na pozostałe. Niekiedy jednak zachodzi potrzeba przechowywania informacji wspólnej dla wszystkich obiektów klasy. Może to być np. liczba utworzonych obiektów tej klasy, wartość pewnego parametru statystycznego im przypisanego i inne. W takim przypadku wykorzystywane są pola tzw. **pola statyczne**. Pole statyczne jest reprezentowane przez jedną lokalizację w pamięci, wspólną dla wszystkich obiektów klasy. Każda metoda dowolnego obiektu klasy, odwołująca się do pola statycznego, realizuje dostęp do zmiennej o tej samej fizycznej lokalizacji.

Przykład 5.

Przedstawiony program demonstruje zastosowanie pól statycznych w celu przechowywania zbiorczej informacji o grupie obiektów, reprezentujących zamówienia sklepowe.

```
#include <iostream>
#include <string>

using namespace std;

class Zamowienie
{
      string nazwa;
      float cena;
      int liczbaSztuk;
      static float cenaSuma;
      static int liczbaZamowien;
public:
      Zamowienie(string nazwaZ, float cenaZ, int liczbaSztZ);
      void infoZamowienie();
      void infoZbiorcze();
};

float Zamowienie::cenaSuma;         // NIEZBĘDNA DEFINICJA !!!
int Zamowienie::liczbaZamowien;     // NIEZBĘDNA DEFINICJA !!!

Zamowienie::Zamowienie(string nazwaZ, float cenaZ, int liczbaSztZ)
{
      nazwa = nazwaZ;
      cena = cenaZ;
      liczbaSztuk = liczbaSztZ;
      liczbaZamowien++;             // aktualizacja pola statycznego
      cenaSuma = cenaSuma + cena;   // aktualizacja pola statycznego
}

void Zamowienie::infoZamowienie()
{
      cout << "Zamowienie: " << nazwa << ", cena: " << cena
            << "zl, sztuk: " << liczbaSztuk << endl;
}
```

```
void Zamowienie::infoZbiorcze()
{
      cout << "Liczba zamowien: " << liczbaZamowien
            << ", sumaryczna cena: " << cenaSuma << "zl\n";
}


void main()
{
      Zamowienie z1("Zeszyt", 6.50, 4);
      Zamowienie z2("Spinacze", 12.0, 400);
      Zamowienie z3("Klej", 2.40, 7);

      z1.infoZamowienie();
      z1.infoZbiorcze();
      cout << endl;
      z3.infoZamowienie();
      z3.infoZbiorcze();
      cout << endl;
}
```

Klasa *Zamowienie* posiada dwa pola statyczne: *float cenaSuma* oraz *int liczbaZamowien*. Pierwsze z nich przechowuje sumaryczną cenę wszystkich zamówień, a drugie ich liczbę. Informacje te są aktualizowane przy tworzeniu każdego nowego obiektu *Zamowienie*, za pomocą odpowiednich instrukcji w konstruktorze. Informacja zbiorcza, zawarta w tych polach, wypisywana jest poprzez metodę *infoZbiorcze* – informacja ta jest taka sama niezależnie od tego, na rzecz którego obiektu metoda zostanie wywołana. W aspekcie technicznym stosowania pól statycznych trzeba pamiętać, że ich deklaracje w ciele klasy nie są jednocześnie definicjami (rezerwacją pamięci), jak w przypadku zwykłych pól. Pola statyczne należy dodatkowo zdefiniować na zewnątrz klasy, korzystając ze specyfikacji zakresu (przykład 5). W takich definicjach słowo kluczowe *static* jest już opuszczane.

5. Dziedziczenie

W programowaniu obiektowym bardzo często zachodzi potrzeba dokonania uszczegółowienia definicji wcześniej utworzonej klasy. Np. na podstawie klasy *Miernik* należy zdefiniować klasę *Woltomierz*, na podstawie klasy *Komputer* klasę *Laptop*, na podstawie klasy *Mebel* klasę *Szafa* itd. Wygodnym rozwiązaniem byłoby w takim przypadku utworzenie nowej definicji poprzez uzupełnienie poprzedniej, gdyż uszczegółowienie (specjalizacja) zawsze wnosi nowe składniki i w ogólnym przypadku zachowuje wcześniejsze. Istnieje powszechny dla języków obiektowych (obecny także w C++) mechanizm, który umożliwia taką specjalizację i nazywa się **dziedziczeniem**. Dziedziczenie polega na definiowaniu nowych klas na podstawie już istniejących poprzez ich uzupełnienie o nowe składniki (zmienne i funkcje).

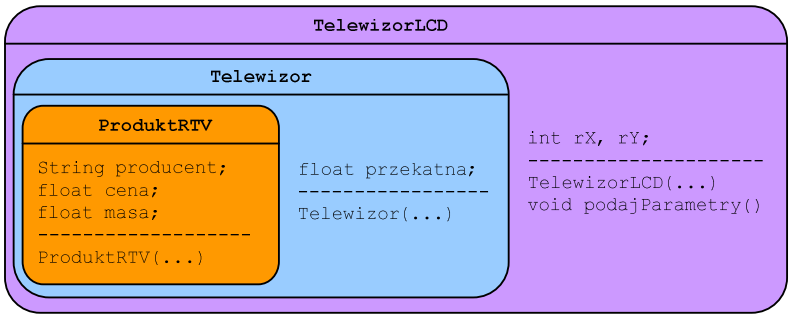
Przykład 6.

Podany program ukazuje sposób realizacji dziedziczenia w języku C++.

```
#include <iostream>
#include <string>

using namespace std;

// ---------------------------------------------
```

```
class ProduktRTV
{
protected:
      string producent;
      float cena;
      float masa;
public:
      ProduktRTV(string producentP = "Sony", float cenaP = 1000,
                  float masaP = 6);
};

ProduktRTV::ProduktRTV(string producentP, float cenaP, float masaP)
{
      producent = producentP;
      cena = cenaP;
      masa = masaP;
}

// ----------------------------------------------
class Telewizor : public ProduktRTV // lista pochodzenia
{
protected:
      float przekatna;
public:
      Telewizor(float przekatnaT = 32);
};

Telewizor::Telewizor(float przekatnaT)
{
      przekatna = przekatnaT;
}

// ----------------------------------------------
class TelewizorLCD : public Telewizor // lista pochodzenia
{
      int rX, rY;
public:
      TelewizorLCD(int rozdzielczoscX = 1440,
                        int rozdzielczoscY = 900);
      void podajParametry();
};

TelewizorLCD::TelewizorLCD(int rozdzielczoscX, int rozdzielczoscY)
{
      rX = rozdzielczoscX;
      rY = rozdzielczoscY;
}
void TelewizorLCD::podajParametry()
{
      cout << "Telewizor LCD: masa: " << masa << ", przekatna: ";
      cout << przekatna << ", rodzielczosc: " << rX << "x"
            << rY << "\n\n";
}

// ----------------------------------------------
void main()
{
      TelewizorLCD tv;
      tv.podajParametry();
}
```

Klasa *Telewizor* dziedziczy z klasy *ProduktRTV*, natomiast klasa *TelewizorLCD* z klasy *Telewizor*. Dla uproszczenia, funkcjami składowymi klas są jedynie konstruktory, za wyjątkiem metody *podajParametry*. Relacje dziedziczenia można zaprezentować graficznie jak na rysunku 1.



Rys. 1. Przykładowa hierarchia dziedziczenia.

Klasa *Telewizor* posiada składowe (pola i metody) odziedziczone z klasy *ProduktRTV* oraz dodatkowe indywidualnie zdefiniowane. Klasa *TelewizorLCD* posiada składowe odziedziczone (bezpośrednio) z klasy *Telewizor,* w tym także (pośrednio) z klasy *ProduktRTV,* oraz dodatkowe indywidualnie zdefiniowane. Klasa dziedziczona nazywana jest podstawową lub bazową, klasa dziedzicząca – pochodną.

Podstawowa charakterystyka mechanizmu dziedziczenia w języku C++, na podstawie kodu z przykładu 6, omówiona zostanie w punktach:

a) Możliwe jest dziedziczenie wielostopniowe (jak w przykładzie 6). W takim przypadku odróżnia się klasy dziedziczące bezpośrednio (*ProduktRTV-Telewizor, Telewizor-TelewizorLCD*) oraz pośrednio (*ProduktRTV- TelewizorLCD*).
b) Możliwe jest także dziedziczenie wielokrotne (wielobazowe), gdy jedna klasa dziedziczy bezpośrednio z wielu klas bazowych – nie będzie ono jednak rozważane w tym ćwiczeniu.
c) W deklaracji klasy fakt dziedziczenia określa się za pomocą tzw. **listy pochodzenia**, która występuje po nazwie klasy pochodnej i składa się ze znaku dwukropka, specyfikatora dostępu oraz nazwy klasy bazowej.
d) **Specyfikator dostępu** określa poziom dostępu do odziedziczonych składników klasy bazowej z wnętrza klasy pochodnej. Tak jak w przypadku składników klasy, może mieć on trzy postacie: *private*, *protected* lub *public*.
e) W przypadku dziedziczenia ze specyfikatorem dostępu *public*, klasa pochodna nie ma dostępu do składników prywatnych klasy bazowej, natomiast specyfikacje dostępu składników *protected* i *public* z klasy bazowej są zachowane bez zmian w klasie pochodnej. Specyfikatory dostępu dziedziczenia inne niż *public* nie będą rozważane w ćwiczeniu.
f) Dziedziczone są wszystkie składniki klasy bazowej, oprócz konstruktorów oraz destruktora (i operatora przypisania).
g) W czasie tworzenia obiektu klasy pochodnej, najpierw tworzony jest obiekt klasy bazowej (i wywoływany jej konstruktor), a następnie tworzone są dodatkowe składniki zdefiniowane w klasie pochodnej (i wywoływany jest jej konstruktor). Tylko jeśli klasa bazowa podsiada konstruktor domniemany (jak w przykładzie 6 klasy), zostanie on wywołany automatycznie, bez potrzeby dodatkowej specyfikacji w klasie pochodnej.

Jeżeli klasa bazowa nie posiada konstruktora domniemanego, klasa pochodna musi jawnie wskazać jaki konstruktor i z jakimi wartościami parametrów ma być wywołany, w

przeciwnym przypadku wystąpi błąd kompilacji. Do wskazania takiego konstruktora służy **lista inicjalizacyjna** konstruktora klasy pochodnej.

Przykład 7.

Przytoczony kod ukazuje zastosowanie listy inicjalizacyjnej do jawnego wywołania konstruktora klasy bazowej podczas inicjalizacji obiektu klasy pochodnej.

```
#include <iostream>
#include <string>
#include <cmath>

using namespace std;

// ********************************************
class SygnalElektryczny
{
protected:
      string rodzaj;
      float wartoscSkuteczna;
public:
      SygnalElektryczny(string rodzajS, float wSkutecznaS);
};

SygnalElektryczny::SygnalElektryczny(string rodzajS, float
wSkutecznaS)
{
      rodzaj = rodzajS;
      wartoscSkuteczna = wSkutecznaS;
}

// ............................................
class SygnalSinusoidalny : public SygnalElektryczny
{
      float pulsacja, faza, amplituda;
public:
      SygnalSinusoidalny(string rodzajS, float amplitudaS = 400,
            float fazaS = 0, float pulsacjaS = 314);
      float podajWartoscChwilowa(float t);
};

SygnalSinusoidalny::SygnalSinusoidalny(string rodzajS,
      float amplitudaS, float fazaS, float pulsacjaS)
      : SygnalElektryczny(rodzajS, amplitudaS / 1.42f) // lista
                                                       // inicjalizacyjna
{
      pulsacja = pulsacjaS;
      faza = fazaS;
      amplituda = amplitudaS;
}
float SygnalSinusoidalny::podajWartoscChwilowa(float t)
{
      return float (amplituda * sin(pulsacja * t + faza));
}
```

Dla uproszczenia, przykład ukazuje hierarchię dziedziczenia złożoną tylko z dwóch klas. Klasa *SygnalSinusoidalny* dziedziczy z klasy *SygnalElektryczny*. Dwuparametrowy konstruktor klasy bazowej *SygnalElektryczny* wywoływany jest z listy inicjalizacyjnej konstruktora klasy pochodnej *SygnalSinusoidalny*. Lista inicjalizacyjna towarzyszy zawsze

definicji (nie deklaracji, tj. prototypowi!) konstruktora i zaczyna się dwukropkiem następującym po jego nagłówku, a kończy przed nawiasem klamrowym otwierającym ciało konstruktora. W zakresie ważności listy inicjalizacyjnej znajdują się argumenty przekazane do konstruktora, któremu towarzyszy lista. Z tego powodu, przy wywołaniu konstruktora *SygnalElektryczny* mogły zostać wykorzystane zmienne *rodzajS* oraz *amplitudaS*, pobrane jako parametry przez konstruktor klasy pochodnej *SygnalSinusoidalny*. Usunięcie wywołania konstruktora *SygnalElektryczny* z listy inicjalizacyjnej konstruktora *SygnalSinusoidalny* spowodowałoby błąd kompilacji, gdyż klasa *SygnalElektryczny* nie posiada konstruktora domniemanego.

6. Łączenie obiektów w programie

Kompletne programy użytkowe, utworzone przy wykorzystaniu programowania obiektowego, składają się z dużej liczby obiektów. Obiekty te muszą być ze sobą odpowiednio połączone, aby realizować wymaganą funkcjonalność programu. Najczęściej w programowaniu obiektowym stosuje się dwa rodzaje powiązań, które dobrze odpowiadają relacjom występującym między obiektami w rzeczywistym świecie:

a) Specjalizacja (uszczegółowienie) – tj. tworzenie nowego obiektu na podstawie istniejącego, poprzez dodanie nowych elementów. W językach obiektowych operację tą nazywa się dziedziczeniem, dla języka C++ została ona przedstawiona w poprzednim punkcie. Określając relację między obiektami należy wybrać dziedziczenie (specjalizację) obiektu Y z obiektu X, jeśli prawdziwe jest stwierdzenie „obiekt Y jest szczególnym przypadkiem obiektu X”.
b) Agregacja (zawieranie) – tj. tworzenie nowego obiektu jako złożenia kilku obiektów istniejących. Należy wybrać agregację jako sposób utworzenia obiektu Z z obiektów X i Y, jeśli prawdziwe jest stwierdzenie „obiekt Z składa się z obiektów X i Y”.

Typowym przykładem wykorzystania wymienionych relacji pomiędzy obiektami jest struktura programu okienkowego (np. dla systemu Windows), utworzonego w technice obiektowej. Najczęściej główne okno programu odwzorowane jest poprzez główny obiekt, który zawiera w sobie inne obiekty: przyciski, etykiety, tabele, suwaki, panele, obiekty niewizualne i inne. Niektóre z nich (np. panele) mogą zawierać w sobie kolejne obiekty itd. W taki sposób agregacja odwzorowuje hierarchię i współzależności pomiędzy obiektami. Dodatkowo wykorzystywane jest dziedziczenie jako realizacja specjalizacji. Zdecydowana większość obiektów oparta jest na klasach specjalizowanych (pochodnych), np. klasa *Panel* może dziedziczyć z klasy *Kontener* (określającej zawieranie wizualne jednych elementów w innych), ta z klasy *ObiektWizualny* (określającej własności wszystkich elementów wizualnych), a ta z kolei z klasy bazowej *Komponent* (określającej znormalizowany składnik obiektowy programu).

Przykład 8.

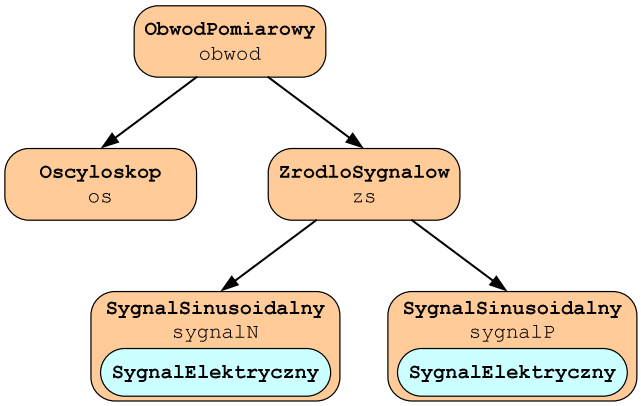
Program symuluje działanie obwodu pomiarowego z oscyloskopem dwukanałowym. Strukturalnie zbudowany jest z wielu obiektów, odwzorowujących fizyczne (oscyloskop, źródło sygnałów) i abstrakcyjne (sygnał elektryczny, sygnał sinusoidalny) składniki systemu.

```
#include <iostream>
#include <string>
#include <iomanip>
#include <cmath>

using namespace std;
```

```
// ***********************************************
// Tutaj należy dołączyć definicje dwóch klas
// SygnalElektryczny oraz SygnalSinusoidalny,
// przedstawionych w przykładzie 7.
// ***********************************************

class ZrodloSygnalow
{
      SygnalSinusoidalny sygnalN, sygnalP;      // AGREGACJA
public:
      ZrodloSygnalow(float aN, float fN, float aP, float fP);
      float podajNapiecie(float t);
      float podajPrad(float t);
};

ZrodloSygnalow::ZrodloSygnalow(float aN, float fN, float aP, float
fP)
      : sygnalN("napiecie",aN, fN), sygnalP("prad", aP, fP) {}

float ZrodloSygnalow::podajNapiecie(float t)
{
      return sygnalN.podajWartoscChwilowa(t);
}
float ZrodloSygnalow::podajPrad(float t)
{
      return sygnalP.podajWartoscChwilowa(t);
}

// ----------------------------------------------
class Oscyloskop
{
      float t, T, mCh1, mCh2;
      int w, k;
public:
      Oscyloskop(float skalaT, float maxCh1, float maxCh2);
      float podajCzas();
      void wyswietl(float ch1 = 0, float ch2 = 0);
      float podajMaxCh1();
      float podajMaxCh2();
      float podajSkaleT();
};

Oscyloskop::Oscyloskop(float skalaT, float maxCh1, float maxCh2)
{
      t = 0;
      w = k = 1;
      T = skalaT;
      mCh1 = maxCh1;
      mCh2 = maxCh2;
}
float Oscyloskop::podajCzas()
{
      return t;
}
void Oscyloskop::wyswietl(float ch1, float ch2)
{
      if(t < 0) return;

      float stepCh1 = 2 * mCh1 / 19;
      float stepCh2 = 2 * mCh2 / 19;
```

```
        if((int)((mCh1 + ch1) / stepCh1) == 19 - w) cout << "*";
        else if((int)((mCh2 + ch2) / stepCh2) == 19 - w) cout << "+";
        else if(w == 1 || w == 19) cout << ".";
        else if(w == 10) cout << "-";
        else cout << " ";
        ++k;
        if(k > 79)
        {
                cout << endl;
                k = 1;
                ++w;
        }
        t = T * (k-1) / 79;
        if(w > 19) t= -1;
}
float Oscyloskop::podajMaxCh1() { return mCh1; }
float Oscyloskop::podajMaxCh2() { return mCh2; }
float Oscyloskop::podajSkaleT() { return T; }

// ---------------------------------------------
class ObwodPomiarowy
{
        ZrodloSygnalow zs;      // AGREGACJA
        Oscyloskop os;          // AGREGACJA
public:
        ObwodPomiarowy(float skalaT, float mCh1, float mCh2);
        void dokonajPomiaru();
};

ObwodPomiarowy::ObwodPomiarowy(float skalaT, float mCh1, float mCh2)
        : zs(360, 0, 2.4f, 3.14f/2), os(skalaT, mCh1, mCh2) {}

void ObwodPomiarowy::dokonajPomiaru()
{
        float t;

        cout << "*** napiecie (zakres +/-" << os.podajMaxCh1()
                << "V)     +++ prad (zakres +/-" << os.podajMaxCh2()
                << "A)     czas " << os.podajSkaleT() << "s\n\n";
        t = os.podajCzas();
        while(t >= 0)
        {
                os.wyswietl(zs.podajNapiecie(t), zs.podajPrad(t));
                t = os.podajCzas();
        }
        cout << "\n\n";
}

// ---------------------------------------------

void main()
{
        ObwodPomiarowy obwod(0.05f, 400, 4);
        obwod.dokonajPomiaru();
}
```

Użytą w programie strukturę obiektową przedstawia w sposób graficzny rysunek 2.



Rys. 2. Struktura obiektowa programu z przykładu 8.

Bez zmian wykorzystano w programie klasy *SygnalElektryczny* oraz *SygnalSinusoidalny*, przedstawione w przykładzie 7. Zawieranie na rys. 2 zaznaczone jest za pomocą strzałek, skierowanych od obiektu agregującego do obiektów składowych. W programie tworzony jest jeden obiekt główny *obwod* klasy *ObwodPomiarowy*. Zawiera on w sobie obiekty składowe: *os* klasy *Oscyloskop* oraz *zs* klasy *ZrodloSygnalow*. Klasa *ZrodloSygnalow* agreguje w sobie dwa obiekty klasy *SygnalSinusoidalny* (która jest pochodną klasy *SygnalElektryczny*) o nazwach *sygnalN* oraz *sygnalP*.

Przekładając opis struktury obiektowej na język potoczny, można ją opisać następująco: „Obwód pomiarowy składa się z oscyloskopu połączonego ze źródłem sygnałów. Źródło sygnałów dostarcza dwóch sygnałów sinusoidalnych, będących szczególnym przypadkiem sygnałów elektrycznych. Jeden z nich jest sygnałem napięciowym, drugi prądowym”.

W języku C++ agregacja realizowana jest poprzez definiowanie obiektów składowych jako pól w obiektach agregujących. W kodzie programu z przykładu 8 odpowiednie miejsca opatrzone są komentarzem *// AGREGACJA*. Istotny jest fakt, że pola (poza statycznymi) uzyskują lokalizacje w pamięci dopiero przy tworzeniu konkretnych obiektów klasy i wtedy także są inicjalizowane. Z tego powodu w ciele klasy deklaracjom pól nie mogą towarzyszyć żadne zapisy inicjalizacyjne – ich inicjalizacja odbywa się w konstruktorze. Także jeżeli polem jest obiekt, w miejscu deklaracji podawana jest tylko nazwa, natomiast wywołanie jego konstruktora może nastąpić w ramach listy inicjalizacyjnej konstruktora klasy agregującej. Jeżeli obiekt składowy posiada konstruktor domniemany, zostanie on wywołany automatycznie przy jego tworzeniu, o ile obiekt agregujący nie wywoła jawnie innej jego implementacji z listy inicjalizacyjnej swojego konstruktora. Jeżeli natomiast obiekt składowy nie posiada konstruktora domniemanego, jawne wywołanie z listy inicjalizacyjnej jest niezbędne, w przeciwnym wypadku wystąpi błąd kompilacji. Sytuacja jest więc analogiczna jak przy dziedziczeniu. Ponieważ w programie z przykładu 8 żadna klasa nie posiada konstruktora domniemanego, wszystkie konstruktory klas agregujących zawierają w swoich listach inicjalizacyjnych wywołania konstruktorów klas składowych. Dotyczy to konstruktorów klas agregujących *ObwodPomiarowy* oraz *ZrodloSygnalow*. Należy zauważyć, że wymienione konstruktory nie wykonują żadnych innych operacji poza inicjalizacją obiektów składowych, dlatego ich ciała pozostają puste.

Właściwie zbudowany program obiektowy powinien całą swoją funkcjonalność zawierać wewnątrz obiektów, z pominięciem funkcji globalnych. Niektóre języki obiektowe (np. Java) w ogóle nie pozwalają na definiowanie funkcji na zewnątrz obiektów. W programie z przykładu 8, działanie globalnej funkcji *main* ogranicza się tylko do utworzenia obiektu głównego i wywołania jednej jego metody.

Komunikacja w strukturze obiektowej, opartej na relacjach specjalizacji i agregacji, zachodzi bezpośrednio tylko na poziomach lokalnych pomiędzy obiektami agregującymi a ich składnikami. Jest to jednokierunkowy tryb komunikacji (od obiektu agregującego do obiektu składowego), realizowany poprzez wywoływanie metod obiektów składowych. Niekiedy może być potrzebne rozszerzenie możliwości komunikacji, tak aby połączyć obiekty odległe w strukturze lub/i zapewnić komunikację dwukierunkową. W języku C++ może to zostać wykonane w prosty sposób poprzez użycie wskaźników do obiektów.

Przykład 9.

Program zawiera dwa obiekty tej samej klasy, które komunikują się wzajemnie (dwukierunkowo) za pomocą wskaźników.

```
#include <iostream>
#include <string>
#include <ctime>

using namespace std;

class Gracz
{
      Gracz *partner;
      string imie;
public:
      Gracz(string imieG = "Unknown", Gracz *partnerG = NULL);
      void ustawPartnera(Gracz *partner);
      void zagraj();
};

Gracz::Gracz(string imieG, Gracz *partnerG)
{
      imie = imieG;
      partner = partnerG;
}
void Gracz::ustawPartnera(Gracz *partnerG)
{
      partner = partnerG;
}
void Gracz::zagraj()
{
      int los;
      los = 2 + rand() % 5;
      cout << imie;
      if(rand() % 7)
      {
            for(int i=0; i<los; ++i)
                  cout << "-";
            partner->zagraj();
      }
}
// ----------------------------------------------
void main()
{
      Gracz zawodnikA("A");
      Gracz zawodnikB("B", &zawodnikA);

      zawodnikA.ustawPartnera(&zawodnikB);
```

```
        srand((unsigned)time(NULL));

        for(int i=0; i<3; ++i)
        {
              zawodnikA.zagraj();
              cout << endl;
              zawodnikB.zagraj();
              cout << endl;
        }
        cout << "\n\n";
}
```

Działanie programu symuluje grę pomiędzy dwoma zawodnikami, reprezentowanymi przez obiekty *zawodnikA* i *zawodniB* klasy *Gracz*. Podstawową funkcjonalność realizuje metoda *zagraj* klasy *Gracz*. Metoda ta wypisuje imię zawodnika, a następnie decyduje w sposób losowy czy kontynuować, czy przerwać grę. W przypadku przerwania, metoda kończy swoje wykonywanie. W przypadku kontynuacji, metoda czeka losowo wybrany czas (symbolizowany liczbą wypisanych na ekranie znaków „–”) i wywołuje metodę o takiej samej nazwie (*zagraj*) na rzecz obiektu, który został określony jako partner. Tak więc, po wywołaniu metody *zagraj* na rzecz jednego z obiektów-zawodników, następuje seria wzajemnych wywołań tej metody, aż do losowej decyzji o rezygnacji z wywołania w jednym z obiektów. Inicjujące wywołania metody *zagraj* wykonywane są w funkcji *main*, trzykrotnie, naprzemiennie dla każdego z obiektów-zawodników.

Dwa współpracujące obiekty istnieją na poziomie równorzędnym, zatem nie występuje pomiędzy nimi relacja specjalizacji ani agregacji. Komunikacja odbywa się poprzez wykorzystanie wskaźników do obiektów. Klasa *Gracz* zawiera pole *partner*, będące wskaźnikiem do obiektu własnego typu. Za pomocą tego pola dwa utworzone obiekty wskazują na siebie nawzajem. Dla przykładu, wprowadzono dwa sposoby ustawienia partnera: wskazanie na niego może zostać przekazane jako drugi parametr wywołania konstruktora lub jako parametr wywołania metody *ustawPartnera*. Oczywiście pierwszy ze sposobów nie może być zastosowany dla obiektu tworzonego w pierwszej kolejności. Postać instrukcji wywołania metody *partner->zagraj()* pokazuje, że odwołanie do składowych obiektu poprzez wskazanie odbywa się analogicznie jak w przypadku struktur w języku C.

Zadania do samodzielnej realizacji

1. Uruchomić podane w treści instrukcji programy z przykładów nr: 2, 3, 5, 6, 8, 9 i przeanalizować ich działanie.
2. Określić, czy kolejność tworzenia obiektów i wywoływania konstruktorów dla programu z przykładu 3 zgadza się z oczekiwaniami.
3. Określić, czy kolejność niszczenia obiektów i wywoływania destruktorów dla programu z przykładu 4 zgadza się z oczekiwaniami.
4. Utworzyć program, w oparciu o przykłady 2 i 3, w którym obiekt *Klasa* posiadał będzie zarówno konstruktor jak i destruktor wypisujące nazwę obiektu. Przeanalizować relacje pomiędzy czasami życia (przedziałami czasu od utworzenia do destrukcji) poszczególnych obiektów.
5. Do kodu zamieszczonego w przykładzie 4 dołączyć funkcję *main*, która pozwoli na przetestowanie działania klasy *simpleALU*. Powinno zostać sprawdzone działanie metod

dostępowych *setA*, *setB*, *getR* z uwzględnieniem przeciążeń oraz metod operacyjnych *add* i *mul*.

6. Klasę *simpleALU* z przykładu 4 uzupełnić o metody *sub* i *div*, realizujące odpowiednio odejmowanie i dzielenie. Próba dzielenia przez zero powinna być wykrywana i skutkować wynikiem 0 w rejestrze *r*. Utworzyć funkcję *main* (lub uzupełnić funkcję utworzoną w poprzednim zadaniu), która pozwoli przetestować dodane metody.
7. W klasie *Zamowienie* z przykładu 5 dodać dwa pola statyczne typu *string* i tak uzupełnić implementację konstruktora, aby jedno z nich przechowywało zawsze nazwę ostatnio utworzonego zamówienia, a drugie przedostatniego. Dodać metodę *infoNajnowsze*, która wypisywać będzie nazwy tych zamówień. Przetestować działanie klasy, tworząc kolejno co najmniej pięć obiektów klasy *Zamowienie* i po każdym utworzeniu sprawdzać wynik wywołania metody *infoNajnowsze*.
8. W programie z przykładu 6 usunąć wszystkie wartości domyślne parametrów konstruktorów we wszystkich klasach dziedziczonych i w zamian wprowadzić jawne wywołania konstruktorów klas bazowych w listach inicjalizacyjnych konstruktorów klas pochodnych, tak aby działanie programu nie uległo zmianie.
9. Uzupełnić kod podany w przykładzie 7, tworząc funkcję *main*, tak aby powstał program będący kalkulatorem wartości chwilowej przebiegu sinusoidalnego. Funkcja *main* powinna pobierać od użytkownika wartość pulsacji, amplitudy i fazy początkowej przebiegu oraz czas, dla którego ma zostać wyliczona wartość chwilowa. Następnie powinna utworzyć obiekt klasy *SygnalSinusoidalny* (zainicjalizowany odpowiednimi wartościami parametrów) i wypisać wynik wywołanej na jego rzecz metody *podajWartoscChwilowa*. UWAGA: W języku C++, wewnątrz funkcji, deklaracje zmiennych mogą przeplatać się z innymi instrukcjami w dowolnej kolejności.
10. Zrealizować zadanie nr 6 bez modyfikowania klasy *simpleALU* (oprócz zmiany specyfikacji dostępu pól z *private* na *protected*), wykorzystując dziedziczenie do utworzenia klasy pochodnej *simpleALU_2*, posiadającej wymagane metody *sub* i *div*.
11. Zrealizować zadanie nr 7 bez modyfikowania klasy *Zamowienie*, wykorzystując dziedziczenie do utworzenia klasy pochodnej *Zamowienie_2*, posiadającej opisaną w zadaniu 7 rozszerzoną funkcjonalność.
12. Przeanalizować strukturę programu z przykładu 8. Odnaleźć odpowiednie miejsca w kodzie i zmodyfikować ja tak, aby:
    a) amplituda pierwszego sygnału (napięcia) wyniosła 280;
    b) zakres czasu obserwacji wyniósł 0.03s;
    c) faza początkowa drugiego sygnału (prądu) wyniosła $3\pi/4$;
    d) zakres pomiarowy oscyloskopu dla drugiego kanału wyniósł 3.

    Każdorazowo uruchomić program i sprawdzić poprawność wprowadzonych modyfikacji.
13. Zmodyfikować program z przykładu 9, dodając jeszcze jeden obiekt klasy *Gracz* (*zawodnikC*) tak, aby współpraca między trzema obiektami odbywała się według następującej reguły:
    a) każdy z obiektów-zawodników wywołuje metodę *zagraj* jednego, trwale ustalonego współzawodnika tak, aby uzyskać stały schemat akcji kolejnych obiektów (np. A, B, C, A, B, C, …);
    b) każdy z obiektów-zawodników wywołuje metodę *zagraj* jednego, losowo wybranego współzawodnika.

Każdorazowo uruchomić program i sprawdzić poprawność wprowadzonych modyfikacji.

WSKAZÓWKA: Wariant a) wymaga jedynie modyfikacji funkcji *main*, wariant b) wymaga także modyfikacji klasy *Gracz*, wraz z metodą *zagraj*.

14. Wypełnić kod metod oznaczonych ramkami tak, aby ich działanie było zgodne z przytoczonymi obok ramek wymogami.

```
#include <iostream>
#include <string>

using namespace std;

class Silnik
{
      char symbol[20];
      float pradN;
      float mocN;
public:
      Silnik(char s[], float p, float m);
      void wypiszInfo();
      void zmienParam(float p, float m);
};

Silnik::Silnik(char s[], float p, float m)
{
  ┌──────────────────────────────────────────┐
  │                                          │
  │                                          │
  │                                          │
  │                                          │
  │                                          │
  └──────────────────────────────────────────┘
}
void Silnik::wypiszInfo()
{
  ┌──────────────────────────────────────────┐
  │                                          │
  │                                          │
  │                                          │
  │                                          │
  │                                          │
  └──────────────────────────────────────────┘
}
void Silnik::zmienParam(float p, float m)
{
  ┌──────────────────────────────────────────┐
  │                                          │
  │                                          │
  │                                          │
}└──────────────────────────────────────────┘

class SilnikIndukcyjny : public Silnik
{
      char polaczenie;
      int liczbaPB;
public:
      SilnikIndukcyjny(char s[], float p, float m);
      void wypisz();
      void zmienPol(char p);
};

SilnikIndukcyjny::SilnikIndukcyjny(char s[], float p, float m)
      : Silnik(s, p, m)
{
      polaczenie = 'Y'; liczbaPB = 2;
}
```

Konstruktor wypełnia kolejne pola prywatne wartościami kolejnych parametrów swojego wywołania.

*Metoda wypisuje na ekranie wartości pól obiektu (*symbol, pradN, mocN*) w czytelnej postaci.*

Metoda przypisuje polom pradN *i* mocN *nowe wartości, będące parametrami jej wywołania.*

```
void SilnikIndukcyjny::wypisz()
{
}
```

*Metoda wypisuje na ekranie wartości wszystkich pól obiektu (*symbol, pradN, mocN, polaczenie, liczbaPB*) w czytelnej postaci.*

```
void SilnikIndukcyjny::zmienPol(char p)
{
}
```

Metoda przypisuje do pola polaczenie *znak 'Y', jeżeli taki znak został podany jako argument jej wywołania, w przeciwnym wypadku przypisuje znak 'D'.*

```
void main()
{
      Silnik A("DC12M", 6.7f, 200);
      SilnikIndukcyjny B("SA222K", 14, 2200);

      A.wypiszInfo();
      cout << endl;
      B.wypiszInfo();
      cout << endl;

      A.zmienParam(7.4f, 240);
      B.zmienPol('D');

      A.wypiszInfo();
      cout << endl;
      B.wypisz();
      cout << endl;
}
```

15. Określić co wypisze program, uzyskany w zadaniu poprzednim po wypełnieniu metod. Sprawdzić przewidywania, uruchomiając program.

16. Utworzyć nową wersję metody *main* dla programu z zadania 14, która wykonywała będzie następujące operacje na obiektach:
    - utworzenie dwóch obiektów klasy *SilnikIndukcyjny* o nazwach *sind1* i *sind2*, pierwszy powinien posiadać symbol „SIA32Y", wartość pola *pradN* równą 12.4f oraz wartość pola *mocN* równą 3000.f, dla drugiego odpowiednie wartości to: „SIA56S", 10.f, 2400.f,
    - wypisanie pełnych informacji o obiektach (wybierając odpowiednie metody), najpierw dla obiektu *sind2*, następnie dla *sind1*,
    - zmiana wartości pól *pradN* i *mocN* obiektu *sind2* na takie, jaki zostały początkowo przypisane obiektowi sind1 (przy jego tworzeniu),
    - wywołanie metody *zmienPol* na rzecz obiektu sind2 z parametrem 'D',
    - ponowne wypisanie pełnych informacji o obiektach, najpierw dla obiektu *sind1*, następnie dla *sind2*.

LITERATURA

1. C. Delannoy, *Ćwiczenia z języka C++*, WNT, Warszawa 1993
2. J. Grębosz, *Symfonia C++ Standard*, Editions 2000, Kraków 2008
3. *C++ Wikibooks*, http://pl.wikibooks.org/wiki/C%2B%2B
4. K. Kuczmarski, *Od zera do gier kodera*, http://www.gamedev.pl/tutorials.php?x=category&id=18