Podstawy programowania obiektowego w języku C++ (1)

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

I. Wprowadzenie – podstawowe rozszerzenia języka C++ względem języka C

Język C++ powstał poprzez rozszerzenie języka C, które umożliwiło programowanie obiektowe. Rozszerzenie to pozwala zasadniczo na definiowanie klas i obiektów, stosowanie dziedziczenia oraz używanie funkcji polimorficznych. Poza elementami związanymi bezpośrednio z techniką obiektową, w języku C++ dodano kilka innych rozszerzeń, które pośrednio wspierają programowanie obiektowe lub korzystają z jego możliwości. Niektóre z nich zostaną rozważone w tej części ćwiczenia.

Punkt 1 prezentuje typową dla języka C++ obsługę wejścia/wyjścia opartą na klasie strumieniowej *iostream*, punkt 2 przedstawia mechanizm przeciążenia funkcji i użycie parametrów domyślnych, punkt 3 prezentuje zmienne typu referencyjnego.

1. Strumieniowa klasa wejścia/wyjścia *iostream*

W języku C++ można nadal używać funkcji zadeklarowanych w pliku nagłówkowym *stdio.h* w celu obsługi konsolowego wejścia/wyjścia, identycznie jak miało to miejsce w języku C. Jednakże dla języka C++ przygotowana została nowa, dedykowana biblioteka przeznaczona do tego celu, dołączana do programu poprzez plik nagłówkowy *iostream* i ona powinna być stosowana dla programów tworzonych w tym języku, gdyż jest rozwiązaniem ulepszonym w stosunku do poprzedniego:

- Wykorzystuje technikę obiektową (*iostream* jest **klasą**), niedostępną w języku C. Dzięki tej technice funkcjonalność biblioteki może być łatwo rozszerzana przez programistę z wykorzystaniem mechanizmu dziedziczenia.
- Nie powtarza błędów i niedociągnięć, które dostrzeżono w zbiorze funkcji zadeklarowanych w *stdio.h*.
- Jest łatwiejsza do zastosowania względem rozwiązania poprzedzającego.

Po dołączeniu pliku nagłówkowego *iostream*, w kodzie programu stają się dostępne klasa i obiekty związane z obsługą wejścia/wyjścia. Wśród nich dostępne są dwa predefiniowane obiekty:

cin – obiekt standardowego strumienia wejściowego,
cout – obiekt standardowego strumienia wyjściowego.

Wykorzystanie wymienionej wyżej pary obiektów wystarcza do zrealizowania podstawowej obsługi konsolowego wejścia/wyjścia, podobnej do tej, jaką umożliwiały funkcje zadeklarowane w *stdio.h*.

Przykład 1.

Program o podanym poniżej kodzie, używa obiektów strumieniowych *cin* oraz *cout* w celu realizacji dialogu z użytkownikiem.

```
#include <iostream>

using namespace std;
```

```
void main()
{
	float a, b;

	cout << "Podaj dwie liczby: a i b, oblicze a / b"
		<< "\n\na = ";
	cin >> a;
	cout << "b = ";
	cin >> b;
	if(b != 0)
		cout << endl << a << " / " << b << " = "
			<< (a/b) << "\n\n";
	else
		cout << endl << a << " / " << b
			<< " - NIEWYKONALNE, nie dzielimy przez 0 !"
			<< "\n\n";
}
```

Strumień jest abstrakcyjnym określeniem formy przekazywania informacji, w którym są one transmitowane kolejno, jedna po drugiej. Wstawienie elementu do strumienia w określonym miejscu powoduje, że musi on być odebrany z tego strumienia w takim samym umiejscowieniu. Strumień wyjściowy, realizowany przez obiekt *cout*, przekazuje wprowadzone do niego informacje do standardowego wyjścia. Będą one wyświetlone w takiej kolejności, w jakiej zostały umieszczone pomiędzy operatorami „<<”. Analogicznie dzieje się dla obiektu *cin*, przy czym w jego przypadku dane wprowadzane z klawiatury przypisywane są do kolejnych zmiennych, rozdzielonych operatorami „>>”. Przy wykorzystaniu obiektów strumieni *cout* oraz *cin* charakterystyczne jest, że:

- Operatory „<<” i „>>”, łączące zmienne i wyrażenia, stwarzają wizualne odwzorowanie strumienia oraz kierunku jego przepływu.
- Typy zmiennych i wyrażeń, umieszczonych pomiędzy operatorami „<<” i „>>”, są samoczynnie odpowiednio interpretowane przez obiekty *cout*, *cin*, bez potrzeby jawnego formatowania, jak w przypadku funkcji *printf* i *scanf* biblioteki *stdio*.

Wymienione własności obiektów strumieniowych wynikają z mechanizmów obiektowych języka C++ (przeciążenie operatorów i typy referencyjne) i nie mogłyby być uzyskane w języku C.

Brak potrzeby określania formatu, gdy wartości zmiennych powinny być formatowane w sposób standardowy dla danego typu danych, upraszcza użycie obiektów *cout* i *cin*. Jeżeli jednak występuje potrzeba modyfikacji formatowania domyślnego, klasa *iostream* daje taką możliwość, udostępniając dwa mechanizmy formatowania. Pierwszym z nich są funkcje formatujące, wywoływane na rzecz obiektów strumieniowych, a drugim manipulatory strumieniowe. W ramach ćwiczenia omówiony zostanie jedynie drugi z mechanizmów.

Manipulatory strumieniowe są specjalnymi obiektami, które wstawione do strumienia realizują określone działanie. Najczęściej zadaniem tym jest przełączenie stanu formatowania, które obowiązuje do końca instrukcji z obiektem *cout* lub *cin*. Nie musi być tak zawsze, np. w przykładzie 1 użyto manipulatora *endl*, który nie zmienia stanu formatowania, ale powoduje przejście do nowej linii w wypisywanym tekście, tj. działa identycznie jak sekwencja ucieczki „\n”.

Przykład 2.

Program poniższy wykorzystuje manipulatory strumieniowe do modyfikacji ustawień wypisywania danych.

```
#include <iostream>
#include <iomanip>
```

```
using namespace std;

void main()
{
	int a, b, c;

	cout << "Podaj trzy liczby calkowite: a, b, c:" << "\n\na = ";
	cin >> a;
	cout << "b = ";
	cin >> b;
	cout << "c = ";
	cin >> c;
	cout << "\na = " << a << "(10) = " << oct << a
		<< "(8) = " << hex << a << "(16)\n";
	cout << "b = " << b << "(10) = " << setbase(8) << b
		<< "(8) = " << setbase(16) << b << "(16)\n";
	if(b != 0)
		cout << a << " / " << b << " = " << setprecision(c)
			<< (a*1./b) << " (precyzja: " << c << ")\n\n";
	else
		cout << a << " / " << b << " - NIEWYKONALNE !\n\n";
}
```

Wyróżnia się dwa rodzaje manipulatorów strumieniowych:

a) manipulatory bezparametrowe – wywoływane poprzez samą tylko nazwę, która jednoznacznie precyzuje ich funkcję,
b) manipulatory parametryzowane – wywoływane wraz z parametrami, specyfikującymi właściwości formatu; **zastosowanie manipulatorów parametryzowanych wymaga dołączenia pliku nagłówkowego *iomanip***.

Użycie manipulatorów demonstruje przykład 2. Manipulatory bezparametrowe *oct*, *hex* ustawiają formatowanie liczb całkowitych poprzez wybór reprezentacji, odpowiednio ósemkowej i szesnastkowej. Parametryzowany manipulator *setbase()* jest funkcjonalnym odpowiednikiem zbioru manipulatorów *dec*, *oct*, *hex*, jednakże nowa baza reprezentacji podawana jest w postaci parametru. Pomimo zastosowania do ostatniego wypisania liczby *a* notacji szesnastkowej, wypisywanie liczby *b* zaczyna się od domyślnej notacji dziesiętnej. Jest to potwierdzenie, że stan zmodyfikowany manipulatorami powraca do postaci domyślnej w każdym nowym użyciu obiektów *cout*, *cin*. Manipulator parametryzowany *setprecision()* określa maksymalną liczbę cyfr po przecinku dla formatów zmiennoprzecinkowych.

2. Przeciążenie funkcji i parametry domyślne

W języku C++ możliwe jest zdefiniowanie w jednym zakresie widoczności kilku funkcji o takiej samej nazwie, różniących się liczbą lub/i typem argumentów. Kompilator automatycznie przyporządkuje odpowiednią funkcję do instrukcji jej wywołania na podstawie dopasowania argumentów. Mechanizm ten nazywa się przeciążeniem (przeładowaniem) funkcji.

Przykład 3.

Program zawiera zbiór funkcji przeciążonych, pozwalający na wypisywanie godziny w oparciu o różne zestawy parametrów.

```
#include <iostream>
#include <iomanip>
```

```
#include <string>

using namespace std;

void printTime(string t);
void printTime(int h, int m, int s);
void printTime(int h, int m);
void printTime(int h);

void main()
{
	printTime("13:13.13");
	printTime(2);
	printTime(5, 35);
	printTime(1, 20, 45);
}

void printTime(string t)
{
	cout << "Jest godzina: " << t << endl;
}

void printTime(int h, int m, int s)
{
	cout << "Jest godzina: " << h << ":" << setfill('0')
		<< setw(2) << m << "." << setw(2) << s << endl;
}

void printTime(int h, int m)
{
	printTime(h, m, 0);
}

void printTime(int h)
{
	printTime(h, 0, 0);
}
```

Przykładowy program zawiera cztery przeciążone definicje funkcji *printTime*. Funkcje o prototypach *void printTime(string t)* oraz *void printTime(int h)* są przeciążone ze względu na typ parametru, natomiast pomiędzy definicjami przyjmującymi parametry typu *int* występuje przeciążenie wynikające z różnej liczby parametrów. Dzięki różnicom w liście parametrów, wszystkie wywołania funkcji *printTime* obecne w funkcji *main* mogą zostać jednoznacznie dopasowane do odpowiedniej definicji. Kolejne wywołania zostaną automatycznie przyporządkowane do definicji odpowiednio: pierwszej, czwartej, trzeciej i drugiej. Warto zauważyć, że możliwe jest uproszczenie kodu funkcji przeciążonej, poprzez odwołanie do definicji innej funkcji przeciążonej, jak dokonano tego w przypadku trzeciej i czwartej definicji w rozważanym przykładzie.

Mechanizm przeciążenia jest istotny dla języka C++, zwłaszcza w odniesieniu do funkcji składowych klas, zwanych metodami. Metody odzwierciedlają działania, które można wykonać na obiekcie. Dzięki przeciążeniu, te same akcje wywoływać można z różnymi alternatywnymi zbiorami parametrów wejściowych. W praktyce przeciążanie najczęściej stosuje się odniesieniu do konstruktorów.

W przykładzie 3 wykorzystano dwa manipulatory parametryzowane: *setw()* – ustawia minimalną szerokość pola dla wypisywanego tekstu, *setfill()* – ustawia znak, który będzie wstawiany w wolnych miejscach pola wyjściowego.

Język C++ pozwala na użycie parametrów domyślnych (domniemanych) wywołania funkcji. W tym celu, dla wybranych parametrów wywołania, w deklaracji funkcji po znaku równości dopisuje się wartość domyślną, np. zapis:

float aproksymacja(float x, int k, char p = 'v', int w = 5);

deklaruje funkcję z dwoma parametrami domniemanymi *char p* i *int w*, o wartościach domyślnych odpowiednio 'v' i 5. Tak zadeklarowana funkcja może zostać wywołana z różną długością listy parametrów aktualnych. Parametry posiadające wartości domyślne zostaną nimi zainicjalizowane, w przypadku opuszczenia wartości im odpowiadającej na liście wywołania, np.:

float aproksymacja(4.3, 5); // p = 'v', w = 5
float aproksymacja(8., 3, 'a'); // w = 5
float aproksymacja(2.2, 5, 'g', 7);

Aby ustrzec się błędów przy tworzeniu funkcji z parametrami domyślnymi, należy mieć na względzie podstawowe zasady składniowe ich definiowania:

a) Jest wymagane, aby wszystkie parametry domniemane umieszczone były na końcu listy, tzn., jeżeli któryś z parametrów posiada wartość domyślną, wszystkie dalsze („na prawo") parametry na liście także muszą mieć przypisaną wartość domyślną. Ten wymóg składniowy wynika ze spostrzeżenia, że przy braku takiego ograniczenia nie byłoby możliwe jednoznaczne dopasowanie parametrów opuszczanych przy wywołaniu funkcji z ich wartościami domyślnymi.
b) Przypisanie wartości domyślnych następuje tylko w ramach deklaracji funkcji, tj. przy pierwszym wystąpieniu jej nagłówka w kodzie programu – jeżeli funkcja posiada prototyp, tylko w prototypie.

Mechanizm parametrów domyślnych wywołania funkcji jest blisko spokrewniony z mechanizmem przeciążenia. Zamiast stosować parametry domyślne rozważonej funkcji *aproksymacja*, można byłoby utworzyć trzy jej wersje przeładowane, przyjmujące odpowiednio dwa, trzy i cztery parametry. Z punktu widzenia instrukcji wykorzystujących tą funkcję, podane alternatywne rozwiązania będą nierozróżnialne. W istocie mechanizm parametrów domyślnych jest funkcjonalnie równoważny mechanizmowi przeciążenia ze względu na liczbę parametrów (nie ze względu na typ) i można je stosować zamiennie, przy czym pierwszy wariant daje zwykle program o bardziej zwartej strukturze (przykład 4).

Przykład 4.

Trzy przeciążone ze względu na liczbę parametrów definicje funkcji *printTime* z przykładu 3 zostały zastąpione jedną definicją z dwoma parametrami domniemanymi.

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

using namespace std;

void printTime(string t);
void printTime(int h, int m = 0, int s = 0);

void main()
{
      printTime("13:13.13");
      printTime(2);
      printTime(5, 35);
      printTime(1, 20, 45);
```

```
}

void printTime(string t)
{
	cout << "Jest godzina: " << t << endl;
}

void printTime(int h, int m, int s)
{
	cout << "Jest godzina: " << h << ":" << setfill('0')
		<< setw(2) << m << "." << setw(2) << s << endl;
}
```

Przykład powyższy demonstruje również, że możliwe jest łączenie mechanizmów przeciążenia i parametrów domyślnych w ramach definicji funkcji o jednej nazwie.

3. Typ referencyjny

W języku C++ wprowadzono nowy w stosunku do języka C wbudowany typ danych, zwany typem referencyjnym. Jest on połączeniem funkcjonalności typu wskaźnikowego ze składnią typu bezpośredniego. Definicja zmiennej typu referencyjnego tworzona jest poprzez dodanie przed nazwą zmiennej symbolu „&", analogicznie jak symbol „*" dodawany jest przed nazwą zmiennej typu wskaźnikowego. Definicji musi towarzyszyć inicjalizacja, która przypisuje referencji zmienną, na jaką będzie ona pokazywać. Tabela poniżej ukazuje porównanie podstawowych zasad użycia wskaźników i referencji.

Operacja	Wskaźnik	Referencja
Zmienne bezpośrednie	`int a, b;`	`int a, b;`
Deklaracja bez inicjalizacji	`int *ptr;`	- brak możliwości -
Deklaracja z inicjalizacją	`int *ptr = &a;`	`int &ref = a;`
Odwołanie do wartości wskazywanej	`*ptr = 12;`	`ref = 12;`
Zmiana zmiennej wskazywanej	`ptr = &b;`	- brak możliwości -

Odwołanie do zmiennej typu referencyjnego jest równoważne odwołaniu do zmiennej, na którą zmienna referencyjna wskazuje. Zmienną referencyjną można traktować jako alias do zmiennej wskazywanej, gdyż obie stanowią odwołania do tej samej lokalizacji w pamięci i podlegają jednakowym zasadom składniowym.

Zmienna referencyjna może i musi zostać zainicjalizowana w chwili tworzenia (jako zmienna globalna, statyczna, automatyczna, parametr funkcji) wskazaniem na zmienną docelową. Po inicjalizacji cel wskazania zmiennej referencyjnej nie może zostać zmodyfikowany. Nie istnieje konstrukcja składniowa, która by to umożliwiła, gdyż zmienna taka nie posiada dwóch poziomów dostępu (bezpośredni i dereferencji), jak w przypadku wskaźnika. Z tego powodu definicja zmiennej referencyjnej bez inicjalizacji wywołuje błąd kompilacji, gdyż zmienna taka byłaby bezużyteczna.

Przykład 5.

Ukazany poniżej program zawiera dwie funkcje do zamiany miejscami wartości przekazanych argumentów: *swapPtr* oraz *swapRef*. Pierwsza funkcja wykorzystuje parametry typu wskaźnikowego, druga typu referencyjnego.

```
#include <iostream>
```

```
using namespace std;

void swapPtr(int *a, int *b);
void swapRef(int &a, int &b);

void main()
{
	int x = 3, y = 7;

	cout << "x = " << x << ", y = " << y << endl;
	swapPtr(&x, &y);
	cout << "Po wywolaniu swapPtr: x = " << x << ", y = "
		<< y << endl;
	swapRef(x, y);
	cout << "Po wywolaniu swapRef: x = " << x << ", y = "
		<< y << endl;
}

void swapPtr(int *a, int *b)
{
	int t;
	t = *a;
	*a = *b;
	*b = t;
}

void swapRef(int &a, int &b)
{
	int t;
	t = a;
	a = b;
	b = t;
}
```

Porównanie funkcji *swapPtr* i *swapRef* pokazuje, że zastosowanie zmiennych typu referencyjnego, w przeciwieństwie do zmiennych typu wskaźnikowego, zupełnie nie różni się składniowo od użycia zmiennych bezpośrednich, za wyjątkiem samej deklaracji, w której występuje znak „&”.

Głównym zastosowaniem referencji w języku C++ jest przekazywanie zmiennych do i z funkcji oraz metod. W przypadku przekazania parametrów wywołania poprzez typ referencyjny, mogą być one obsługiwane wewnątrz funkcji jak zwykłe zmienne bezpośrednie (przykład 5, funkcja *swapRef*), a jednocześnie odwołują się do oryginalnych zmiennych z instrukcji wywołania, a nie do ich kopii. Ma to duże znaczenie optymalizacyjne podczas przekazywania argumentów znacznych rozmiarów, a w niektórych konstrukcjach języka C++ (np. przeładowanie operatorów) jest nieodzowne. W przypadku wartości zwracanej, obiekt może zwracać referencję na samego siebie, pozwalając w prosty sposób na wielokrotne wywoływanie swoich metod w ramach pojedynczej instrukcji – taki mechanizm zastosowano w metodach obiektów *cout* oraz *cin*, umożliwiając nieograniczone dołączanie operatorów „<<” i „>>”.

Zadania ćwiczeniowe uwzględniają jedynie proste użycie zmiennych referencyjnych jako odpowiedników zmiennych wskaźnikowych.

Zadania do samodzielnej realizacji

1. Uruchomić podane w treści instrukcji programy przykładowe i przeanalizować ich działanie.
2. Dany jest kod programu, wykorzystujący do obsługi wejścia/wyjścia funkcje z biblioteki *stdio*:

```
#include <stdio.h>

void main()
{
	int x, a, b, n = 0, t[100];

	printf("\nPodaj liczbe naturalna: ");
	scanf("%d", &x);
	a = x;
	printf("\nPodaj baze reprezentacji (2-9): ");
	scanf("%d", &b);

	if(b<2 || b>9)
	{
		printf("\nNiepoprawna wartosc bazy!!\n");
		return;
	}
	while(x)
	{
		t[n] = x % b;
		n++;
		if(n >= 100) break;
		x = x / b;
	}
	printf("\n%d(10) = ", a);
	while(n)
	{
		n--;
		printf("%d", t[n]);
	}
	printf("(%d)\n\n\n", b);
}
```

   Utworzyć program działający identycznie z powyższym (także co do formatowania wypisywanego tekstu), lecz wykorzystujący klasę *iostream*.
3. Dany jest kod programu, wykorzystujący do obsługi wejścia/wyjścia funkcje z biblioteki *stdio*:

```
#include <stdio.h>

void main()
{
	int a, p;
	float x;

	printf("\nPodaj liczbe naturalna a: ");
	scanf("%d", &a);
	printf("Podaj liczbe rzeczywista x: ");
	scanf("%f", &x);
	printf("Podaj precyzje p: ");
	scanf("%d", &p);

	printf("\nOto liczba a w kilku reprezentacjach: ");
	printf("a = %d(10) = %o(8) = %X(16)", a, a, a);
	printf("\noraz liczba x z precyzja %d: x = %.*f\n\n", p, p, x);
```

```
}
```

Utworzyć program działający identycznie z powyższym (także co do formatowania wypisywanego tekstu), lecz wykorzystujący klasę *iostream*.

4. Utworzyć program, wykorzystujący klasę *iostream*, który wypisze na ekranie tabliczkę mnożenia.
5. Utworzyć zbiór funkcji przeciążonych o nazwie *potega*, które będą zwracać wartość typu *float*, określoną według następującej specyfikacji:

   - w przypadku podania jednego argumentu, jego wartość podnoszona będzie do kwadratu,
   - w przypadku podania dwóch argumentów: jeśli drugi z nich będzie liczbą całkowitą, wartość pierwszego argumentu będzie podnoszona do podanej potęgi całkowitej; jeśli będzie liczbą zmiennoprzecinkową, wypisany zostanie monit „nie potrafie tego obliczyc”.

   Pierwszym z argumentów może być dowolna wartością liczbową (całkowitą lub zmiennoprzecinkową). Należy przyjąć $0^0 = 1$. Utworzyć program, który pozwoli sprawdzić poprawność działania funkcji.
6. Określić, które z podanych niżej zestawów deklaracji funkcji przeciążonych lub/i zawierających parametry domyślne są poprawne (jednoznaczne w dopasowaniu wywołania), a które nie, i dlaczego. Zaproponować przykładowe modyfikacje, które uczynią błędne zestawy poprawnymi.

   a)
   ```
   void funA(float x);
   void funA(float x, float y = 5);
   void funA(float x, float y);
   ```

   b)
   ```
   int funB(float x, float y);
   char funB(int p);
   void funB(float x, float y = 3, char z);
   ```

   c)
   ```
   int funC(float x);
   char funC(float x);
   float funC(float x);
   ```

   d)
   ```
   float funD(float x = 16);
   char funD(int a, float y);
   void funD(float x, float y, int a, int b = 1);
   ```
7. Dane są dwie funkcje, wykorzystujące zmienne wskaźnikowe. Czy obydwie funkcje można zmodyfikować tak, aby wszystkie wskaźniki bezpośrednio zastąpić referencjami? Jeśli dana funkcja na to pozwala – wykonać zamianę, jeżeli nie – uzasadnić brak takiej możliwości.

   a)
   ```
   void clearTab(float *start, float *end)
   {
         do
         {
               *start = 0;
               start++;
         }
         while(start <= end);
   }
   ```

   b)

```
void setAvg(float *a, float *b)
{
	*a = (*a + *b)/2;
	*b = *a;
}
```

8. Utworzyć funkcję o prototypie *ustawKolejno(float &a, float &b, float &c)*, która uporządkuje wartości podanych argumentów, tak aby a ≤ b ≤ c. Utworzyć program wykorzystujący tą funkcję i przetestować jej działanie.

II. Podstawy programowania obiektowego

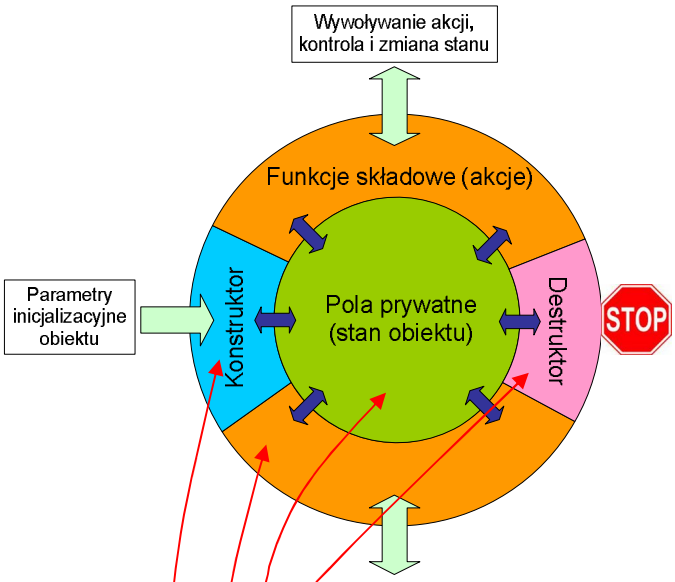
1. Klasa

W przypadku programowania strukturalnego, poznanego na przykładzie języka C, charakterystyczne jest rozdzielenie algorytmu, opisującego działania wykonywane na danych, od organizacji samych danych. Często, zwłaszcza w przypadku programów reprezentujących systemy zbudowane z wielu składników, lepszym rozwiązaniem jest odwzorowanie tych składników na poziomie kodu programu poprzez takie jego fragmenty, które integrują dane i algorytmy ich przetwarzania w nierozerwalny sposób. Takie podejście stanowi jedną z ważnych koncepcji programowania, zwaną **programowaniem obiektowym**. Umożliwienie realizacji paradygmatu programowania obiektowego jest zasadniczym rozszerzeniem języka C++ względem języka C.

W języku C++ rolę danych, przechowujących stan obiektu, mogą pełnić zmienne dowolnych typów (np. prostych typów wbudowanych, znanych z języka C, jak *char*, *int*, *float* itd). Algorytmy, przetwarzające dane obiektu i modyfikujące jego stan, realizowane są przez funkcje, które także posiadają postać znaną z języka C. Pojęciem podstawowym dla obiektowość języka C++ jest pojęcie klasy. Klasa stanowi jednostkę programu grupującą w całość wybrany zbiór danych w postaci zmiennych oraz zbiór algorytmów ich przetwarzania w postaci funkcji. Zmienne, będące składowymi klasy, nazywa się (powszechnie w programowaniu obiektowym, nie tylko w języku C++) **polami**, natomiast funkcje składowe nazywa się **metodami**. Klasę traktuje się w formalizmie języka C++ jako typ „danych", złożony i definiowany przez programistę – tzn. jest to wzorzec do tworzenia zmiennych. Składnię definicji klasy i znaczenie poszczególnych jej elementów wyjaśnia przykład 6.

Przykład 6.

Poniższy przykład ukazuje definicję klasy *Osoba* wraz z ilustracją wyjaśniającą znaczenie jej elementów.




```
class Osoba {
    // pola prywatne:
    string imie;
    string nazwisko;
    float konto;
    int kod;
public:
    // konstruktor:
    Osoba(string i, string n);
    // funkcje składowe (metody):
    void dane();
    void ustawKod(int staryKod, int nowyKod);
    void wybierz(float kwota, int tajnyKod);
    void wplac(float kwota, int tajnyKod);
    // destruktor:
    ~Osoba();
};

// definicje zadeklarowanych dla klasy funkcji:
Osoba::Osoba(string i, string n) {...}
void Osoba::dane() {...}
void Osoba::ustawKod(int staryKod, int nowyKod) {...}
void Osoba::wybierz(float kwota, int kod) {...}
void Osoba::wplac(float kwota, int kod) {...}
Osoba::~Osoba() {...}
```

Składnia definicji klasy jest podobna do składni definicji struktury w języku C. Po słowie kluczowym *class* umieszczana jest nazwa klasy, a następnie wewnątrz nawiasów { } znajduje się ciało klasy, zawierające definicje wszystkich jej składników. Definicja klasy kończy się średnikiem. W odróżnieniu od struktury znanej z języka C, klasa posiada następujące cechy dodatkowe:

a) Jej składnikami mogą być nie tylko zmienne, ale także funkcje. Dla klasy przykładowej są to funkcje o nazwach *dane*, *ustawKod*, *wybierz*, *wplac*, *Osoba*, *~Osoba*, z których dwie ostatnie (konstruktor i destruktor) mają specjalne znaczenie.

b) Ciało klasy podzielone jest na sekcje dostępu, określające poziom dostępu do zawartych w nich składowych. Każda z sekcji rozpoczyna się etykietą dostępu (*private*, *public* lub *protected*) i kończy się na kolejnej etykiecie, rozpoczynającej następną sekcję. Sekcje tego samego poziomu dostępu mogą powtarzać się wielokrotnie. Ciało klasy zaczyna się

domyślnie od poziomu dostępu *private*. Dla klasy przykładowej wszystkie pola zadeklarowane zostały jako *private* i wszystkie funkcje składowe jako *public*.

Składnik klasy zadeklarowany jako *private* dostępny jest tylko z wnętrza klasy, tj. dla jej funkcji składowych. Składnik zadeklarowany jako *public* dostępny jest bez ograniczeń z wnętrza i na zewnątrz klasy. Standardowa metoda spełnienia postulatu hermetyzacji polega na zdefiniowaniu pól jako prywatnych i metod jako publicznych, tak jest też w przykładzie 6. Zawartość pól określa stan obiektu, który jest niedostępny bezpośrednio z zewnątrz, lecz jedynie poprzez pośredniczące funkcje składowe. Funkcjami składowymi o specjalnym działaniu są konstruktor oraz destruktor, które wywoływane są automatycznie, gdy obiekt klasy jest tworzony oraz niszczony. Manipulowanie poziomami dostępu pól i metod, co do którego język C++ daje pełną swobodę, pozwala na wzmacnianie lub osłabianie poziomu hermetyzacji klasy. W ramach ćwiczeń stosować będziemy zawsze **pola zadeklarowane jako prywatne**.

Definicje funkcji składowych klasy mogą być połączone z deklaracjami tych funkcji i zawarte w ciele klasy lub też w ciele klasy mogą znajdować się tylko deklaracje (prototypy), a ich definicje (implementacje) dołączane są na zewnątrz klasy. Druga z tych możliwości stosowana jest częściej i dotyczy także klasy zaprezentowanej w przykładzie 6. Warianty te są podobne, ale nie w pełni funkcjonalnie równoważne. W pierwszym przypadku kod funkcji składowej rozwijany jest w miejscu wywołania (funkcja *inline*), co jest rekomendowane tylko dla funkcji bardzo krótkich. W drugim przypadku kod funkcji osiągany jest poprzez skok z punktu wywołania, co jest rozwiązaniem standardowym. **W ramach ćwiczenia definicje wszystkich funkcji składowych umieszczać będziemy poza ciałem klasy, jak w przykładzie 6.** W takim wypadku należy użyć specyfikacji zakresu o postaci *<nazwaKlasy>::*, poprzedzającej bezpośrednio nazwę metody, aby zaznaczyć, że chodzi o funkcję składową określonej klasy, a nie o zwykłą funkcję globalną. W przykładzie 6 pominięto zawartości kodu poszczególnych metod, gdyż nie są one istotne dla aktualnych rozważań.

2. Obiekt

Jak zaznaczono w punkcie poprzednim, klasa jest złożonym, definiowanym przez programistę typem danych – klasa jest więc wzorcem do tworzenia zmiennych. Zmienne takie nazywamy zmiennymi typu obiektowego lub krótko **obiektami**. Składnia definiowania obiektów jest analogiczna do składni definiowania pozostałym zmiennych, znanej z języka C. Przykłady pokazuje poniższa tabela.

Zmienne typu wbudowanego	Obiekty

`int maksimum, minimum;`
`char x, y, z;`
`float promien, pole;`
	`Osoba gospodarz("Jan", "Kowalski");`
	`Osoba ja("Piotr", "Kos"), on("X", "X");`

Po nazwie klasy wpisujemy nazwy definiowanych obiektów, może być ich kilka, wtedy oddzielane są przecinkami. Definicja kończy się średnikiem. Jeżeli konstruktor klasy posiada niepustą listę parametrów, musimy ich wartości podać w nawiasie bezpośrednio po nazwie definiowanego obiektu. Ponieważ konstruktor klasy *Osoba* (przykład 6) przyjmuje dwa argumenty typu *string*, dla każdego tworzonego obiektu tej klasy podawane są w nawiasie dwa łańcuchy znaków. Parametry konstruktora umożliwiają wstępną inicjalizację stanu obiektu przy jego tworzeniu. W przypadku rozważanej klasy *Osoba*, określają one imię i nazwisko osoby, którą obiekt reprezentuje.

Przykład 7.

Przykład ukazuje kompletny program, wykorzystujący zmienne typu obiektowego. Oparto się na definicji klasy podanej w przykładzie 6.

```
/***** plik: osoba.h *****/

#include <string>

using namespace std;

class Osoba {
      string imie;
      string nazwisko;
      float konto;
      int kod;

public:
      Osoba(string i, string n);
      void dane();
      void ustawKod(int staryKod, int nowyKod);
      void wybierz(float kwota, int tajnyKod);
      void wplac(float kwota, int tajnyKod);
      ~Osoba();
};

/***** plik: osoba.cpp *****/

#include <iostream>
#include "osoba.h"

using namespace std;

Osoba::Osoba(string i, string n)
{
      imie = i;
      nazwisko = n;
      konto = 0;
      kod = 0;
}

void Osoba::dane()
{
      cout << "Klient banku: " << imie << " " << nazwisko;
      cout << ", stan konta: " << konto << endl;
}

void Osoba::ustawKod(int staryKod, int nowyKod)
{
      if((staryKod == kod) && (nowyKod != 0))
      {
            kod = nowyKod;
            cout << "\tZmiana kodu powiodla sie.\n";
      }
      else
            cout << "\tZmiana kodu nie powiodla sie!\n";
}

void Osoba::wybierz(float kwota, int tajnyKod)
{
```

```
	if((tajnyKod == kod) && (tajnyKod != 0))
	{
		konto = konto - kwota;
		if(konto < 0) konto = 0;
		cout << "\tDokonano wyplaty, obecne saldo: "
			<< konto << "zl\n";
	}
	else
		cout << "\tKod jest niepoprawny!\n";
}

void Osoba::wplac(float kwota, int tajnyKod)
{
	if((tajnyKod == kod) && (tajnyKod != 0))
	{
		konto = konto + kwota;
		cout << "\tDokonano wplaty, obecne saldo: "
			<< konto << "zl\n";
	}
	else
		cout << "\tKod jest niepoprawny!\n";
}

Osoba::~Osoba() { }

/***** plik: konta.cpp *****/

#include <iostream>
#include "osoba.h"

using namespace std;

void main()
{
	Osoba klientA("Jan", "Biel");
	Osoba klientB("Anna", "Kowalska");

	klientA.dane();
	klientA.ustawKod(0, 123);
	klientA.wplac(1400, 123);
	klientA.wybierz(2000, 121);
	klientA.wybierz(2000, 123);
	cout << endl;

	klientB.dane();
	klientB.wplac(1200, 0);
	klientB.ustawKod(0, 345);
	klientB.wplac(2300, 345);
	klientB.wybierz(1100, 345);
	cout << endl;
}
```

Program składa się z trzech plików: *osoba.h*, *osoba.c* oraz *konta.c*. Pierwszy plik zawiera deklarację pól i metod klasy, drugi plik zawiera definicję metod zadeklarowanych dla klasy, trzeci plik zawiera właściwy kod programu, wykorzystujący obiekty klasy opisanej przez dwa poprzednie pliki. Jest to typowe rozwiązanie podziału na pliki programu wykorzystującego obiekty w języku C++. Z każdą klasą wiążemy jeden plik nagłówkowy (rozszerzenie *.h*) zawierający deklarację klasy i jeden plik źródłowy (rozszerzenie *.cpp*) zawierający definicję implementacji metod. Dzięki takiemu rozwiązaniu, zdefiniowaną klasę możemy następnie

użyć w wielu plikach programu, jeżeli tylko dołączymy w nich (*#include*) plik nagłówkowy definiujący klasę. W przypadku środowiska *MS Visual Studio*, pliki nagłówkowe należy umieszczać w podsekcji projektu *Header files*.

Jeżeli nie potrzeba współdzielić definicji klasy pomiędzy wiele plików, można całość zamknąć w pojedynczym pliku, umieszczając w nim najpierw deklarację klasy, dalej definicję jej metod, a następnie kod wykorzystujący obiekty zdefiniowanej klasy. **W taki sposób tworzone będą w ramach ćwiczeń wszystkie dalsze programy.**

Przykład 8.

Poniższy program jest funkcjonalnie równoważny programowi z przykładu 7, jednak cały jego kod został zawarty w pojedynczym pliku.

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

using namespace std;

class Osoba {
	string imie;
	string nazwisko;
	float konto;
	int kod;

public:
	Osoba(string i, string n);
	void dane();
	void ustawKod(int staryKod, int nowyKod);
	void wybierz(float kwota, int tajnyKod);
	void wplac(float kwota, int tajnyKod);
	~Osoba();
};

Osoba::Osoba(string i, string n)
{
	imie = i;
	nazwisko = n;
	konto = 0;
	kod = 0;
}

void Osoba::dane()
{
	cout << "Klient banku: " << imie << " " << nazwisko;
	cout << ", stan konta: " << konto << endl;
}

void Osoba::ustawKod(int staryKod, int nowyKod)
{
	if((staryKod == kod) && (nowyKod != 0))
	{
		kod = nowyKod;
		cout << "\tZmiana kodu powiodla sie.\n";
	}
	else
		cout << "\tZmiana kodu nie powiodla sie!\n";
}

void Osoba::wybierz(float kwota, int tajnyKod)
{
```

```
        if((tajnyKod == kod) && (tajnyKod != 0))
        {
                konto = konto - kwota;
                if(konto < 0) konto = 0;
                cout << "\tDokonano wyplaty, obecne saldo: "
                        << konto << "zl\n";
        }
        else
                cout << "\tKod jest niepoprawny!\n";
}

void Osoba::wplac(float kwota, int tajnyKod)
{
        if((tajnyKod == kod) && (tajnyKod != 0))
        {
                konto = konto + kwota;
                cout << "\tDokonano wplaty, obecne saldo: "
                        << konto << "zl\n";
        }
        else
                cout << "\tKod jest niepoprawny!\n";
}

Osoba::~Osoba() { }

void main()
{
        Osoba klientA("Jan", "Biel");
        Osoba klientB("Anna", "Kowalska");

        klientA.dane();
        klientA.ustawKod(0, 123);
        klientA.wplac(1400, 123);
        klientA.wybierz(2000, 121);
        klientA.wybierz(2000, 123);
        cout << endl;

        klientB.dane();
        klientB.wplac(1200, 0);
        klientB.ustawKod(0, 345);
        klientB.wplac(2300, 345);
        klientB.wybierz(1100, 345);
        cout << endl;
}
```

Podobnie jak zmienne nieobiektowe (zmienne typów prostych, tablicowych, strukturalnych, wskaźnikowych), tak również obiekty mają przydzielane oddzielne miejsca w pamięci, tzn. poszczególne pola obiektów, określające ich stan wewnętrzny, posiadają odrębne adresy (wyjątek stanowią pola statyczne, omówione w kolejnym ćwiczeniu). Tak więc, informacje dotyczące osoby *klientA* są całkowicie niezależne od informacji dotyczących osoby *klientB*.

Konstruktor wywoływany jest automatycznie zawsze, gdy tworzony jest obiekt. W przykładzie powyższym konstruktor klasy *Osoba* wywoływany jest dwukrotnie, w czasie opracowywania definicji:

```
Osoba klientA("Jan", "Biel");
Osoba klientB("Anna", "Kowalska");
```

Zadanie konstruktora polega tutaj na przypisaniu argumentów jego wywołania do pól obiektów oznaczających imię i nazwisko oraz wyzerowaniu wartości pól *konto* i *kod*. Destruktor wywoływany jest automatycznie, bezpośrednio przed usunięciem obiektu z

pamięci. W rozważanym przykładzie jego implementacja jest pusta, nie wykonuje więc żadnych działań. Pozostałe funkcje składowe (metody) wywoływane są w kodzie programu jawnie, zgodnie ze składnią *<nazwaObiektu>.<nazwaMetody>(listaParametrów)*, składnia jest zatem analogiczna, jak w przypadku dostępu do pól struktury. Dokładniejsze omówienie funkcji składowych klasy zawarte jest w kolejnym ćwiczeniu.

Należy mieć na uwadze, iż pojęcie obiektu w programowaniu obiektowym jest szerokie i ogólne. Obiekt może reprezentować każdy fragment rzeczywistości, który stanowi połączenie danych i operacji ich dotyczących. Może to być abstrakcyjny (uproszczony do odpowiedniego zakresu) opis rzeczywistego przedmiotu: samochodu, budynku, osoby; reprezentacja wybranej warstwy zarządzania: konta bankowe, zamówienia sklepowe, zbiory biblioteczne; abstrakcja matematyczna: wektory, macierze, liczby zespolone; element oprogramowania komputerowego: urządzenia wejścia/wyjścia (np. *iostream*, *cin*, *cout*), okna dialogowe, gniazda połączeń komunikacyjnych; itp.

Zadania do samodzielnej realizacji

1. Utworzyć klasę *Wektor*, której obiekty reprezentowały będą wektory trójwymiarowe. Do konstruktora jako parametry przekazywane mają być początkowe wartości składowych wektora. Klasa powinna posiadać metody: *void ustawSkaldowe(float x, float y, float z)*, *void wypisz()*, *float podajModul()*, które pozwolą odpowiednio: zmienić wartości trzech składowych, wypisać składowe wektora, obliczyć jego moduł.
2. Utworzyć klasę *Licznik*, służącą do zliczania osób obecnych w pomieszczeniu. Klasa powinna posiadać metody *void zwieksz()*, *void zmniejsz()* oraz *int liczba()*. Pierwsza z metod powinna być wywoływana, gdy kolejna osoba wchodzi do pomieszczenia, druga metoda, gdy jedna osoba wychodzi, a trzecia metoda powinna zwracać liczbę osób aktualnie obecnych wewnątrz.
3. Utworzyć klasę *Skarbonka*. Klasa powinna zawierać metody *void wrzuc(float m)* oraz *float rozbij()*. Wywołanie metody *wrzuc* oznacza dodanie do zawartości skarbonki kwoty (liczby) podanej jako jej parametr wywołania. Metoda *rozbij* powinna zwrócić sumę zgromadzonych środków, wynikającą z wcześniejszych wywołań metody *wrzuc*. Jednak wartość tej sumy ma być zwrócona tylko przy pierwszym wywołaniu metody *rozbij*, każde kolejne wywołanie powinno zwracać wartość zero.

W rozwiązaniach zadań 1, 2, 3 przygotować przykładową funkcję *main*, która pozwoli na weryfikację poprawności definicji klasy.