Instrukcja 3

Programowanie obiektowe 1

Część A

Kontenery

1. Wprowadzenie do pól i metod statycznych (klasy zawierające pola i metody statyczne)

Ponieważ język programowania C++ jest językiem obiektowym (ang. *object-oriented programming*), możliwe jest definiowanie (tworzenie) własnych, nowych typów danych, w odróżnieniu od typowych struktur z języka C, które nacechowane są wyłącznie obecnością typów podstawowych: *double*, *int*, *char* itp. W języku C++ dodatkowo można opisać szereg czynności, które realizuje się przy użyciu określonych danych [1]. Obiekt, a właściwie klasę obiektów opisują zatem czynności ukierunkowane na realizację określonych zadań (realizowane funkcje – metody).

Szkielet podstawowy klasy zawiera następujące elementy (*Listing 1*):

```
class Primary
{
     //   konstruktory
     //   destruktory
     //   pola
     //   metody
     //   właściwości
     //   indeksatory
     //   zdarzenia
     //   obiekty zagnieżdżone
};
```

W kodzie źródłowym 1 (*Listing 1*) domyślnie składowe są prywatne (*private*). Zamiast słowa kluczowego *class* można użyć *struct.*

```
struct Primary
{
     //   konstruktory
     //   destruktory
     //   pola
     //   metody
     //   właściwości
     //   indeksatory
     //   zdarzenia
     //   obiekty zagnieżdżone
};
```

W sytuacji takiej wszystkie domyślnie składowe będą publiczne (*public*).
Po utworzeniu klasy za nawiasem, a przed średnikiem można umieścić jeszcze definicje obiektów zdefiniowanej uprzednio klasy, czego przykład zaprezentowano w kodzie źródłowym 2 (*Listing 2*):

[1] Do tego celu w języku C wykorzystuje się funkcje globalne.

```
 struct Primary
{
     // ...
} jeden, dwa, trzy;
```

Ponadto każda klasa stanowi odrębną przestrzeń nazw. Użytkownik może zdefiniować nowe typy danych wewnątrz danej klasy, podobnie jak i nowe aliasy dla nazw typów (z wykorzystaniem instrukcji *typedef*). Do zdefiniowanych w ten sposób typów i aliasów można odwołać się z zewnątrz z zastosowaniem operatora zasięgu „::”. Definicję klasy można utworzyć na zewnątrz klas i funkcji, choć możliwe jest również zdefiniowanie klasy wewnątrz danej funkcji.

Wszystkie składowe klasy są widoczne wewnątrz danej klasy dla metod stanowiących funkcje składowe. Dla przypomnienia, dostępność określa się za pomocą słów kluczowych: *private*, *public* i *protected* w sposób zaprezentowany w przykładowym kodzie źródłowym nr 3 (*Listing 3*) [2]:

```
     class Klasa
     {
          int aa;       // domyślny dostep private
                        // (bez uzycia specyfikatora
                        // private)
public:
          int bb;
          double cc;
private:
          float dd;
          void ee (float, float);
     };
```

W kodzie nr 3 (*Listing 3*) występują trzy sekcje (dwie prywatne oraz jedna publiczna). W następstwie tego prywatne będą pola *aa*, *dd* oraz funkcja (metoda) *ee*, z kolei publiczne będą pola *bb* oraz *cc*. Dany specyfikator obejmuje swoim zasięgiem wszystkie następujące po nim części klasy aż do końca klasy lub napotkania innego specyfikatora [3].

Klasę można postrzegać zatem jako typ, wzorzec – matrycę, według której tworzone są kolejne instancje, czyli obiekty programu. W języku C++ klasa pierwszorzędowa funkcjonuje dokładanie tak samo jak typ wbudowany, a wszelkie składowe należące do tej samej klasy są do siebie podobne z racji dostępnych pól i metod [4].

[2] Specyfikator *private* umożliwia dostęp do nazw zmiennych i metod tylko z poziomu danej klasy. Dostęp do nazw zmiennych i metod nie będzie możliwy spoza danej klasy. Specyfikator *protected* posiada identyczne właściwości jak *private* za wyjątkiem dziedziczenia (słowo kluczowe *protected* pozwala na dziedziczenie nazw zmiennych i metod danej klasy, które uprzednio sklasyfikowane jako *private* nie byłyby dziedziczone). Dostęp do nazw składowych oznaczonych jako publiczne (*public*) możliwy jest z każdego miejsca w programie, jeśli tylko widoczna jest definicja klasy.

[3] Standard C++ nie wprowadza limitu specyfikatorów.

[4] Różnica dotyczyć może jedynie wartości poszczególnych pól, realizowane metody zawsze są takie same.

1.1 Pola i metody jako składowe klasy

Obiekty mogą zawierać zarówno zmienne, czyli pola, jak również mogą realizować na sobie określone funkcje, zwane metodami. Zmienne należące do wskazanych typów obiektowych nazywane są obiektami. Istnieje kilka sposobów tworzenia obiektów, z których najprostszy sposób opiera się na tworzeniu obiektu w sposób analogiczny do typowej struktury (*Listing 4*):

```
Klasa AAA;
```

Pola klasy określa definicja danych, z których składa się dany obiekt klasy. Niestatyczne pole klasy może posiadać typ wskaźnika do obiektu tej klasy. W większości przypadków niedozwolone jest inicjowanie zmiennych w definicji pola, co przedstawiono w kodzie źródłowym nr 5 (*Listing 5*).

```
class Klasa
{
       float    aa, bb;
       double cc = 4.45; // Takie zainicjowanie spowoduje blad
                         // kompilacji

       // dalsza czesc ciala klasy
};
```

Samo zdefiniowanie klasy nie wpływa na utworzenie jakiegokolwiek obiektu. Deklaracja niestatycznego pola spowoduje, że każdy obiekt danej klasy będzie posiadał zmienną o nazwie i typie opisanym w deklaracji tej klasy. Użytkownik może zadeklarować pola zarówno przed, jak i po metodach wewnątrz danej klasy, przy czym kolejność nie jest wymuszona przez standard C++ . Trzeba jednak pamiętać, że kolejność ma znaczenie podczas procesu inicjowania i destrukcji, gdyż elementy są inicjowane zgodnie z kolejnością ich deklaracji, zaś usuwane w kolejności przeciwnej.

Szczególny rodzaj pól stanowią pola statyczne [5]. Ich deklaracja jest możliwa poprzez użycie słowa kluczowego *static* jako specyfikatora. Specyfikator ten pozwala na utworzenie tylko jednej zmiennej o określonej nazwie, która będzie dostępna w zasięgu danej klasy. Ze zmiennej tej można korzystać nawet w sytuacji, gdy żaden z obiektów danej klasy nie został utworzony, co pozwala zauważyć, że zmienne statyczne działają w sposób niezależny od obiektów (zmiennych) klasy. Aby utworzyć zmienną statyczną danej klasy w pierwszej kolejności należy ją zadeklarować w klasie, a następnie zdefiniować poza tą klasą. Jako że nazwa pola statycznego należy do zasięgu klasy, dostęp do zmiennej statycznej na zewnątrz klasy możliwy jest poprzez użycie operatora zasięgu klasy „::”.
W kodzie źródłowym nr 6 zaprezentowano przykład deklaracji i definicji zmiennej statycznej klasy (*Listing 6*):

[5] Zmienne statyczne są tworzone (i inicjalizowane) tuż po załadowaniu programu do pamięci, zanim rozpocznie się wykonywanie funkcji głównej *main* (analogia do zmiennych globalnych).

```
class Secondary
{
static int aa;  // deklaracja zmiennej statycznej
static float bb;  // deklaracja zmiennej statycznej
// dalsza czesc ciala klasy };
int Secondary::aa;      // definicja zmiennej statycznej poza
                        // definicja klasy –
                        // domyslne inicjowanie zerem
float Secondary::bb = 7.45;  // definicja zmiennej statycznej poza
                             // definicja klasy –
                             // inicjowanie okreslona wartoscia
```

W odróżnieniu od typowych zmiennych, składowe statyczne mogą być definiowane i inicjalizowane również w miejscu ich deklaracji wewnątrz danej klasy, ale tylko pod warunkiem, że ich wartość jest ustalona z użyciem specyfikatora typu *const*.

W kodzie źródłowym nr 7 zamieszczono przykład definicji i inicjalizacji statycznych składowych stałych (*Listing 7*):

```
class Secondary
{
static const int aa;
static const float bb=2.23;
// dalsza czesc ciala klasy
};
const int Secondary::aa=7;
```

Składowa *bb* jest tworzona i zainicjalizowana po deklaracji wewnątrz danej klasy. Ponieważ składowa *aa* wewnątrz danej klasy jest tylko zadeklarowana, należy ją jeszcze zdefiniować i zainicjalizować (standard języka wymusza inicjalizację stałych w momencie ich tworzenia).

Istnieją dwie podstawowe możliwości odwoływania się do zmiennych statycznych klasy. Pierwszy, zaprezentowany powyżej sposób dotyczy użycia pełnej nazwy kwalifikowanej (np. *Secondary::aa*) drugi – podobnie jak dla normalnych (niestatycznych) składowych, opiera się na użyciu nazwy dowolnego obiektu tej klasy, a następnie operatora wyboru składowej („strzałka" – dotyczy nazwy będącej nazwą wskaźnika do obiektu lub kropki dla nazwy będącej nazwą obiektu). Z racji unikalności (niepowtarzalności) zmiennej statycznej (jest tylko jeden egzemplarz tej zmiennej) nie ma znaczenia, który ze sposobów zostanie wykorzystany.

Składową statyczną może być również obiekt danej klasy, gdyż składowa statyczna (niepowtarzalny egzemplarz danej składowej statycznej) nie współtworzy obiektu danej klasy. Należy przez to rozumieć, że składowe statyczne są przypisane do klasy lub struktury jako całości, a nie do jej poszczególnych instancji (obiektów). Poniżej w kodzie źródłowym nr 8 (*Listing 8*) zaprezentowano przykład użycia obiektu danej klasy jako zmiennej statycznej:

```
class Wspolrzedne_odcinka
{
    double x, y; // niestatyczne pola x, y – wspolrzedne odcinka
```

```
    static Wspolrzedne_odcinka srodek;   // pole statyczne – srodek,
                                         // klasy
                                        // Wspolrzedne_odcinka
};
Wspolrzedne_odcinka Wspolrzedne_odcinka::srodek;
        // definicja składowej statycznej
       // zadeklarowanej w definicji klasy
      // Wspolrzedne_odcinka
```

Z praktycznego punktu widzenia pola statyczne stosuje się wówczas, jeżeli potrzebne są klasy zliczające liczbę swoich instancji (liczniki obiektów), parametry są takie same (wspólne) dla wszystkich obiektów należących do danej klasy (np. stałe liczbowe, jednostki miary, kursy walut, ceny towarów i usług, deskryptor pliku) lub niezbędne jest korzystanie z flag do komunikacji obiektów.

Statyczność metod opiera się na ich uniezależnieniu od jakiegokolwiek obiektu danej klasy. Oznacza to, że istnieje możliwość wywołania metod poprzedzonych słowem kluczowym *static* bez konieczności posiadania instancji danej klasy. Odbywa się to jednak kosztem braku dostępu do składowych niestatycznych (zarówno pól, jak i metod) danej klasy, z których korzystanie wymaga obecności obiektu [6]. Deklaracja metody ma postać deklaracji funkcji wewnątrz danej klasy. Podobnie jak dla typowych funkcji, definicja może być połączona z deklaracja, ewentualnie definicja może znajdować się na zewnątrz klasy. Wówczas w definicjach jej metod należy korzystać z nazw kwalifikowanych. Składowe niestatyczne, nie będące konstruktorem, wywołuje się „na rzecz" wcześniej istniejącego obiektu klasy. Wywołanie danej metody na zewnątrz klasy (z funkcji nie będącej funkcją składową tej samej klasy) może być realizowane poprzez zmienną będącą obiektem tej klasy lub referencją do obiektu tej klasy, albo też poprzez wskaźnik na obiekt tej klasy.

Standard C++ umożliwia tworzenie przez użytkownika publicznego konstruktora (kompilator nie dołączy wówczas niejawnie konstruktora domyślnego), jak również zdefiniowanie konstruktora prywatnego. W momencie utworzenia konstruktora prywatnego, utworzenie obiektu klasy nie będzie możliwe. Jest to działanie celowe podczas implementacji wzorca projektowego *Singleton*, o czym będzie mowa w następnej części instrukcji.

1.2 Implementacja singletonów

Składowe statyczne ułatwiają implementację singletonów, służąc do kontroli unikalnego obiektu. Wynika to z faktu, że są one niepowtarzalne w skali całej klasy, podobnie jak niepowtarzalny jest pojedynczy obiekt singleton w konkretnym momencie działania aplikacji [7].

[6] W języku C++ *this* stanowi nazwę wskaźnika na obiekt w stosunku do którego wykonywana jest określona metoda. Nazwą obiektu będzie zatem **this*. W metodach statycznych nie ma dostępu do wskaźnika *this*, reprezentującego bieżący obiekt klasy. Słowo kluczowe *this* może być zatem wykorzystywane wyłącznie w metodach (niestatycznych funkcjach składowych), konstruktorach i destruktorach klasy.

[7] Korzystanie z singletonu gwarantuje, że dana klasa będzie miała maksymalnie jedną instancję. Oznacza to, że będzie ona reprezentowana wyłącznie przez jeden obiekt. Singleton można wykorzystać zatem w komponentach dostępnych dla wielu użytkowników (np. w celu udostępnienia wartości domyślnych kilku aplikacjom).

Wzorzec projektowy singleton stanowi klasę, której jedyna instancja (obiekt) spełnia najważniejszą rolę w całym programie (singletony przechowują najważniejsze dane globalne aplikacji, wykonują kluczowe czynności w programie, stanowiąc nadrzędny obiekt wobec wszystkich innych obiektów). Dostęp do singletonów jest możliwy również bez użycia obiektu głównego, za pomocą własnych zmiennych globalnych singletonu.

W celu implementacji singletonu, w pierwszej kolejności należy dokonać deklaracji statycznego pola, które będzie przechowywało wskaźnik na dany obiekt (*Listing 9*).

```
class CSingleton     // klasa singletonu
{
private:
static CSingleton* Third; // statyczne pole, przechowujące wskaźnik
                          // na obiekt singletonu
// dalsza czesc ciala klasy
};
```

W module kodu znaleźć się powinien nagłówek z definicją klasy. Należy także zainicjować pole wartością zerową (*Listing 10*).

```
CSingleton* CSingleton::Third= NULL;
```

Umieszczenie deklaracji w sekcji *private* zabezpiecza pole przed przypadkową, niepożądaną modyfikacją. Aby mieć dostęp do niego należy zatem utworzyć metodę umożliwiającą dostęp do składowej, co zaprezentowano w kodzie źródłowym nr 11 (*Listing 11*):

```
class CSingleton // klasa singletonu, dodanie kodu zrodlowego
{
public:
static CSingleton* Obiekt()
{
       // obiekt tworzymy, jeżeli wczesniej nie istnial –
       // wskaźnik Third ma wartość początkową zero)
if (Third == NULL) CSingleton();
return Third;        // zwracamy wskaźnik na obiekt
}
```

Metoda ta pozwala zatem stwierdzić czy dany obiekt istnieje (jeżeli nie istnieje, to zostanie utworzony), a dodatkowo zostanie zwrócony wskaźnik do tego obiektu.

2. Wprowadzenie do kontenerów i iteratorów (wykorzystanie kontenerów w praktyce)

Dotychczas studenci kierunku informatyka poznali jeden kontener będący częścią standardowej biblioteki szablonów języka C++ (ang. Standard Template Library STL), a mianowicie kontener *vector*. Teraz przyszedł czas na poznanie kolejnych.

2.1. Lista dwukierunkowa

Lista dwukierunkowa, czyli klasa *list* jest kolejnym kontenerem udostępnianym przez bibliotekę STL. Dostęp do elementów listy dwukierunkowej jest możliwy tylko przy użyciu iteratorów w odróżnieniu od wektora, do którego elementów dostęp jest możliwy zarówno przy pomocy iteratorów, jak i przez podanie indeksu. Dodawanie i usuwanie elementów listy jest szybsze niż dodawanie i usuwanie elementów wektora, poza tym operacje te w przypadku listy realizowane w stałym czasie w odróżnieniu od wektora, dla którego czas dostępu zależy od usytuowania elementu w kontenerze. Trzecią, dość istotną różnicą pomiędzy klasą *list* oraz klasą *vector* jest to, że adresy elementów listy są niezmienne, natomiast adresy elementów wektora ulegają dość częstym zmianom.

Sposób korzystania z list bardzo przypomina korzystanie z wektorów. Oto przykład wykorzystania listy:

```
#include <list>       // Nagłówek <list> zawiera deklarację
                           // std::list

using namespace std;

int main()
{
     // Tak wygląda utworzenie pustej listy typu int i
     // nazwanie jej lista:
     list <int> lista;

     // Aby umieścić na końcu listy pojedynczy element,
     // używamy funkcji "push_back():"
     lista.push_back(1);

     // Aby usunąć pojedynczy element z końca listy, używam
     // funkcji "pop_back()":
     lista.pop_back();

     // Funkcja "push_front()" umieszcza pojedynczy element na
     // początku listy:
     lista.push_front(0);  // Teraz pierwszym elementem jest
                                // 0, a nie 1

     // Podobnie, funkcja "pop_front()" usuwa pierwszy element
     // listy:
     lista.pop_front();
     return 0;
}
```

2.2. Iteratory

Iteratory usprawniają pracę związaną z obsługą kontenerów. Poza tym umożliwiają dotarcie do danego składnika pojemnika bez konieczności poznawania jego struktury. Posługiwanie się iteratorem przypomina używanie zwykłych wskaźników. Jest on jednak czymś więcej niż wskaźnik – w przeciwieństwie do wskaźników, korzystając z iteratora nie musimy martwić się czy przypadkiem nie przekroczyliśmy zakresu pojemnika ani czy poprawnie wskazuje on na wybrany element. Tym wszystkim zajmuje się sam iterator, co pozwala programiście skupić się na innych problemach w trakcie tworzenia programu.

Oto przykład wykorzystania iteratorów do wyświetlania elementów listy dwukierunkowej:

```
#include <iostream>
#include <list>

using namespace std;

int main ()
{
  list<int> lista;

  // Inicjujemy listę kolejnymi liczbami naturalnymi:
  lista.push_back(1);
  lista.push_back(2);
  lista.push_back(3);

  // Wyświetlenie składników listy w pętli przy pomocy iteratora przejścia w
  // przód:
  list<int>::iterator it;
  for( it=lista.begin(); it!=lista.end(); ++it )
  {
    cout<< *it <<'\n';
  }

  return 0;
}
```

Metody begin() i end() skonstruowane są do przeglądania kontenera od początku do końca. Jeśli chcemy działać na składowych listy w odwrotnej kolejności, korzystamy w tym celu z metody rbegin(), która zwraca odwrócony iterator wskazujący na ostatni element pojemnika (mówi się także, że jest to odwrócony początek). Odwołuje się on do elementu bezpośrednio poprzedzającego iterator wskazywany przez end. Jest to odwrócony iterator bezpośredniego dostępu. Mamy także metodę rend(), która zwraca odwrócony iterator do elementu odwołującego się do elementu bezpośrednio poprzedzającego pierwszy element kontenera list (zwany także odwróconym końcem). rend() wskazuje miejsce bezpośrednio poprzedzające składnik do którego odwoływałby się begin(). Oto przykład:

```
#include <iostream>
#include <list>

using namespace std;

int main ()
{
  list<int> lista;

  // Inicjujemy listę kolejnymi liczbami naturalnymi:
  lista.push_back(1);
  lista.push_back(2);
  lista.push_back(3);

  // Wyświetlenie składników listy w pętli przy pomocy iteratora przejścia w
  // przód:
  list<int>::reverse_iterator it;
  for( it=lista.rbegin(); it!=lista.rend(); ++it )
  {
    cout<< *it <<'\n';
  }

  return 0;
}
```

2.3. Zbiory

Zbiory są jednym z kontenerów biblioteki STL, których struktura oparta jest na drzewach. Elementy, które są w nich przechowywane są posortowane, według pewnego klucza. Drzewiasta struktura zapewnia szybkie wyszukiwanie, jednak są z tym związane także pewne mankamenty, mianowicie modyfikacja elementu jest możliwa tylko w taki sposób, że kasujemy stary element, a następnie wstawiamy w to miejsce nowy. Zbiory są tzw. kontenerami asocjacyjnymi (o zmiennej długości, pozwalającymi na operowanie elementami przy użyciu kluczy). Oto przykład wykorzystujący kontener *set*:

```
#include<iostream>
#include<set>

using namespace std;

int main()
{
  set<int> zbior;
  zbior.insert(4);
  zbior.insert(3);
  zbior.insert(1);
  zbior.insert(2);

  set<int>::iterator it;
```

```
  for( it=zbior.begin(); it!=zbior.end(); ++it )
    cout<<*it<<'\n';

  return 0;
}
```

2.4. Mapy

Mapy są posortowanymi kontenerami asocjacyjnymi, czyli zbiornikami o zmiennej długości gromadzącymi dane, które można dodawać i usuwać. Nie można jednak dodawać danych na konkretną pozycję, ponieważ kolejność ustalana jest według danego klucza. Mapa jest również parowym zbiornikiem asocjacyjnym, czyli jej elementami są pary wartości klucz i dana. Pierwszej wartości (first), czyli klucza mapy, nie można zmieniać, natomiast druga wartość, czyli wartość danej (second) jest możliwa do zmiany. Mapa jest w końcu unikalnym kontenerem asocjacyjnym, co oznacza, że każdy element ma indywidualny klucz.
Oto przykład wykorzystujący kontener *map*:

```
#include<iostream>
#include<map>

using namespace std;

int main()
{
  map<int, string> miesiac;
  miesiac[1] = "styczen";
  miesiac[2] = "luty";
  miesiac[3] = "marzec";
  miesiac[4] = "kwiecien";
  miesiac[5] = "maj";
  miesiac[6] = "czerwiec";
  miesiac[7] = "lipiec";
  miesiac[8] = "sierpien";
  miesiac[9] = "wrzesien";
  miesiac[10] = "pazdziernik";
  miesiac[11] = "listopad";
  miesiac[12] = "grudzien";

  cout << "5 miesiac to:  " << miesiac[5] << '\n';

  map<int, string>::iterator cur;

  // zwrócenie elementu o kluczu 5
  cur = miesiac.find(5);

  // pierwszy sposób dostępu do klucza i danej
  cout<<cur->first<<" miesiąc to: "<<cur->second<<endl;
```

```
  // drugi sposób dostępu do klucza i danej
  cout<<(*cur).first<<" miesiąc to: "<<(*cur).second<<endl;

  // elementy o kluczach większych i mniejszych
  map<int, string>::iterator prev = cur;
  map<int, string>::iterator next = cur;
  ++next;
  --prev;

  cout << "Wczesniejszy, czyli "<<(*prev).first<<" element mapy to " << (*prev).second <<
'\n';
  cout << "Nastepny, czyli "<<next->first<< " element mapy to "<< next->second << '\n';

  return 0;
}
```

Instrukcja 3

Programowanie obiektowe 1

Część B

Przeciążanie operatorów

1. Wprowadzenie do przeciążania operatorów

Oprócz słów kluczowych i typów, operatory stanowią podstawowy element każdego języka programowania wysokiego poziomu (Python, Haskel czy C++). Korzystając z języka programowania C++ użytkownik ma do dyspozycji typy wbudowane (int, float, unsigned, itd.), jak również może utworzyć własne typy przechowujące różne typy danych (struktury, klasy). O ile w specyfikacji standardu C++ zdefiniowano zestaw operatorów, o tyle tworzenie nowych operatorów w przeciwieństwie do nowych typów nie jest możliwe. Istnieje jednak możliwość przeciążania operatorów już istniejących dla czynności realizowanych na obiektach typu klasa, co umożliwia w razie potrzeby odmienne od typowego działanie operatorów. Operatory można zatem przeciążać, podobnie jak inne funkcje i metody języka C++.

Przeciążanie operatorów (ang. *operator overloading*), nazywane zamiennie również **przeładowaniem**, pozwala nadawać operatorom nowe znaczenie, dzięki czemu mogą być one wykorzystane w odniesieniu do obiektów zdefiniowanych klas (implementacja klasy przechowującej złożony typ danych z przeciążonymi operatorami).

Przeciążania operatora można dokonać na dwa sposoby. Pierwszym jest wykorzystanie niezależnej funkcji (na ogół zaprzyjaźnionej z jedną lub większą liczbą klas). Przykładowo dla operatora dwuargumentowego o nazwie „przeciazenie” otrzymujemy:

x przeciazenie y

Wyrażenie to jest tożsame z wywołaniem:

operator przeciazenie (x , y).

Drugi ze sposobów opiera się na użyciu funkcji składowej, co odpowiada wywołaniu:

x.operator przeciazenie (y)

Aby funkcja globalna posiadała dostęp do danych prywatnych klasy, funkcja operatorowa musi być zadeklarowana jako funkcja zaprzyjaźniona z klasa np.:

friend nazwa_funkcji *operator* * (liczba_1, liczba_2)

W tym celu konieczne jest umieszczenie prototypu funkcji w definicji klasy, który poprzedzony jest słowem kluczowym *friend.* Jako że do funkcji zaprzyjaźnionych nie jest przekazywany wskaźnik *this*, konieczne staje się przekazanie operandów w sposób jawny, co wymusza przekazanie dwóch argumentów dla funkcji operatorowej dwuargumentowej,

w odróżnieniu od przeładowania za pomocą zwykłej funkcji składowej klasy. Zgodnie ze standardem C++, w sytuacji, gdy operator dwuargumentowy przeładowany jest za pomocą funkcji składowej klasy, jawnie należy przekazać do niego tylko jeden argument. Drugi z argumentów przekazywany jest w sposób niejawny poprzez wskaźnik *this*.

Operatory przeciąża się poprzez uprzednie zdefiniowanie funkcji operatorowej. Jako argumentów operatorów używa się typów zadeklarowanych zewnętrznie (class, enum, struct, union). Istnieje zasada, że obiekt, który spowodował wywołanie funkcji operatora zawsze umiejscowiony jest po lewej stronie operacji, zaś obiekt występujący po prawej stronie przekazywany jest do funkcji w postaci argumentu.

W przypadku, gdy operator modyfikuje operandy należy go zdefiniować jako funkcję składową klasy. Jako przykład można podać operatory: „ = ” , „ += ”, „ -= ”, „ ++ ”, itp. W innym przypadku operator powinien być zdefiniowany jako funkcja globalna lub zaprzyjaźniona. Jako przykład można podać m.in. operatory: „ + ”, „ - ”, ” == ”.
Przykładowa implementacja przeciążenia operatora mnożenia liczb ułamkowych z użyciem operatorowej funkcji składowej klasy ma postać:

```
ulamek ulamek :: operator * (ulamek a)
{ulamek wynik;
wynik.licznik = licznik * a.licznik;
wynik.mianownik = mianownik * a.mianownik;
return wynik;}
```

1.1 Wykaz operatorów możliwych do przeładowania

W ramach realizacji niniejszej instrukcji laboratoryjnej studenci zapoznają się z mechanizmem przeciążania operatorów arytmetycznych, przypisania i porównania. Ponadto celem jest realizacja przeciążania operatorów strumienia wejścia >> i wyjścia <<.

− >	Operator dostepu (Structure memeber pointer reference)
New	Dynamicznie alokowana pamięć (Dynamically allocate memory)
Delete	Dynamicznie usuwana pamięć (Dynamically deallocated memory)
++	Inkrementacja (Increment)
−	Dekrementacja (Decrement)
-	Minus (Unary minus)
!	Logiczna negacja (Logical negation)
~	Dopełnienie logiczne (One's complement)
*	Dereferencja (Indirection)
*	Mnożenie (Multiplication)
/	Dzielenie (Division)
%	Modulo (Modulus (remainder))
+	Dodawanie (Addition)
-	Odejmowanie (Subtraction)
<<	Przesunięcie (Left shift)
>>	Przesunięcie (Right shift)
<	Mniej niż (Less than)
<=	Mniej niz lub rowne (Less than or equel to)
>	Większe niż (Greater than)
>=	Większe niż lub równe (Greater than or equel to)
==	Równe (Equal to)
!=	Różne (Not equal to)
&&	Logiczne AND (Logical AND)
||	Logiczne OR (Logical OR)
&	Bitowe AND (Bitwise AND)
^	Bitowe XOR (Bitwise exlusive OR)
|	Bitowe OR (Bitwise inclusive OR)
=	Przypisanie (assignment)
+= -= *=	Przypisanie (assignment)
/= %= &=	Przypisanie (assignment)
^= | =	Przypisanie (assignment)
<<=>>=	Przypisanie (assignment)

Tab. 1. Wykaz operatorów możliwych do przeładowania

[Źródło: P. Mikołajczyk, „Język C++ Podstawy Programowania", Lublin 2011, s. 145]

Oprócz wskazanych powyżej operatorów przeciążyć jeszcze można operator wywołania funkcji (ang. *Function call*): „ () "; operator elementu tablicy (ang. *Array element*): „ [] " oraz operator przecinka (ang. *Comma*): „ , ". W ten sposób, przedstawiony w instrukcji zbiór operatorów do przeciążenia jest zbiorem kompletnym.

O ile każdy z wymienionych wyżej operatorów można przeciążyć (przeładować) w dowolny sposób, tak aby realizował inną operację, o tyle istnieją pewne pisane reguły

mówiące o tym, że operator jednoargumentowy musi pozostać operatorem unarnym (związanym z jednym operandem), zaś operator dwuargumentowy – operatorem binarnym (związanym z dwoma operandami). W praktyce nie wszystkie operatory można przeciążyć. Jako przykład można podać: operator kropki „ . ”, operator zasięgu „ :: ”, operator wyłuskania wskaźnika do składowej „ .* ”, operator rozmiaru „ sizeof ” czy operator warunkowy „ ?: ”. Poza tym standard C++ uniemożliwia tworzenie nowych symboli operatorów, modyfikowanie łączności i kolejności operatorów. Ponadto operator musi być zdefiniowany w taki sposób, aby co najmniej jeden członek klasy stanowił jego operand, albo też operator był członkiem klasy.

1.2 Przeciążanie operatora <<

Załóżmy, że mamy do dyspozycji klasę *Punkt* składającą się z trzech prywatnych współrzędnych typu *int*. Współrzędne punktu można zainicjować przy pomocy odpowiedniego konstruktora lub przy użyciu klawiatury, wykorzystując w tym celu metodę *wprowadz*(). Natomiast wyświetlić współrzędne na ekranie można przy pomocy metody *wyswietl*(). Aby tego dokonać wykorzystać należy między innymi operatory strumienia wejściowego oraz wyjściowego. Oto przykład:

```
#include <iostream>

using namespace std;

class Punkt{
        int x,y,z;
public:
        Punkt (int wart_x=0, int wart_y=0, int wart_z=0):
        x(wart_x), y(wart_y), z(wart_z){
        cout<<"Punkt (0, 0, 0) został powołany do życia"<<endl;
        }

        void wprowadz (){
        cout<<"Wprowadz wspolrzedne punktu: x y z"<<endl;
```

```
        cin>>x>>y>>z;
        }

        void wyswietl(){
        cout<<"Oto wsoplrzedne punktu (x, y, z)"<<endl;
        cout<<"("<<x<<", "<<y<<", "<<z<<")"<<endl;
        }
};

int main(void)
{
        Punkt p;
        p.wprowadz();
        p.wyswietl();
        return 0;
}
```

Alternatywny sposób wyprowadzania wartości współrzędnych na ekran polega na przeciążeniu operatora strumienia wyjściowego << (który jest de facto przeciążonym operatorem przesunięcia bitowego w lewo), aby odpowiednio wykorzystać możliwości klasy *ostream* do wyświetlenia składowych punktu na ekranie komputera. Poniżej zamieszczono przykład ilustrujący przeciążenie operatora << w celu wyświetlenia współrzędnych punktu na ekranie komputera:

```
#include <iostream>

using namespace std;

class Punkt{
        int x,y,z;
public:
        Punkt (int wart_x=0, int wart_y=0, int wart_z=0):
        x(wart_x), y(wart_y), z(wart_z){
        cout<<"Punkt (0, 0, 0) został powołany do życia"<<endl;
```

```
        }

        void wprowadz (){
        cout<<"Wprowadz wspolrzedne punktu: x y z"<<endl;
        cin>>x>>y>>z;
        }

        friend ostream & operator<<(ostream & wyjscie , Punkt & p){
        wyjscie<<"Oto wspolrzedne punktu (x, y, z)"<<endl;
        wyjscie<<"("<<p.x<<", "<<p.y<<", "<<p.z<<")"<<endl;
        return wyjscie;
        }
};

int main(void)
{
        Punkt p;
        p.wprowadz();
        cout<<p;
        return 0;
}
```

1.3 Przeciążanie operatora >>

Przeładowanie operatora strumienia wejściowego >> (który jest de facto przeciążonym operatorem przesunięcia bitowego w prawo), w celu wprowadzenia współrzędnych punktu p z klawiatury, odbywa się analogicznie jak w przypadku przeciążania operatora operatora <<, z tą różnicą, że tym razem będziemy wykorzystywać klasę *istream*. Oto przykład ilustrujący takie właśnie przeciążenie:

```
#include <iostream>

using namespace std;
```

```
class Punkt{
        int x,y,z;
public:
        Punkt (int wart_x=0, int wart_y=0, int wart_z=0):
        x(wart_x), y(wart_y), z(wart_z){
        cout<<"Punkt (0, 0, 0) został powołany do życia"<<endl;
        }

        friend istream & operator>>(istream & wejscie , Punkt & p){
        wejscie>>p.x>>p.y>>p.z;
        return wejscie;
        }

        friend ostream & operator<<(ostream & wyjscie , Punkt & p){
        wyjscie<<"Oto wspolrzedne punktu (x, y, z)"<<endl;
        wyjscie<<"("<<p.x<<", "<<p.y<<", "<<p.z<<")"<<endl;
        return wyjscie;
        }
};

int main(void)
{
        Punkt p;
        cout<<"Wprowadz wspolrzedne punktu: x y z"<<endl;
        cin>>p;
        cout<<p;
        return 0;
}
```

Z przykładów zamieszczonych w niniejszej instrukcji wynika elastyczność języka C++, który umożliwia dość znaczną swobodę w przeciążaniu dozwolonych operatorów. Ograniczeniem tej swobody jest inwencja programisty oraz czytelność kodu, która przykładowo nie powinna dopuszczać przeciążania operatora dodawania tak, żeby za jego pomocą wykonywane było odejmowanie.

Instrukcja 3

Programowanie obiektowe 1

Część C

Wyjątki

1. Wprowadzenie do szablonów funkcji i klas (templates)

Szablony *(ang. Templates)* w innych obiektowych językach programowania (Java, C++) nazywane też **Typami generycznymi** *(ang. Generic Types)* lub **Typami uogólnionymi**, to element języka C++ pozwalający na tworzenie kodu niezależnego od typów, algorytmów oraz struktur danych. Szablony są techniką realizacji polimorfizmu na innym poziomie niż za pomocą funkcji wirtualnych i dziedziczenia.

Mechanizm szablonów można rozumieć jako pewną formę makrodefinicji (preprocesora). Szablony ze względu na szerokie rozpowszechnienie się zastosowań biblioteki **STL** *(ang. Standard Template Library)* uważane są za jedną z najważniejszych właściwości języka C++. Dzięki szablonom mamy możliwości **programowania ogólnego** *(ang. generic programming)*. Tworząc szablon funkcji (podobnie jak szablon klasy) tworzymy kod działający na nieopisanych jeszcze typach – funkcja działa na typach ogólnych (które nie istnieją w języku C++), w momencie wywołania funkcji pod te typy ogólne podstawiane są konkretne typy, jak np. **int** czy **char**. Przekazujemy szablonowi typy jako parametry, podczas kompilacji następuje tak zwana **konkretyzacja szablonu** *(ang. template instantiation)*, podczas której kompilator na podstawie typów danych przekazanych wzorcowi generuje docelowy kod do obsługi danego typu.

Przykłady implementacji tej samej funkcji dla różnych typów (bez użycia szablonów):

```
// Wersja 1 int:
void Zamień(int &x, int &y) {
int temp = x;
x = y;
y = temp;
}

// Wersja 2 double:
void Zamień(double &x, double &y) {
double temp = x;
x = y;
y = temp;
}
```

```
// Wersja 3 string:
void Zamień(string &x, string &y) {
string temp = x;
x = y;
y = temp;
}
```

Dzięki wykorzystaniu szablonów programista C++ może skupić się bardziej na algorytmach niż na danych, które są przetwarzane. Użycie szablonów pozwala zredukować ilość nadmiarowego kodu, ponieważ jedną funkcjonalność można zaprogramować dla wielu typów danych. Kompilator w kolejnym kroku precyzuje określony typ oraz dostosowuje odpowiednio wywołania funkcji, metod czy tworzenia obiektów na podstawie klas.

Wadą działanie szablonów w języku C++ jest to, że przypominają one bardzo zaawansowane makra preprocesora. Powoduje to, że kompilator ma zasadnicze trudności z wygenerowaniem prawidłowych, czytelnych komunikatów diagnostycznych w przypadku błędnego użycia poprawnego szablonu.

Szablony w C++ możemy podzielić na:

- **Szablony funkcji**,
- **Szablony klas**.

1.1. Szablony funkcji

Szablon funkcji to mechanizm umożliwiający automatyczną generację funkcji. Jest to schemat, według którego postępuje kompilator. Programista dostarcza jedynie ogólny zarys ciała funkcji bez określania konkretnych argumentów wywołania. Na tej podstawie kompilator jest w stanie wytworzyć dowolną ilość funkcji działających identycznie, a różniących się jedynie typem argumentów funkcji. Szablony zezwalają na definiowanie całych rodzin funkcji, które następnie mogą być używane dla różnych typów argumentów.

Definicja szablonu funkcji wygląda następująco:

```
template <typename T> T MojaFunkcja(T a, T b, ...) {
//instrukcje
...};
lub
template <class T> T MojaFunkcja(T &a, T &b, ...) {
//instrukcje
...};
```

Definicja umieszczona bezpośrednio przed definicją funkcji informuje, że funkcja będzie korzystała z fikcyjnego typu o nazwie T. Definicja dotyczy tylko pojedynczej funkcji zdefiniowanej bezpośrednio po frazie „template".

Przykłady implementacji funkcji dla różnych typów z użyciem szablonów:

```
// Listing 1
template <typename T> T Sprawdz(T a, T b) {
return (a>b)?a:b;
};
// Listing 2
template <typename T> T Minimum(T a, T b, T c) {
if (a <= b && a <= c) return(a);
if (b <= a && b <= c) return(b);
return(c);
};
// Listing 3
template <class T> void Zamień(T &x, T &y) {
T temp = x;
x = y;
y = temp;};
```

Wyrażenie template <typename T> lub template <class T> oznacza, że mamy do czynienia z szablonem, który posiada jeden parametr formalny nazwany T. Słowo kluczowe typename/class oznacza, że parametr ten jest typem (nazwą typu) lub klasy. Nazwa tego parametru może być następnie wykorzystywana w definicji funkcji w miejscach, gdzie spodziewamy się nazwy typu/klasy. I tak powyższe wyrażenie (*Listing 1*) definiuje funkcję, która przyjmuje dwa argumenty typu T i zwraca wartość typu T, będącą wartością większego z dwu argumentów. Typ T jest na razie niewyspecyfikowany. W tym sensie szablon definiuje całą rodzinę funkcji. Konkretną funkcję z tej rodziny wybieramy poprzez podstawienie za formalny parametr T konkretnego typu będącego argumentem szablonu. Takie podstawienie nazywamy konkretyzacją szablonu. Argument szablonu umieszczamy w nawiasach <> za nazwą szablonu.

1.2. Szablony klas

Na podobnej zasadzie co szablony funkcji można także definiować szablony klas (inne określenia: wzorce klas, klasy ogólne). Szablony klas są podobne do makr, tyle że wykonywane są przez kompilator, a nie przez preprocesor. Cechują je pewne własności,

których jednak nie ma preprocesor, np. można tworzyć rekurencyjne wywołania. Podobnie jak w przypadku szablonów funkcji, szablon klasy definiuje nam w rzeczywistości całą rodzinę klas.

Definicja szablonu klasy wygląda następująco:

```
template <class T> class MojaKlasa {
//definicja klasy };
```

Definicja umieszczona bezpośrednio przed definicją klasy informuje, że klasa będzie korzystała z fikcyjnego typu o nazwie T. Definicja dotyczy tylko pojedynczej klasy zdefiniowanej bezpośrednio po frazie „template".

Przykłady implementacji i użycia klas z użyciem szablonów:

```
// Listing 4
template <class T> class c_klasa {
public:
T zmienna;
c_klasa(T l) { zmienna = l; }
void wyswietl() { cout << zmienna << endl; }
};

int main() {
c_klasa<int> calkowita(2);
calkowita.wyswietl();
c_klasa<char> znak('b');
znak.wyswietl();
}

// Listing 5
template <class T> class c_klasa {
public:
T zmienna;
c_klasa(T l);
void wyswietl();
};

template <class T>
c_klasa<T>::c_klasa(T l) : zmienna(l) {}
template <class T>
void c_klasa<T>::wyswietl() { cout << zmienna << endl; }
```

```
int main() {
c_klasa<int> calkowita(2);
calkowita.wyswietl();
c_klasa<char> znak('b');
znak.wyswietl();
}
```

2. Wyjątki (exceptions)

Wyjątki (ang. Exceptions) to mechanizm obsługi sytuacji wyjątkowych (zazwyczaj błędów). W trakcie działania programu zawsze może dojść do powstania sytuacji nietypowej, trudnej do uniknięcia. Można jednak przewidywać, iż taka sytuacja wystąpi i się na nią odpowiednio przygotować. Tak jak Java, C# i wiele innych języków obiektowych, język C++ umożliwia obsługę wyjątków, czyli sytuacji, gdy powstaje w czasie wykonania programu błąd uniemożliwiający dalszy jego przebieg (na przykład dzielenie przez zero, wywołanie metody za pomocą pustego wskaźnika, błąd odczytu/zapisu strumienia). Normalnie, powstanie takiej sytuacji powoduje przerwanie programu, zwykle z mało czytelnym komunikatem o naturze błędu. Obsługa wyjątków pozwala na zdefiniowanie przez programistę akcji, które należy podjąć po powstaniu sytuacji wyjątkowej (w szczególności programista może świadomie podjąć decyzję o ich całkowitym zignorowaniu).

W języku C++ wprowadzono mechanizm pozwalający programiście na reagowanie na sytuacje błędne czy nietypowe. Gdy taka sytuacja wystąpi, programista może wygenerować wyjątek. Wyjątek to **obiekt** pewnej klasy. Wygenerowanie wyjątku polega na przekazaniu obiektu opisującego wyjątek z fragmentu kodu, w którym wystąpił problem, do fragmentu, w którym przewidziano jego obsługę. Wygenerowanie (zgłoszenie) wyjątku powoduje przerwanie wykonywania sprawiającego problemy kodu i przejście do obsługi sytuacji problematycznej. Obsługa ta może znajdować się w innym miejscu kodu. Wyjątek jest obiektem, jego klasa określa typ sytuacji wyjątkowej. Obiekt może w sobie posiadać pola oraz funkcje składowe, pozwalające na sprecyzowanie informacji o zaistniałej sytuacji wyjątkowej.

Wyjątki w C++ możemy podzielić na:

- **Wyjątki własne**,
- **Wyjątki standardowe**.

2.1. Blok try...catch

Do obsługi wyjątków wykorzystuje się blok **try...catch**. Definicja bloku **try...catch** wygląda następująco:

```
try {
     // "Wyrzucenie" wyjątku:
     if(coś poszło nie tak)
          throw wyjątek;
     // KOD
}
catch (typ nazwa) {
     // Obsługa wyjątku
}
```

- **throw** – jest operatorem, po nim następuje wyrażenie, które określi rodzaj wyjątku, mogą być różne typy/klasy wyjątków,
- **catch** – może pojawić się tylko po **try** lub po innym **catch**, w nawiasie określamy typ wyjątku i zmienną/obiekt, która przekazuje informacje o wyjątku.

Jeżeli do obsługi błędu wystarczy sam fakt jego wykrycia, to podajemy typ nie podając zmiennej/obiektu. Blok po catch nazywamy *procedurą obsługi wyjątku*. Procedura obsługi wyjątku może znajdować się w tej samej funkcji, w której następuje zgłoszenie wyjątku.

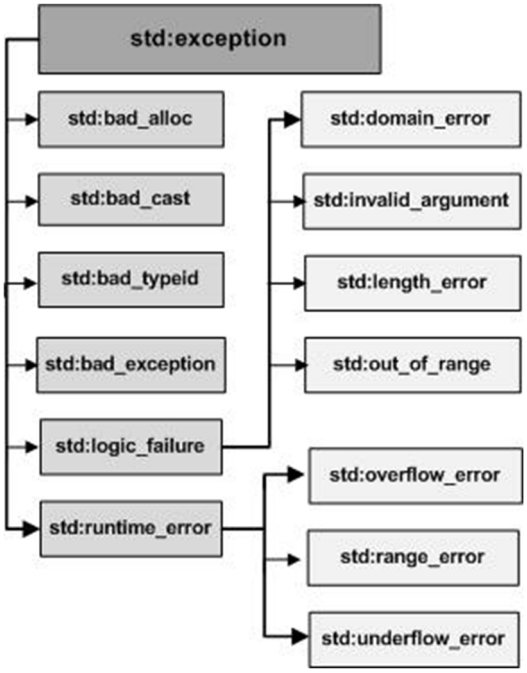
Wyjątki wysłane w zakresie instrukcji **try**, ale nieprzechwycone przez odpowiedni blok **catch** zostają odebrane przez funkcję obsługi z parametrem *„wielokropek”* **catch(...)**. Taki blok łapie wszystkie błędy (pozostałe niepasujące do innych bloków catch błędy).

```
try {
     // KOD
}
catch (int ex) {
  cout << "wyjątek int";
}
catch (float ex) {
  cout << "wyjątek float";
}
catch (...) {
  cout << "wyjątek domyślny";
}
```

Przykłady implementacji i użycia bloku try...catch:

```
// Listing 1
try {
     int n;
     cout << "Podaj liczbę: ";
     cin >> n;

     // Czy ujemna?
     if (n < 0)     throw 1;
     // Czy zero?
     if (n == 0)    throw 2;
     // Czy za duża?
     if (n > 100000) throw 3;

     // Wszystko OK:
     cout << "Kwadrat liczby = " << n*n << endl;
}
catch (int x) {
     cout << "Wyjątek nr " << x << endl;
}

// Listing 2
int FunkcjaWczytująca() {
     try {
          int A, B;
          cout << "Podaj dwie liczby: ";
          cin >> A >> B;

          if (A < 0)     throw "Liczba A ujemna!";
          if (B < 0)     throw "Liczba B ujemna!";
          if (B == 0)    throw "Dzielenie przez zero!";

          int wynik = A / B;

          if (wynik == 0)     throw "Wynik zerowy!";
          cout << "WYNIK = " << wynik << endl;
     }
     catch (char *s) {
          cout << "WYJĄTEK: " << s << endl;
     }
}
```

W języku C++ można zdefiniować własne klasy wyjątków (tworząc własne klasy lub dziedzicząc po standardowej klasie **exception**), albo do opisu wyjątku wykorzystać już istniejące typy/klasy standardowe. W bibliotekach klas korzystających z wyjątków zazwyczaj korzysta się ze specjalnych klas do określania rodzaju wyjątków. Stosowanie klas jako typów wyjątków jest wygodne, możemy tworzyć klasy o nazwach opisujących typy wyjątków. Klasa standardowa **std::exception** to klasa bazowa wyjątków wykorzystywanych przez wiele klas standardowych. Najważniejsza metoda klasy to metoda **what()**, która zwraca tekstowy opis wyjątku.



Przykłady implementacji i użycia własnej klasy wyjątku:

```
// Listing 3
#include <iostream>
#include <exception>

using namespace std;

class MojWyjatek: public exception {
  virtual const char* what() const throw() {
    return "Moj wyjatek";
  }
};
```

```
int main () {
  MojWyjatek myex;

  try  {
    throw myex;
  }
  catch (exception& e) {
    cout << e.what() << endl;
  }
  return 0;
}
```

Przykłady implementacji i użycia klasy standardowej wyjątku:

```
// Listing 4
#include <exception>
using namespace std;


try {
     int* myarray = new int[1000];
}
catch (exception& e) {
     cout << "Wyjątek: " << e.what() << endl;
}
```

Zadania do wykonania

1. Napisać program, który będzie implementował stos przy pomocy kontenera list. W szczególności chodzi o umożliwienie wykonywanie takich operacji jak umieszczanie elementu na stosie, zdejmowanie elementu ze stosu, wyświetlanie zawartości stosu, przy użyciu prostego menu. Należy zabezpieczyć ten program tak, aby obsługiwał wyjątki.
2. Napisać program, który będzie implementował listę uczniów w klasie przy pomocy kontenera set. W szczególności chodzi o umożliwienie wykonywanie takich operacji jak umieszczanie ucznia na liście, usuwanie ucznia z listy, wyświetlanie listy uczniów, przy użyciu prostego menu. Należy zabezpieczyć ten program tak, aby obsługiwał wyjątki.
3. Napisać program, który będzie implementacją słownika angielsko-polskiego przy pomocy kontenera map. W szczególności chodzi o umożliwienie wykonywanie takich operacji jak umieszczanie słowa angielskiego wraz z odpowiednim tłumaczeniem w słowniku, wyszukanie słowa angielskiego i wyświetlenie odpowiednika polskiego, wyświetlenie zawartości słownika na ekranie, przy użyciu prostego menu. Należy zabezpieczyć ten program tak, aby obsługiwał wyjątki.
4. Napisać program z wykorzystaniem funkcji operatora jako funkcji składowej klasy, będącej przeciążonym operatorem mnożenia. Program powinien umożliwić mnożenie dwóch dowolnych ułamków oraz zwracać ułamek będący iloczynem tych ułamków.
5. Zrealizować zadanie 4 z wykorzystaniem przeciążonego operatora mnożenia jako niezależnej operatorowej funkcji globalnej (zaprzyjaźnionej z klasą). Dodatkowo uzupełnić program tak, aby zaprezentować przeciążanie operatorów strumienia wejściowego i wyjściowego.
6. Napisać program wyliczający pierwiastki równania kwadratowego w postaci:

$$ax^2 + bx + c = 0,$$

Program powinien pytać użytkownika o podanie trzech parametrów **a**, **b** oraz **c**. W programie należy wykorzystać obsługę wyjątków zabezpieczającą program przed nietypowymi sytuacjami jak: *dzielnie przez zero*, *pierwiastek z liczby ujemnej* itp. itd.