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;
}
```

3. Zadania do wykonania

1. Napisać aplikację konsolową służącą do zamiany jednostek (z kilograma na angielskie funty [8]). Użytkownik powinien wprowadzić liczbę rzeczywistą określającą liczbę kilogramów do konwersji oraz zaimplementować kod do wyświetlania niezbędnych komunikatów w programie głównym.

2. Zmodyfikować kod źródłowy z aplikacji 1, aby program wykorzystywał w swoim działaniu klasy z polami i metodami niestatycznymi do konwersji jednostek miary kg na funty. Wyświetlanie komunikatów powinno zostać przypisane do metody danej klasy.

3. Zmodyfikować kod źródłowy z aplikacji 2, aby program wykorzystywał w swoim działaniu klasy z polami i metodami statycznymi do konwersji jednostek miary kg na funty. Wyświetlanie komunikatów powinno zostać przypisane do metody danej klasy.

   Uwaga: Statyczne dane muszą być zdefiniowane poza definicją klasy. Definicja może wystąpić tylko raz w całym kodzie programu.

4. Zmodyfikować kod źródłowy z aplikacji 2, aby program wykorzystywał w swoim działaniu tylko jedną klasę o nazwie „StaleMatematyczne” zawierającą tylko jedno statyczne pole. Wyświetlanie stosownych komunikatów powinno nastąpić w programie głównym.

   Uwaga: Statyczne dane muszą być zdefiniowane poza definicją klasy. Definicja może wystąpić tylko raz w całym kodzie programu.

[8] Kilogram jest określany jako równy masie międzynarodowego prototypu kilograma przechowywanego w Międzynarodowym Biurze Miar i Wag w miejscowości Sèvres pod Paryżem. Warto przypomnieć, że stanowi on jedyną jednostkę z układu *SI*, którą jako wzorzec (w odróżnieniu do właściwości fizycznych) definiuje przedmiot fizyczny. Funt wykorzystywany jest przede wszystkim w systemie imperialnym jako jednostka masy, a zarazem akceptowana jednostka wagi (siła grawitacji działająca na dowolny obiekt).

5. Zmodyfikować kod źródłowy z aplikacji 3, aby program wykorzystywał w swoim działaniu klasy z polami i metodami statycznymi do konwersji jednostek miary kg na funty (w tym pole stałe z użyciem specyfikatora *const*). Wyświetlanie komunikatów powinno zostać przypisane do metody danej klasy.

   Uwaga: Deklaracja stałości musi wystąpić zarówno w definicji, jak i w deklaracji danej metody.

6. Zapoznać się z działaniem singletonu w oparciu o przykładowy kod źródłowy zamieszczony w instrukcji laboratoryjnej nr 1.

7. 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.

8. 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.

9. 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.