



**POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI**

**KIERUNEK STUDIÓW
INFORMATYKA**

***MATERIAŁY DO ZAJĘĆ
LABORATORYJNYCH***

Zaawansowane programowanie obiektowe (C++)

Autor:
mgr inż. Monika Kaczorowska

Lublin 2021

INFORMACJA O PRZEDMIOCIE

Cele przedmiotu:

- Cel 1. Zapoznanie studentów z zaawansowanymi technikami programowania obiektowego w C++.
- Cel 2. Nabycie umiejętności tworzenia efektywnego kodu z wykorzystaniem bibliotek w C++.

Efekty kształcenia w zakresie umiejętności:

- Efekt1. Potrafi posługiwać się dokumentacją opisującą biblioteki języka C++, wyszukiwać niezbędne informacje w literaturze.
- Efekt 2. Potrafi tworzyć aplikacje obiektowe w sposób nowoczesny z wykorzystaniem możliwości oferowanych programistom w zasobach bibliotek języka C++.
- Efekt 3. Potrafi zaprojektować algorytm postawionego zadania, wybrać i zastosować w praktyce właściwy sposób organizacji prac programistycznych, w tym technikę testowania aplikacji.

Literatura do zajęć:

- Literatura podstawowa
 1. Prata S., Szkoła programowania. Język C++, Helion, Gliwice, 2013.
 2. Josuttis N. M., C++ Biblioteka standardowa. Podręcznik programisty, Helion, Gliwice, 2003.
 3. Aleksandrescu A., Nowoczesne programowanie w C++. Uogólnione implementacje wzorców projektowych, Helion, Gliwice, 2011..
- Literatura uzupełniająca
 1. Aleksandrescu A., Sutter H., Język C++. Standardy kodowania. 101 zasad, wytycznych i zalecanych praktyk, Helion, Gliwice, 2005.
 2. Models and implementation of network communication in Windows USING Languages: C++ and Delphi / Elżbieta Miłosz. [W]: Varia Informatica 2011.2011, s. 225-252.
 3. Analiza porównawcza narzędzi RAD do wizualnego programowania w języku C++ / Elżbieta Miłosz, Tetiana Pasikova // Jcsi - Journal of Computer Sciences Institute. 2016, vol. 2, s. 76-80.

Metody i kryteria oceny:

- Oceny częściowe:
 - Przygotowanie merytoryczne do zajęć laboratoryjnych na podstawie: wykładów, literatury, pytań kontrolnych do zajęć.
 - Dwa kolokwia: próg zaliczeniowy 51% z każdego z kolokwiów.
 - Realizacja zadań praktycznych w trakcie zajęć laboratoryjnych.
- Ocena końcowa - zaliczenie przedmiotu:
 - Pozytywne oceny częściowe.
 - Ewentualne dodatkowe wymagania prowadzącego zajęcia.

Plan zajęć laboratoryjnych:

Lab1.	Właściwości programowania obiektowego: hermetyzacja, dziedziczenie, polimorfizm.
Lab2.	Wielokrotne wykorzystanie kodu. Polimorfizm dynamiczny (metody wirtualne).
Lab3.	Wielokrotne wykorzystanie kodu. Polimorfizm statyczny (szablony funkcji i klas).
Lab4.	STL. Kontenery sekwencyjne.
Lab5.	STL. Kontenery asocjacyjne.
Lab6.	Programowanie generyczne z wykorzystaniem biblioteki Boost.
Lab7.	Programowanie generyczne z wykorzystaniem biblioteki Qt.
SLab8.	Kolokwium 1.
Lab9.	Obsługa WE/WY. Obsługa błędów i wyjątków.
Lab10	Klasa string i elementy bibliotek STL i Boost. Wyrażenia regularne.
Lab11.	Zarządzanie pamięcią. Stosowanie inteligentnych wskaźników.
Lab12.	Wyrażenia i funkcje lambda. Krotki w C++.
Lab13.	Testowanie w C++.
Lab14.	Kolokwium 2.
Lab15.	Zaliczenie końcowe .

Zadania realizowane podczas laboratorium będą implementowane w środowisku *Code::Blocks* lub *Qt*.



LABORATORIUM 1. WŁAŚCIWOŚCI PROGRAMOWANIA OBIEKTOWEGO: HERMETYZACJA, DZIEDZICZENIE, POLIMORFIZM.

Cel laboratorium:

Przypomnienie wiadomości o programowaniu obiektowym.

Zakres tematyczny zajęć:

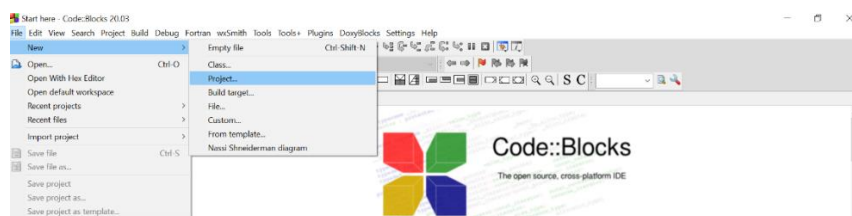
- zapoznanie się ze środowiskiem programistycznym *Code::Blocks* oraz *Qt*,
- tworzenie pliku nagłówkowego oraz źródłowego dla klasy,
- dziedziczenie,
- tablice obiektów.

Pytania kontrolne:

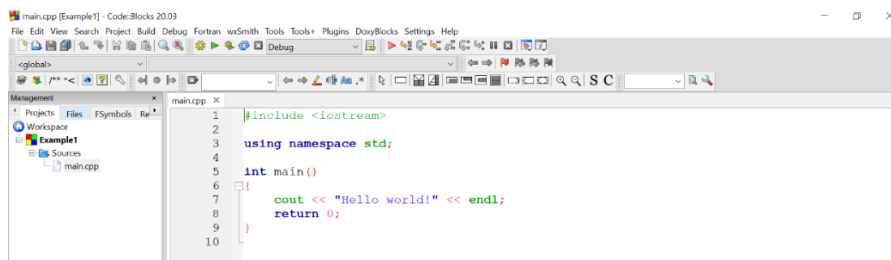
1. Co to jest plik nagłówkowy oraz źródłowy?
2. Na czym polega dziedziczenie?
3. Jakie są rodzaje tablic przechowujących obiekty i czym się różnią?
4. Jakie są modyfikatory dostępu?

Środowisko programistyczne:

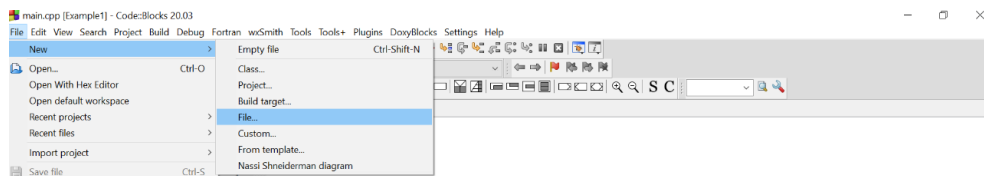
Podczas laboratoriów zadania będą implementowane z języku C++ w środowisku programistycznym *Code::Blocks* lub *Qt*. Dla każdej klasy należy stworzyć plik nagłówkowy, który będzie zawierać definicję klasy oraz deklarację zmiennych i metod. Plik źródłowy powinien zawierać definicję metod, które zostały zadeklarowane w odpowiadającym mu pliku nagłówkowym. Na rysunkach 1.1.-1.8. przedstawione zostało w jaki sposób należy tworzyć projekt, plik nagłówkowy oraz źródłowy w *Code::Blocks*. Rysunek 1.1. pokazuje w jaki sposób należy stworzyć projekt. Następnie po wybraniu *new Project* należy wybrać *Console Application*, język C++ oraz nazwę projektu. Rysunek 1.2. przedstawia widok po utworzeniu projektu. Po utworzeniu projektu można przystąpić do tworzenia pliku nagłówkowego. Rysunek 1.3. pokazuje w jaki sposób dodać nowy plik do projektu. Na rysunkach 1.4. oraz 1.5. przedstawione zostało tworzenie pliku nagłówkowego. Na rysunku 1.6. przedstawiony został widok środowiska po dodaniu pliku nagłówkowego. Do projektu dodany został folder *Headers*. W bardzo podobny sposób należy postąpić z tworzeniem pliku źródłowego. Na rysunku 1.7. zaprezentowane zostało tworzenie pliku źródłowego.



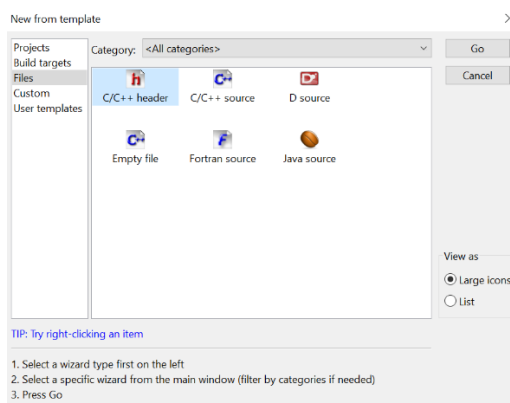
Rys. 1.1. Tworzenie projektu w *Code::Blocks*



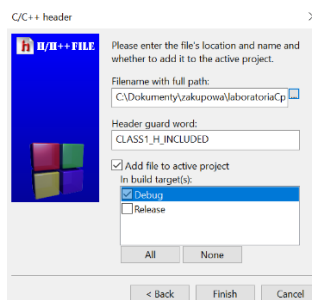
Rys. 1.2. Widok po utworzeniu projektu w Code::Blocks



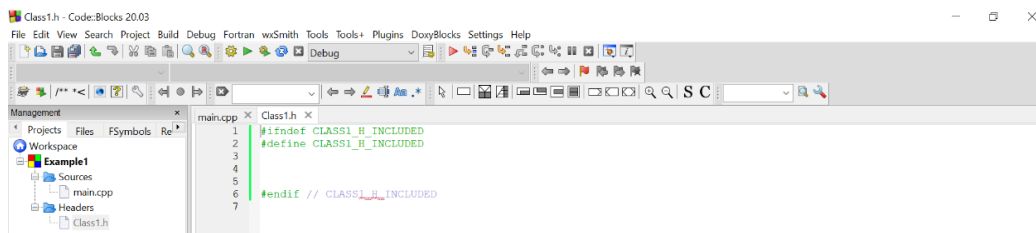
Rys. 1.3. Dodawanie nowego pliku do projektu



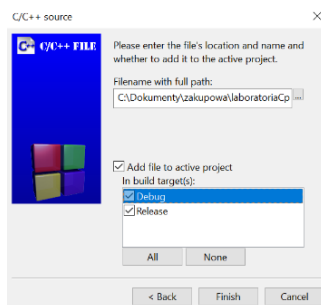
Rys. 1.4. Tworzenie pliku nagłówkowego



Rys. 1.5. Tworzenie pliku nagłówkowego – ciąg dalszy



Rys. 1.6. Widok po dodaniu pliku nagłówkowego

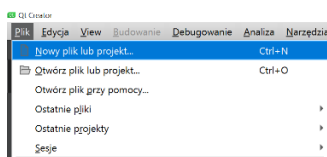


Rys. 1.7. Tworzenie pliku źródłowego

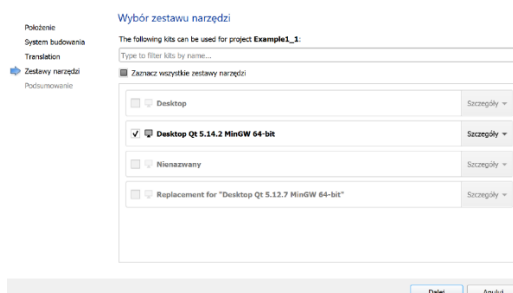


Rys. 1.8. Widok po stworzeniu pliku źródłowego

Rysunki 1.9. – 1.15. przedstawiają tworzenie projektu, pliku nagłówkowego i źródłowego w środowisku *Qt*. Rysunek 1.9. przedstawia w jaki sposób stworzyć projekt w *Qt*. Po wybraniu projektu należy wybrać rodzaj projektu: *Qt Widget Application* lub *Aplikacja konsolowa Qt*. Jeśli nie będziemy tworzyć aplikacji okienkowej, to wybieramy *Aplikacja konsolowa Qt*. Ważne jest, aby przy tworzeniu projektu wybrać kompilator – rysunek 1.10. Rysunek 1.11. zawiera widok po stworzeniu projektu. W celu dodania nowego pliku do projektu należy kliknąć na nazwę projektu prawym przyciskiem myszy i wybrać *Add new*, co zostało pokazane na rysunku 1.12. W środowisku *Qt* jest możliwość dodania od razu pliku nagłówkowego i źródłowego dla klasy. Można również oddzielnie dodać plik nagłówkowy i źródłowy. Jeśli chcemy jednocześnie stworzyć plik nagłówkowy oraz źródłowy, należy wybrać opcję zaznaczoną na rysunku 1.13. Na rysunku 1.14. zaprezentowane zostało tworzenie klasy i dwóch plików dla niej: nagłówkowego oraz źródłowego. Rysunek 1.15. przedstawia projekt wraz z dodanym plikiem nagłówkowym oraz źródłowym dla klasy.



Rys. 1.9. Tworzenie projektu w *Qt*

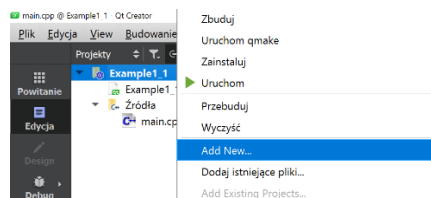


Rys. 1.10. Wybór kompilatora

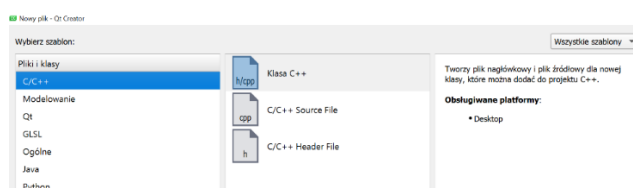




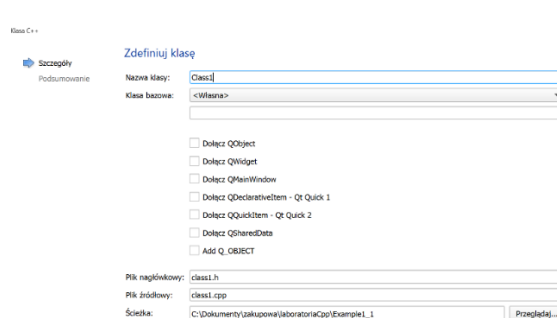
Rys. 1.11. Widok po stworzeniu projektu



Rys. 1.12. Dodawanie nowego pliku



Rys. 1.13. Dodawanie pliku – ciąg dalszy

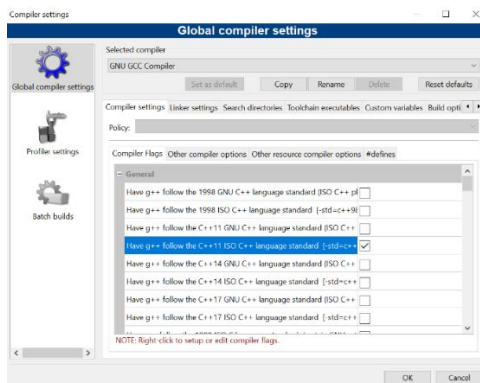


Rys. 1.14. Definiowanie klasy



Rys. 1.15. Widok projektu z plikiem nagłówkowym oraz źródłowym

Zarówno w środowisku *Code::Blocks* jak i *Qt* należy pamiętać o ustawieniu standardu języka, w tym przypadku C++11. Na Rysunku 1.16. zaprezentowane zostało w jaki sposób ustawić standard kompilatora w środowisku *Code::Blocks*. Należy wejść z zakładkę *Settings* a następnie wybrać *Compiler*. W celu ustawienia standardu języka w środowisku *Qt* należy dodać *CONFIG+=c++11* do pliku z rozszerzeniem *pro*. Na rysunku 1.17. został zaprezentowany ten plik.



Rys. 1.16. Ustawienie standardu kompilatora w Code::Blocks



Rys. 1.17. Ustawienie standardu kompilatora w Qt

Powtórzenie wiadomości o dziedziczeniu:

Dziedziczenie umożliwia wykorzystanie nowych klas w oparciu o klasy, które już istnieją. Klasa, która dziedziczy po innej klasie nazywana jest klasą pochodną lub dzieckiem. Natomiast klasa, po której dziedziczy inna klasa nazywana jest klasą bazową lub rodzicem. Klasa pochodna tworzona jest aby opisać w bardziej szczegółowy sposób klasę bazową. Klasa pochodna rozszerza klasę bazową. Każdy obiekt klasy pochodnej jest obiektem klasy bazowej. Dzięki mechanizmowi dziedziczenia nie trzeba powielać kodu. Można powiedzieć, że dzięki dziedziczeniu wspólne, powtarzające się fragmenty kodu są „wyciągane przed nawias”. Dziedziczenie może być typu: *public*, *protected* lub *private*. Z reguły dziedziczenie jest typu *public*. W tej sytuacji, jeśli w klasie bazowej znajdują się zmienne i metody opatrzone modyfikatorem dostępu *private*, to nie będzie do nich dostępu w klasie pochodnej, podobnie będzie z modyfikatorem dostępu *protected*. Tylko zmienne oraz metody z modyfikatorem dostępu *public* będą dostępne z poziomu klasy pochodnej. Jeśli nie chcemy, aby było możliwe dziedziczenie po jakiejś klasie, to po nazwie tej klasy trzeba dodać słowo *final* (na przykład: `class Example final`).

Jako przykłady dziedziczenia można podać:

- Osoba jako klasa bazowa, Nauczyciel, Student, Lekarz jako klasy pochodne;
- Zwierzę jako klasa bazowa, Ptak, Małpa, Pies jako klasy pochodne;
- Pojazd jako klasa bazowa, Samochód osobowy, Traktor, Autobus jako klasy potomne.

Listingi 1.1.-1.5. prezentują przykład dziedziczenia. Stworzone zostały dwie klasy: *Person* oraz *Teacher*. Klasa *Person* opisuje osobę, a klasa *Teacher* opisuje nauczyciela. Klasa *Teacher* dziedziczy po klasie *Person* rozszerzając klasę *Person* o dodatkowe zmienne oraz metody. Na

listingu 1.1. przedstawiony został plik nagłówkowy klasy *Person*. Każda osoba posiada imię, nazwisko oraz wiek (*name*, *surname*, *age*). Listing 1.1. zawiera:

- linijki 8 – 10 zawierają pola klasy;
- linijka 12 zawiera konstruktor z argumentami;
- linijka 13 zawiera konstruktor bezargumentowy;
- linijki 14 -17 zawierają metody.

```
1  #ifndef PERSON_H_INCLUDED
2  #define PERSON_H_INCLUDED
3  #include <iostream>
4  using namespace std;
5  class Person
6  {
7  private:
8      string name;
9      string surname;
10     int age;
11 public:
12     Person(string name1, string surname1, int age1);
13     Person();
14     void setAge(int age1);
15     string getSurname();
16     bool is_18();
17     void showInfoPerson();
18 };
19 #endif // PERSON_H_INCLUDED
```

Listing 1.1. Plik nagłówkowy Person.h

Na listingu 1.2. przedstawiony został plik źródłowy klasy *Person* i zawiera on definicję konstruktorów oraz metod klasy *Person*.

```
1  #include<iostream>
2  #include"Person.h"
3  using namespace std;
4  Person::Person(string name1, string surname1, int age1)
5  {
6      name=name1;
7      surname=surname1;
8      age=age1;
9      cout<<"Konstruktor klasy bazowej - Person"<<endl;
10 }

11 Person::Person()
12 {
13     cout<<"Konstruktor bez. klasy bazowej - Person"<<endl;
14 }

15 void Person::setAge(int age1)
```

```

16 {
17     age=age1;
18 }

19 string Person::getSurname()
20 {
21     return surname;
22 }

23 bool Person::is_18()
24 {
25     if(age>=18)
26         return true;
27     else
28         return false;
29 }

30 void Person::showInfoPerson()
31 {
32     cout<<"Imie: "<<name<<" nazwisko: "<<surname<<
33         "wiek: "<<age<<endl;
34 }

```

Listing 1.2. Plik źródłowy *Person.cpp*

Na listingu 1.3. przedstawiony został plik nagłówkowy klasy *Teacher*. Nauczyciel cechuje się dodatkowo tytułem naukowym oraz doświadczeniem (*title*, *experience*). Pogrubioną czcionką zostały zaznaczone instrukcje które realizują dziedziczenie. Listing 1.3. zawiera:

- linijka 4 – klasa *Teacher* dziedziczy publicznie po klasie *Person*;
- linijki 10 -11 - konstruktor zawierający argumenty klasy bazowej oraz swoje, czyli *experience* i *title*. Po dwukropku następuje wywołanie klasy bazowej czyli po tej po której klasa dziedziczy;
- linijka 12 – konstruktor bezargumentowy. W tym przypadku nie trzeba wywoływać konstruktora bezargumentowego klasy *Person*;
- linijki 13 – 16 – zawierają metody.

```

1  #ifndef TEACHER_H_INCLUDED
2  #define TEACHER_H_INCLUDED
3  #include"Person.h"
4  class Teacher: public Person
5  {
6  private:
7      int experience;
8      string title;
9  public:
10     Teacher(string name1, string surname1, int age1,int
11         experience1, string title1);
12     Teacher();
13     void setTitle(string title1);

```



```
14 int getExperience();
15 bool is_PhD();
16 void showInfoTeacher();
17 };
18 #endif // TEACHER_H_INCLUDED
```

Listing 1.3. Plik nagłówkowy Teacher.h

```
1 #include<iostream>
2 #include "Teacher.h"
3 Teacher::Teacher(string name1, string surname1,
4                   int age1,int experience1, string title1):
5                   Person(name1,surname1,age1)
6 {
7     experience=experience1;
8     title=title1;
9     cout<<"Konstruktor klasy pochodnej Teacher"<<endl;
10 }
11 Teacher::Teacher()
12 {
13     cout<<"Konstruktor bez. klasy pochodnej Teacher"<<endl;
14 }
15 void Teacher::setTitle(string title1)
16 {
17     title=title1;
18 }
19 int Teacher::getExperience()
20 {
21     return experience;
22 }
23 bool Teacher::is_PhD()
24 {
25     if(title=="PhD")
26         return true;
27     else
28         return false;
29 }
30 void Teacher::showInfoTeacher()
31 {
32     showInfoPerson();
33     cout<<"Staz pracy: "<<experience<<" tytul naukowy: "<<
34         title<<endl;
35 }
```

Listing 1.4. Plik źródłowy Teacher.cpp



Na listingu 1.5. przedstawiony został plik *main.cpp*, w którym przetestowane zostały klasy: *Person* oraz *Teacher*. Listing 1.5. zawiera:

- linijki 7 – 10 przetestowanie klasy *Person*;
- linijki 12 -16 – przetestowanie klasy *Teacher*.

Obiekty z klasy *Teacher* mają dostęp do metod z klasy *Person*. W linijce 16 dla obiektu z klasy *Teacher* wywoływana jest metoda *setAge*, która została zdefiniowana w klasie *Person*. Klasa *Teacher* rozszerza klasę *Person*, między nimi zachodzi następująca relacja: każdy nauczyciel jest osobą, ale nie każda osoba to nauczyciel. Z związku z tym, obiekt klasy *Teacher* może korzystać z metod klasy *Person*.

```
1  #include <iostream>
2  #include "Person.h"
3  #include "Teacher.h"
4  using namespace std;
5  int main()
6  {
7      cout<<endl<<"Obiekty klasy Person"<<endl;
8      Person o1("Grazyna", "Nowak", 45);
9      o1.showInfoPerson();
10     cout<<"Czy pełnoletnia?: "<<o1.is_18()<<endl;
11     o1.setAge(16);

12     cout<<endl<<"Obiekty klasy Teacher"<<endl;
13     Teacher n1("Barbara", "Kowalska", 56, 30, "Phd");
14     n1.showInfoTeacher();
15     cout<<"Czy ma Phd?: "<<n1.is_Ph()<<endl;
16     n1.setAge(34);
17     return 0;
18 }
```

Listing 1.5. Plik *main.cpp*

Na rysunku 1.18. przedstawiony został wynik działania programu z listingu 1.5. Przy tworzeniu obiektu z klasy *Person* wywołany został konstruktor z klasy *Person*, natomiast przy tworzeniu obiektu z klasy *Teacher* wywoływane są dwa konstruktory: z klasy *Person* oraz z klasy *Teacher*, ponieważ klasa *Teacher* dziedziczy po klasie *Person*.

```
Obiekty klasy Person
Konstruktor klasy bazowej - Person
Imie: Grazyna nazwisko: Nowak wiek: 45
Czy pełnoletnia?: 1

Obiekty klasy Teacher
Konstruktor klasy bazowej - Person
Konstruktor klasy pochodnej Teacher
Imie: Barbara nazwisko: Kowalska wiek: 56
Staz pracy: 30 tytuł naukowy: Phd
Czy ma Phd?: 1
```

Rys. 1.18. Wynik działania programu z listingu 1.5.

Powtórzenie wiadomości o tablicach obiektów:

Wyróżniamy cztery rodzaje tablic jednowymiarowych przechowujących obiekty:

- statyczna tablica statycznych obiektów,

- dynamiczna tablica statycznych obiektów,
- statyczna tablica dynamicznych obiektów,
- dynamiczna tablica dynamicznych obiektów.

Dynamiczna tablica tworzona jest w oparciu o operator *new*, natomiast w statycznej tablicy określana jest długość tablicy w momencie jej deklaracji. Jeśli element w tablicy jest obiektem dynamicznym, czyli utworzonym za pomocą operatora *new*, odwołanie się do metody lub pola w klasie następuje poprzez użycie operatora „->” (strzałka). Natomiast, jeśli obiekt jest obiektem statycznym, to odwołanie następuje przez operator „.” (kropka). Obiekty statyczne tworzone są przy deklaracji tablicy, a więc dla nich wywoływany jest niejawnie konstruktor bezargumentowy. W celu zainicjalizowania takich obiektów danymi należy użyć metody inicjalizującej. Dla obiektów dynamicznych występuje jawne wywołanie konstruktora.

Do klasy *Person* została dodana metoda *init* umożliwiająca inicjalizację pól klasy, kiedy obiekt już istnieje. Listing 1.6. pokazuje, co należy dodać do pliku nagłówkowego *Person.h* a listing 1.7. pokazuje, co należy dodać do pliku źródłowego *Person.cpp*.

```
void init(string name1, string surname1, int age1);
```

Listing 1.6. Dodanie metody *init* do pliku nagłówkowego *Person.h*

```
1 void Person::init(string name1, string surname1, int age1)
2 {
3     name=name1;
4     surname=surname1;
5     age=age1;
6 }
```

Listing 1.7. Dodanie metody *init* do pliku źródłowego *Person.cpp*

Na listingu 1.8. przedstawione zostały cztery rodzaje tablic omówione powyżej. Kod przedstawiony na tym listingu został dodany do pliku *main.cpp*. Listing 1.8. zawiera:

- linijki 1 - 4 – deklaracja tablic;
- linijki 5 – 9 – statyczna tablica statycznych obiektów. W momencie deklaracji tablicy został wywołany konstrukt bezargumentowy, więc dla tych obiektów wywoływana jest metoda *init*;
- linijki 10 – 15 – dynamiczna tablica statycznych obiektów. Długość tablicy określona za pomocą operatora *new*. W momencie przydzielenia pamięci wywołał się konstruktor bezargumentowy, więc dla tych obiektów wywoływana została metoda *init*;
- linijki 16 – 20 – statyczna tablica dynamicznych obiektów. Obiekt tworzony jest za pomocą operatora *new* i w tym momencie wywoływany jest konstruktor parametrowy;
- linijki 21 - 26 – dynamiczna tablica dynamicznych obiektów. Długość tablicy jest określona za pomocą operatora *new*. Obiekt tworzony jest za pomocą operatora *new* i w tym momencie wywoływany jest konstruktor parametrowy;
- linijki 27 – 29 – zwolnienie pamięci przydzielonej dla każdego obiektu;
- linijki 30 – 33 – zwolnienie pamięci przydzielonej dla każdego obiektu a następnie dla tablicy.



```
1  Person p1[3];
2  Person* p2;
3  Person* p3[3];
4  Person** p4;

5  for(int i=0; i<3; i++)
6  {
7      p1[i].init("Anna", "Nowak", 20+i);
8      p1[i].showInfoPerson();
9  }

10 p2=new Person[3];
11 for(int i=0; i<3; i++)
12 {
13     p2[i].init("Ola", "Adamek", 20+i);
14     p2[i].showInfoPerson();
15 }

16 for(int i=0; i<3; i++)
17 {
18     p3[i]=new Person("Kasia", "Kowalska", 20+i);
19     p3[i]->showInfoPerson();
20 }

21 p4=new Person*[3];
22 for(int i=0; i<3; i++)
23 {
24     p4[i]=new Person("Pawel", "Wojcik", 20+i);
25     p4[i]->showInfoPerson();
26 }

27 for(int i=0; i<3; i++){
28     delete p3[i];
29 }

30 for(int i=0; i<3; i++){
31     delete p4[i];
32 }
33 delete [] p4;
```

Listing 1.8. Stworzenie tablic obiektów – dodanie kodu do pliku main.cpp

Rysunek 1.19. przedstawia wynik działania kodu z listingu 1.8. Każda czerwona ramka dotyczy innego rodzaju tablicy, zgodnie z listingiem 1.8. Jak można zauważyć, dla tablic (*p1*,*p2*) przechowujących obiekty statyczne razem wywołują się wszystkie konstruktory. Z kolei dla tablic (*p3*,*p4*), które przechowują obiekty dynamiczne konstruktory wywołane są kolejno.

```

konstruktor bez. klasy bazowej - Person
konstruktor bez. klasy bazowej - Person
konstruktor bez. klasy bazowej - Person
Imie: Anna nazwisko: Nowak wiek: 20
Imie: Anna nazwisko: Nowak wiek: 21
Imie: Anna nazwisko: Nowak wiek: 22
konstruktor bez. klasy bazowej - Person
konstruktor bez. klasy bazowej - Person
konstruktor bez. klasy bazowej - Person
Imie: Ola nazwisko: Adamek wiek: 20
Imie: Ola nazwisko: Adamek wiek: 21
Imie: Ola nazwisko: Adamek wiek: 22
konstruktor klasy bazowej - Person
Imie: Kasia nazwisko: Kowalska wiek: 20
konstruktor klasy bazowej - Person
Imie: Kasia nazwisko: Kowalska wiek: 21
konstruktor klasy bazowej - Person
Imie: Kasia nazwisko: Kowalska wiek: 22
konstruktor klasy bazowej - Person
Imie: Pawel nazwisko: Wojcik wiek: 20
konstruktor klasy bazowej - Person
Imie: Pawel nazwisko: Wojcik wiek: 21
konstruktor klasy bazowej - Person
Imie: Pawel nazwisko: Wojcik wiek: 22
    
```

p1
p2
p3
p4

Rys. 1.19. Wynik działania programu z listingu 1.8.

Polimorfizm:

Polimorfizm związany jest z występowaniem kilku metod o tej samej nazwie w różnych wersjach. Wyróżniane są dwa rodzaje polimorfizmu: dynamiczny i statyczny. Polimorfizm dynamiczny związany jest z metodami wirtualnymi i polega na tym, że daje możliwość nadpisywania/przesłaniania (*ang. override*) metod z klasy rodzica, w klasie dziecka. Natomiast polimorfizm statyczny oznacza przeciążanie metod (*ang. overload*), czyli tworzenie różnych metod o tej samej nazwie ale z różnym typem lub różną liczbą argumentów. W polimorfizmie statycznym odpowiednia wersja metody wybierana jest na etapie kompilacji (Compiletime), a w polimorfizmie dynamicznym na podstawie wskaźnika lub referencji do obiektu na etapie wykonania programu (Runtime). Polimorfizm dynamiczny będzie tematem laboratorium 2, a polimorfizm statyczny będzie omówiony podczas laboratorium 3.

Zadania do wykonania:

Zadanie 1.1. Przykładowy kod

Uruchom kod przedstawiony i umówiony podczas tego laboratorium. Następnie dodaj klasę *Student* dziedziczącą po klasie *Person* zawierającą:

- atrybut prywatny: *index* – typu *string*;
- konstruktor inicjalizujący składowe klasy *Student* korzystający z parametrycznego konstruktora klasy bazowej. W konstruktorze powinna zostać wyświetlana informacja identyfikująca, z jakiej klasy pochodzi konstruktor;
- metodę *setIndex(string newIndex)* – na podstawie przekazanego parametru ustawia wartość pola *index*;
- metodę *getIndex()* – zwracającą wartość pola *index*;
- metodę *showInfoStudent()* – wyświetlającą informacje o studencie.

W pliku *main.cpp* należy stworzyć:

- statyczną tablicę statycznych obiektów klasy *Student*,
- dynamiczną tablicę statycznych obiektów klasy *Student*,
- statyczną tablicę dynamicznych obiektów klasy *Student*,
- dynamiczną tablicę dynamicznych obiektów klasy *Student*.

Popowiedź: do inicjalizacji danymi po utworzeniu obiektu można użyć metody *init* z klasy *Person* a następnie metody *setIndex* z klasy *Student*.



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Zadanie 1.2. Zwierzęta

Stwórz klasę *Animal*, zawierającą następujące elementy:

- atrybuty prywatne:
 - *limbNr* – liczba kończyn,
 - *name* – nazwa zwierzęcia,
 - *protectedAnimal* – typ logiczny, czy zwierzę jest chronione;
- konstruktor bezparametrowy;
- konstruktor z parametrami ustawiający wszystkie składowe klasy, przy czym parametr dla atrybutu *protected* powinien być parametrem o wartości domyślnej *true*;
- dla pól dodaj setery umożliwiające ustawienie wartości pola wartością przekazaną w argumencie oraz gettery zwracające wartości pól. Metody należy nazwać według następującego schematu: *getNazwa* lub *setNazwa*;
- metodę *giveVoice()* – wyświetla na konsolę ciąg znaków „Chrum, miau, hau, piii”;
- metodę *info()* – wyświetla dane składowe na konsolę.

Stwórz klasę *Dog*, która będzie dziedziczyła publicznie po klasie *Animal*, zawierającą następujące elementy:

- atrybuty prywatne:
 - *breed* – rasa psa,
 - *levelOfGuideSkills* – poziom umiejętności przewodnika, liczba całkowita, z zakresu <1,10>,
 - *levelOfTrackerSkills* – poziom umiejętności tropiciela, liczba całkowita, z zakresu <1,10>;
- konstruktor inicjalizujący wszystkie składowe klasy *Dog*;
- konstruktor inicjalizujący tylko składowe klasy *Animal*;
- konstruktor bezparametrowy;
- metodę *setSkillLevel(int type, int value)* – ustawia poziom wybranej umiejętności na podstawie podanego typu umiejętności oraz jego wartości. Pierwszy parametr określa, która umiejętność jest ustawiana, drugi – jaka jest wartość umiejętności;
- metodę *getSkillLevel(int type)* – zwraca poziom umiejętności, którego rodzaj został podany jako argument. Jeśli poziom nie jest ustawiony, metoda powinna zwrócić 0;
- metodę *giveVoice()* przesłaniając metodę z klasy bazowej, aby metoda wyświetlała „Hau, hau”;
- metodę *info()* przesłaniając metodę z klasy bazowej, aby wyświetlała pełną informację o obiekcie klasy *Dog*.

Stwórz klasę *Cat* dziedziczącą po klasie *Animal*, zawierającą następujące elementy:

- atrybut prywatny:
 - *levelOfMouseHunting* – poziom umiejętności łowienia myszy, liczba całkowita z zakresu <1,10>,
 - *mice[5]* – tablica przechowująca liczbę upolowanych myszy w ciągu 5 lat z podziałem na lata;
- konstruktor inicjalizujący tylko składowe klasy *Animal*;
- konstruktor bezparametrowy;
- metodę *initMice*, umożliwiającą zainicjalizowanie wartościami tablicy *mice*;

- metodę *initCat*, umożliwiającą inicjalizację wszystkich składowych klasy *Cat*.
- metodę *setLevelOfMouseHunting(int value)* – ustawiającą poziom umiejętności łowienia myszy;
- metodę *getLevelOfMouseHunting()* – zwracającą poziom umiejętności łowienia myszy. Jeśli poziom jest nieustawiony należy zwrócić 0.
- metodę *getMice(int index)* – zwracającą liczbę upolowanych myszy w danym roku. *Index* jest wartością całkowitą z przedziału <1,5>;
- metodę *giveVoice()* przesłaniając metodę z klasy bazowej, aby metoda wyświetlała „Miau miau”;
- metodę *info()* przesłaniając metodę z klasy bazowej, aby wyświetlała pełną informację o obiekcie klasy *Cat*.

Zaimplementuj następujące funkcje:

- *howManyProtectedAnimals*, przyjmującą jako argument tablicę obiektów klasy *Animal* oraz jej rozmiar. Funkcja ma za zadanie policzyć ile zwierząt jest chronionych i zwrócić wynik;
- *howManyTrackerDogs*, przyjmującą jako argument tablicę obiektów klasy *Dog* oraz jej rozmiar. Funkcja ma za zadanie wyświetlić wszystkie psy których poziom tropiciela jest większy niż przewodnika.
- *howManyCats*, przyjmującą jako argument tablicę obiektów klasy *Cat* oraz jej rozmiar. Funkcja ma za zadanie policzyć dla każdego kota liczbę upolowanych mysz w ciągu 5 lat i wyświetlić na konsoli.

W funkcji *main* należy przetestować działanie powyższych klas oraz funkcji.

LABORATORIUM 2. WIELOKROTNE WYKORZYSTANIE KODU. POLIMORFIZM DYNAMICZNY (METODY WIRTUALNE).

Cel laboratorium:

Przypomnienie wiadomości o polimorfizmie dynamicznym.

Zakres tematyczny zajęć:

- metody wirtualne,
- polimorfizm dynamiczny.

Pytania kontrolne:

1. Co to jest polimorfizm?
2. Co to jest metoda wirtualna?
3. Jaka jest różnica między metodą wirtualną a metodą niewirtualną?
4. Co to znaczy, że klasa jest abstrakcyjna?
5. Jaka jest zaleta stosowania polimorfizmu?

Polimorfizm dynamiczny:

Polimorfizm umożliwia implementację różnego zachowania tych samych metod w czasie wykonywania programu. Oznacza to, że obiekt może zachowywać się różnie w zależności od kontekstu wywołania. Można powiedzieć, że polimorfizm to jeden interfejs, który zawiera wiele realizacji. Polimorfizm daje możliwość decydowania jaka metoda zostanie wywołana pod daną nazwą nie w momencie kompilacji, ale w momencie wywołania. Bezpośrednio związany jest z metodami wirtualnymi i klasą abstrakcyjną. Listing 2.1. przedstawia nagłówek funkcji wirtualnej. Klasę można nazwać abstrakcyjną jeśli zawiera przynajmniej jedną czysto wirtualną metodę. Metoda czysto wirtualna to metoda, która występuje w klasie bazowej i zawiera tylko deklarację, bez definicji. Definicja tej metody znajduje się w klasach potomnych. W definicji takiej metody, w klasie bazowej, zawsze na końcu występuje `=0`. Listing 2.2. prezentuje nagłówek metody czysto wirtualnej. Jeśli klasa jest abstrakcyjna, jej obiekt nie może istnieć.

```
virtual type functionName(arguments)
```

Listing 2.1. Deklaracja metody wirtualnej w klasie bazowej

```
virtual type functionName(arguments)=0
```

Listing 2.2. Deklaracja metody czysto wirtualnej w klasie bazowej

Listing 2.3. przedstawia nagłówek metody wirtualnej w klasie potomnej. Dodawanie słowa *virtual* nie jest konieczne ale zwiększa czytelność kodu. Podobnie jest ze słowem *override*.

```
virtual type functionName(arguments) override;
```

Listing 2.3. Deklaracja metody wirtualnej w klasie potomnej



Sposób wywołania metody wirtualnej zależy od typu dynamicznego obiektu, a nie od typu statycznego. Typem statycznym obiektu wskazywanego przez wskaźnik jest typ tego wskaźnika, a typem dynamicznym obiektu wskazywanego przez wskaźnik jest typ na jaki dany wskaźnik wskazuje. Typ dynamiczny obiektu może być zazwyczaj określony dopiero podczas wykonania programu, a typ statyczny jest znany już w czasie kompilacji. Listing 2.4. prezentuje różnice między typem statycznym a dynamicznym. Listing 2.4. zawiera:

- linijka 3 – stworzenie metody wirtualnej w klasie bazowej;
- linijka 9 – stworzenie metody wirtualnej w klasie potomnej;
- linijka 16 – stworzenie wskaźnika *p* typu *Parent** do obiektu klasy *Child*, typ statyczny to *Parent*, typ dynamiczny to *Child*;
- linijka 18 – wywołanie *show()* dla typu dynamicznego czyli dla typu *Child*;
- linijka 20 – stworzenie wskaźnika *c* typu *Child** do obiektu klasy *Child*, typ statyczny i dynamiczny to *Child*;
- linijka 22 – wywołanie *show()* dla typu dynamicznego czyli dla typu *Child*.

```
1  class Parent
2  { attributes, methods, virtual destructor
3
4      virtual void show() {
5          cout<<"Metoda wirtualna Parent"<<endl;
6      }
7  };
8  class Child: public Parent
9  { attributes, methods
10
11      virtual void show()override{
12          cout<<"Metoda wirtualna Child"<<endl;
13      }
14  };
15
16  int main()
17  {
18      //typ statyczny Parent, typ dynamiczny Child
19      Parent* p=new Child();
20      //metoda show z klasy Child
21      p->show();
22      //typ statyczny Child, typ dynamiczny Child
23      Child* c=new Child();
24      //metoda show z klasy Child
25      c->show();
26  }
```

Listing 2.4. Przykład typu statycznego oraz dynamicznego

Stosując mechanizm dziedziczenia w kontekście polimorfizmu, należy pamiętać, aby w klasie bazowej umieścić wirtualny destruktor. Wirtualny destruktor zapewnia wywołanie wszystkich destruktorów znajdujących się w klasach potomnych. Zadaniem destruktorów jest zwalnianie zasobów. Niepoprawne użycie destruktorów może doprowadzić do wycieków pamięci.

Dzięki polimorfizmowi nie trzeba posiadać wiedzy o tym jakie są typy klas potomnych, ponieważ, dysponując obiektem klasy bazowej, na etapie wykonywania kodu znany jest typ dynamiczny a więc wiadomo, z której klasy metoda powinna zostać wywołana. Jako przykłady zastosowania polimorfizmu można podać:

- klasa bazowa: Pojazd z metodą wirtualną wyświetlającą maksymalną prędkość oraz klasy potomne: Samochód osobowy, Autobus, Traktor. Każda z tych klas ma również nadpisaną metodę wirtualną wyświetlającą maksymalną prędkość dla samochodu osobowego, autobusu czy traktora. Każda z klas potomnych jest pojazdem ale ma inną określoną maksymalną prędkość. W związku z tym w każdej klasie „po swoim” będzie nadpisana metoda;
- klasa bazowa Człowiek z metodą wirtualną wyświetlającą obowiązki oraz klasy potomne: Niemowlak, Student, Emeryt. Każda z tych osób ma inne obowiązki, na przykład: niemowlak śpi, student uczy się, emeryt ogląda TV. Każda z klas potomnych jest człowiekiem ale w inny sposób wypełnia swoje obowiązki.

Powyższe przykłady (klasa Pojazd i Człowiek), można zinterpretować na dwa sposoby:

- przy użyciu metod wirtualnych. Zakładamy, że obiekty klas bazowych mogą istnieć. Oznacza to, że można stworzyć obiekt klasy Pojazd oraz klasy Człowiek nie wchodząc w szczegóły jaki rodzaj pojazdu lub jaka grupa społeczna;
- przy użyciu metod czysto wirtualnych. Zakładamy, że obiekty klas bazowych nie mogą istnieć. Oznacza to, że klasa Pojazd i Człowiek są klasami abstrakcyjnymi.

Listingi 2.5. – 2.16. prezentują przykład polimorfizmu. Zostały stworzone trzy klasy. Klasa bazowa to klasa *Figure*, reprezentująca figurę geometryczną oraz dwie klasy dziedziczące po klasie *Figure*: *Square*, reprezentującą kwadrat oraz klasę *Circle*, reprezentującą koło. Dla każdej figury można policzyć jej pole. Pole kwadratu liczy się używając innego wzoru niż pole koła. Na listingu 2.5. przedstawiony został plik nagłówkowy *Figure.h*. Klasa *Figure* jest klasą abstrakcyjną, ponieważ zawiera metodę czysto wirtualną. Listing 2.5. zawiera:

- linijka 9 – wirtualny destruktor;
- linijka 12 – definicję metody czysto wirtualnej.

```

1  #ifndef FIGURE_H_INCLUDED
2  #define FIGURE_H_INCLUDED
3  class Figure
4  {
5      private:
6          float area;
7      public:
8          Figure();
9          virtual ~Figure();
10         float getArea();
11         void setArea(float area1);
12         virtual void calculateArea()=0;
13         void show();
14     };
15 #endif // FIGURE_H_INCLUDED
    
```

Listing 2.5. Plik nagłówkowy klasy *Figure*

Plik źródłowy *Figure.cpp* został przedstawiony na listingu 2.6. W pliku nie ma definicji metody wirtualnej *calculateArea*. Jej definicja znajdzie się w klasach potomnych.



```
1  #include <iostream>
2  #include "Figure.h"
3  using namespace std;
4  Figure::Figure()
5  {
6      cout<<"Konstruktor klasy bazowej Figure"<<endl;
7  }
8  Figure::~~Figure()
9  {
10     cout<<"Wirtualny destruktory klasy bazowej Figure"<<endl;
11 }

12 float Figure::getArea()
13 {
14     return area;
15 }

16 void Figure::setArea(float area1)
17 {
18     area=area1;
19 }

20 void Figure::show()
21 {
22     cout<<"Pole: "<<area<<endl;
23 }
```

Listing 2.6. Plik źródłowy klasy Figure

Na listingu 2.7. przedstawiona została zawartość pliku nagłówkowego dla klasy *Square*. Klasa *Square* dziedziczy po klasie *Figure*. W linijce 11 znajduje się definicja metody wirtualnej, która w tej klasie zostanie zaimplementowana. W przypadku klas potomnych nie ma konieczności używania słowa *virtual*, natomiast zaleca się jednak to robić, ponieważ zwiększa to czytelność kodu.

```
1  #ifndef SQUARE_H_INCLUDED
2  #define SQUARE_H_INCLUDED
3  #include "Figure.h"
4  class Square: public Figure
5  {
6      private:
7          float a;
8      public:
9          Square(float a1);
10         ~Square();
11         virtual void calculateArea() override;
12 };
13 #endif // SQUARE_H_INCLUDED
```

Listing 2.7. Plik nagłówkowy klasy Square



Listing 2.8. zawiera plik źródłowy klasy *Square*. Linijki 14 – 18 przedstawiają definicje metody *calculateArea*.

```
1  #include<iostream>
2  #include "Figure.h"
3  #include "Square.h"
4  using namespace std;
5  Square:: Square(float a1):Figure()
6  {
7      a=a1;
8      cout<<"Konstruktor klasy Square"<<endl;
9  }

10 Square:: ~Square()
11 {
12     cout<<"Destruktor klasy Square"<<endl;
13 }

14 void Square::calculateArea()
15 {
16     float p=a*a;
17     setArea(p) ;
18 }
```

Listing 2.8. Plik źródłowy klasy Square

Na listingach 2.9. oraz 2.10. przedstawione zostały pliki nagłówkowe oraz źródłowe dla klasy *Circle*.

```
1  #ifndef CIRCLE_H_INCLUDED
2  #define CIRCLE_H_INCLUDED

3  class Circle: public Figure
4  {
5      private:
6          float r;
7      public:
8          Circle(float r1);
9          ~Circle();
10         virtual void calculateArea() override;
11 };
12 #endif // CIRCLE_H_INCLUDED
```

Listing 2.9. Plik nagłówkowy klasy Circle

```
1  #include<iostream>
2  #include "Figure.h"
3  #include "Circle.h"
4  using namespace std;
```



```

5  Circle::Circle(float r1)
6  {
7      r=r1;
8      cout<<"Konstruktor w klasie Circle"<<endl;
9  }

10 Circle::~~Circle()
11 {
12     cout<<"Destruktor w klasie Circle"<<endl;
13 }

14 void Circle:: calculateArea()
15 {
16     float p=3.14*r*r;
17     setArea(p) ;
18 }

```

Listing 2.10. Plik źródłowy klasy Circle

Listing 2.11. przedstawia plik *main.cpp*, który prezentuje użycie polimorfizmu. Listing 2.12. zawiera:

- linijka 8 – stworzenie wskaźnika *f1* o typie statycznym *Figure* i typie dynamicznym *Square*. Został wywołany konstruktor klasy *Square*;
- linijka 9 – stworzenie wskaźnika *f2* o typie statycznym *Figure* i typie dynamicznym *Circle*. Został wywołany konstruktor klasy *Circle*;
- linijka 10 – polimorficzne wywołanie metody *calculateArea*, odpowiedniej dla typu dynamicznego;
- linijka 11 – wywołanie metody *show* z klasy *Figure* na obiekcie *f1*, ponieważ metoda ta nie była wirtualna;
- linijki 14 – 15 – zwolnienie pamięci – wywołanie destruktorów.

W liniijkach 10 i 12, kiedy wywoływana jest metoda *calculateArea*, wtedy sprawdzany jest typ dynamiczny i na jego podstawie wywoływana jest odpowiednia metoda.

```

1  #include <iostream>
2  #include "Figure.h"
3  #include "Square.h"
4  #include "Circle.h"
5  using namespace std;
6  int main()
7  {
8      Figure* f1=new Square(4);
9      Figure* f2=new Circle(2);

10     f1->calculateArea();
11     f1->show();

12     f2->calculateArea();
13     f2->show();

```



```
14 delete f1;
15 delete f2;
16 return 0;
17 }
```

Listing 2.11. Plik *main.cpp* w wersji ze wskaźnikiem

Na listingu 2.12. przedstawiony został polimorfizm z wersji z przekazaniem referencji. Listing 2.12. zawiera:

- linijka 1 – stworzenie wskaźnika typu *Figure**;
- linijka 2 – stworzenie obiektu typu *Square*;
- linijka 3 – obiekt *f3* odnosi się do referencji obiektu *s1*;
- linijka 4 – polimorficzne wywołanie metody *calculateArea*;
- linijka 5 – wywołanie *show*.

```
1 Figure *f3;
2 Square s1(4);
3 f3=&s1;
4 f3->calculateArea();
5 f3->show();
```

Listing 2.12. Funkcja *main* w wersji z referencją

Rysunek 2.1. przedstawia wykonanie kodu z listingu 2.11. Przy tworzeniu obiektu *f1* wywoływany jest konstruktor klasy *Figure* oraz konstruktor klasy *Square*. Podobna sytuacja dotyczy tworzenia obiektu *f2*. Następnie wyświetlone są pola obydwu figur. Na końcu programu wywoływane są destruktory. Wywoływany jest destruktory klasy potomnej: *Square* lub *Circle* a następnie wirtualny destruktory klasy bazowej – klasy *Figure*. Jeśli w klasie bazowej wirtualny destruktory zastąpić zwykłym destruktorem, to wtedy nie wywołają się destruktory klas potomnych a tylko klasy bazowej.

```
Konstruktor klasy bazowej Figure
Konstruktor klasy Square
Konstruktor klasy bazowej Figure
Konstruktor w klasie Circle
Pole: 16
Pole: 12.56
Destruktor klasy Square
Wirtualny destruktory klasy bazowej Figure
Destruktor w klasie Circle
Wirtualny destruktory klasy bazowej Figure
```

Rys. 2.1. Wynik działania kodu z listingu 2.11.

W klasie *Figure* zdefiniowana została metoda *show*, która wyświetla pole figury. Metoda ta na następnych listingach zostanie zmodyfikowana i dodana do klasy potomnej *Square*, aby jeszcze lepiej pokazać ideę metod wirtualnych. Listing 2.13. przedstawia modyfikację metody *show* w pliku *Figure.h*. W pliku *Figure.cpp* nic się nie zmienia.

```
virtual void show();
```

Listing 2.13. Modyfikacja metody *show* w pliku *Figure.h*.



Do klasy *Square* została dodana metoda *show*. Listing 2.14 przedstawia, co zostało dodane do pliku nagłówkowego *Square.h*.

```
virtual void show() override;
```

Listing 2.14. Dodanie metody *show* do klasy *Square* – plik nagłówkowy

Listing 2.15. przedstawia definicję metody *show* w pliku źródłowym *Square.cpp*. Dla rozróżnienia na późniejszym etapie, metoda wyświetla stosowny komunikat.

```
1 void Square::show()
2 {
3     cout<<"Show w klasie Sqaure, pole: "<<getArea()<<endl;
4 }
```

Listing 2.15. Definicja metody *show* w klasie *Square* – plik źródłowy

W klasie *Circle* oraz funkcji *main* nie zostały wprowadzone żadne zmiany. Jak można zauważyć na rysunku 2.2., który przedstawia fragment kodu z listingu 2.11., dla obiektu *f1*, posiadającego typu dynamiczny *Sqaure* została wywołana metoda *show*, która została zdefiniowana w klasie *Square*. Natomiast dla obiektu *f2*, posiadającego typ dynamiczny *Circle*, została wywołana metoda *show*, z klasy *Figure*, ponieważ w klasie *Circle* nie ma zdefiniowanej metody *show*.

```
Show w klasie Sqaure, pole: 16
Pole: 12.56
```

Rys. 2.2. Wynik działania funkcji *show* z listingu 2.11.

Teraz usuńmy słowo *virtual* z deklaracji metody *show* w klasie *Figure* oraz słowa: *virtual* i *override* z klasy *Square*. Rysunek 2.3. przedstawia wynik fragmentu kodu z listingu 2.11. Jak można zauważyć, pomimo, że w klasie *Square* jest zdefiniowana metoda *show*, nie jest ona wywoływana, a zamiast tego jest wywoływana metoda *show* z klasy *Figure*, ponieważ metoda *show* nie została zdefiniowana jako metoda wirtualna.

```
Pole: 16
Pole: 12.56
```

Rys. 2.3. Wynik działania funkcji *show* z listingu 2.11.

Na listingu 2.16. przedstawiony został kod pokazujący polimorficzne działanie programu na przykładzie tablicy. Kod został dodany do funkcji *main*. Stworzona została tablica o typie statycznym *Figure*. Dwa pierwsze elementy tablicy zawierają typ dynamiczny *Square*, a ostatni element *Circle*. Następnie na każdym elemencie tablicy wywoływana jest metoda *calculateArea()*, w momencie wywoływania tej metody znany jest typ dynamiczny i dla niego wywoływana jest metoda z odpowiedniej klasy.

```
1 Figure* tab[3];
2 tab[0]=new Square(4);
3 tab[1]=new Square(2);
4 tab[2]=new Circle(5);
```

```
5   for(int i=0;i<3;i++)
6   {
7       tab[i]->calculateArea();
8       delete tab[i];
9   }
```

Listing 2.16. Zaprezentowanie polimorfizmu na przykładzie tablicy

Zadania do wykonania:

Zadanie 2.1. Przykładowy kod

Uruchom kod przedstawiony i omówiony podczas tego laboratorium. Następnie dodaj możliwość policzenia obwodu dla kwadratu oraz koła.

Zadanie 2.2. Firma informatyczna

Stwórz klasę *Employee*, która będzie reprezentować pracownika firmy informatycznej i posiadać następujące elementy:

- atrybuty prywatne:
 - *surname* jako nazwisko pracownika,
 - *age* jako wiek pracownika,
 - *experience* jako staż pracy wyrażony w latach,
 - *salary* jako miesięczna wypłata.
- konstruktor argumentowy pozwalający na zainicjalizowanie pól klasy;
- konstruktor bezargumentowy;
- wirtualny destruktor;
- wirtualną metodę *void show()*, która będzie wyświetlać informacje o pracowniku;
- czysto wirtualną metodę *int calculateSalary(int value)*;
- metodę *int ageEmployment()*, obliczającą wiek w momencie zatrudnienia;
- odpowiednie gettery do pól.

Stwórz klasę *Developer*, która dziedziczy po klasie *Employee* oraz implementuje metodę *float calculateBonus(int value)*, obliczając wypłatę dla developera, według następującego wzoru: $value + 0.2 * value * (salary + experience)$.

Stwórz klasę *TeamLeader*, która dziedziczy po klasie *Employee* oraz implementuje metodę *float calculateBonus(int value)*, obliczając wypłatę dla team leader'a, według następującego wzoru: $value * (1 + salary + experience)$. Dodatkowo powinna zostać nadpisana metoda *show()*, która ma za zadanie wyświetlać informacje dotyczące team leader'a oraz tekst „Jestem Team Leader z X letnim doświadczeniem”, gdzie w miejsce X należy wstawić liczbę lat doświadczenia.

Należy pamiętać aby do klasy *Developer* oraz *TeamLeader* dodać odpowiednie konstruktory.

Stwórz następujące funkcje:

- *whoWorkMoreThan5Years*, która przyjmie jako argument tablicę typu *Employee*** oraz jej rozmiar i wyświetli pracowników, którzy pracują więcej niż 5 lat.



- *howManyEarnLessThanMeanBonus*, która przyjmie jako argument tablicę typu *Employee*** oraz jej rozmiar i policzy ilu pracowników dostało premię niższą niż średnia wszystkich premii.

W funkcji *main* należy stworzyć tablicę i zaprezentować polimorficzne działanie programu oraz przetestuj funkcje.

Zadanie 2.3. Bufor

Stwórz klasę *Bufor*, która posiada następujące elementy:

- atrybuty prywatne:
 - wskaźnik, który będzie reprezentować dynamiczną tablicę przechowującą liczby całkowite,
 - rozmiar tablicy,
 - indeks pierwszego wolnego miejsca w tablicy. Zakładamy, że elementy do tablicy są dodawane po kolei.
- bezargumentowy konstruktor, w którym zostanie stworzona tablica o rozmiarze 10, a indeks pierwszej wolnej komórki w tablicy zostanie ustawiony na 0;
- konstruktor argumentowy, który jako argument dostanie rozmiar tablicy. W konstruktorze powinna zostać stworzona tablica oraz wartość pierwszego wolnego miejsca w tablicy ustawiona na 0;
- wirtualny destruktor, w którym zostanie usunięta przydzielona pamięć;
- wirtualną metodę *void add(int value)*, która umożliwi dodanie liczby w pierwsze wolne miejsce tablicy. Zakładamy, że będzie wolne miejsce w tablicy;
- czysto wirtualną metodę *double calculate()*;
- metodę *int getIndex()*, która będzie zwracała indeks pierwszego wolnego miejsca w tablicy;
- metodę *int getSize()*, która będzie zwracała rozmiar tablicy;
- metodę *int getTab(int i)*, która będzie zwracała *i*-ty element w tablicy;
- metodę *int getFirst()*, która zwraca pierwszy dostępny indeks;
- metodę *void setFirst(value)*, która ustawia wartość pola przechowującego pierwszy wolny indeks w tablicy;
- metodę *void setTab(int pos, int value)*, która będzie ustawiała wartość znajdującą się pod indeksem *pos* na wartość *value*;
- metodę wyświetlającą zawartość tablicy;

Stwórz klasę *MeanBufor*, która dziedziczy po klasie *Bufor* oraz nadpisuje metodę *calculate()*, zwracając średnią arytmetyczną z liczb, które zostały wstawione do tablicy.

Stwórz klasę *MaxBufor*, która dziedziczy po klasie *Bufor* oraz nadpisuje metodę *calculate()*, zwracając maksymalną liczbę z pośród liczb wstawionych do tablicy. Ponadto klasa ta nadpisuje metodę *void add(int value)*. Ma ona działać tak samo z tą różnicą, że ma sprawdzać czy w tablicy jest jeszcze miejsce i, jeżeli nie ma już miejsca, to ma nic nie wstawiać do tablicy oraz wyświetlić odpowiedni komunikat na konsolę.

Należy pamiętać aby do klasy *MeanBufor* oraz *MaxBufor* dodać odpowiednie konstruktory

W funkcji *main* należy stworzyć tablicę i zaprezentować polimorficzne działanie programu.



LABORATORIUM 3. WIELOKROTNE WYKORZYSTANIE KODU. POLIMORFIZM STATYCZNY (SZABLONY FUNKCJI I KLAS).

Cel laboratorium:

Omówienie polimorfizmu statycznego i tworzenia szablonów klas i funkcji.

Zakres tematyczny zajęć:

- polimorfizm statyczny,
- szablony funkcji,
- szablony klas,
- specjalizacja klas.

Pytania kontrolne:

1. Co to jest polimorfizm statyczny?
2. Jaka jest różnica między polimorfizmem statycznym a dynamicznym?
3. Do czego służą szablony?
4. Jak stworzyć szablon funkcji?
5. Jak stworzyć szablon klasy?
6. Do czego służy specjalizacja?

Polimorfizm statyczny:

Polimorfizm statyczny polega na przypisaniu konkretnej metody/funkcji każdemu wywołaniu na etapie kompilacji programu. Dotyczy on przeciążania metod (*ang. overload*), czyli tworzenia różnych metod o tej samej nazwie ale z różnym typem lub różną liczbą argumentów.

Szablony:

Szablony umożliwiają tworzenie kodu niezależnego od typów. Podczas kompilacji dochodzi do konkretyzacji szablonu, wtedy kompilator generuje kod do obsługi odpowiedniego typu na podstawie argumentów dostarczonych do parametrów szablonu. Kod źródłowy zawiera tylko funkcję/klasę, ale skompilowany kod może zawierać wiele kopii tej samej funkcji/klasy. Używanie szablonów pozwala na redukcję ilości kodu, w tym przypadku uniezależnienia kodu od typu danych. Dzięki użyciu szablonów nie trzeba pisać tego samego kodu dla różnych typów danych. Jako przykład można podać implementację funkcji sortującej dane w tablicy. Chcemy mieć możliwość posortowania zarówno danych typu całkowitego (tablica liczb całkowitych) jak i zmiennoprzecinkowego (tablica liczb zmiennoprzecinkowych). Jeśli stworzymy szablon funkcji, która będzie sortowała, wtedy nie będziemy musieli tworzyć dwóch funkcji – jednej dla liczb całkowitych a drugiej takiej samej dla liczb zmiennoprzecinkowych, zmieniając tylko i wyłącznie typ danych. Można tworzyć szablony funkcji oraz klas. Z reguły szablony umieszcza się w plikach nagłówkowych z rozszerzeniem `.h`. Można również umieścić kod w pliku źródłowym, ale częstsze rozwiązanie to umieszczanie szablonów w plikach nagłówkowych.

Szablony funkcji:

Listing 3.1. przedstawia w jaki sposób tworzyć szablon funkcji. Aby stworzyć szablon należy użyć słowa *template*, a następnie w ostrych nawiasach podać typy, każdy typ musi być poprzedzony słowem *typename* chyba, że jest określony.

```
1  template <typename T1, typename T2, ... >
2  type functionName(T1 arg1, T2 arg2, ... , other arguments){ }
```

Listing 3.1. Tworzenie szablonu funkcji

Listing 3.2. pokazuje, w jaki sposób wywołać funkcję szablonową. Po nazwie funkcji należy w ostrych nawiasach podać, z jakimi typami ma zostać wywołana funkcja, a następnie w nawiasach okrągłych trzeba podać argumenty. Jeśli stosujemy typy proste, na przykład: *int* czy *char*, nie trzeba ich pisać w ostrych nawiasach, ale zwiększa to czytelność kodu.

```
functionName<type1,type2, ... >(arguments);
```

Listing 3.2. Wywołanie funkcji szablonowej

Na listingu 3.3. przedstawiona została funkcja szablonowa *mySwap*, która zamienia dwie wartości między sobą. Dla funkcji stworzony został typ *T*. Funkcja przyjmuje referencję na dwa argumenty, po to, aby zmiana w funkcji była widoczna również po za funkcją. Dla uproszczenia funkcja została umieszczona w pliku *main.cpp*.

```
1  template <typename T>
2  void mySwap(T& a, T& b)
3  {
4      T temp;
5      temp=a;
6      a=b;
7      b=temp;
8  }
```

Listing 3.3. Tworzenie funkcji szablonowej *mySwap*

Listing 3.4. zawiera przykładowe wywołanie funkcji *mySwap*, dla typu całkowitego (*int*). Jeśli chcielibyśmy użyć tej funkcji dla typu znakowego (*char*), wystarczy tylko zmienić typ.

```
1  int main()
2  {
3      int a=6;
4      int b=8;
5      cout<<"przed: "<<a<<"   "<<b<<endl;
6      mySwap<int>(a,b) ;
7      cout<<"po: "<<a<<"   "<<b<<endl;
8
9      return 0;
10 }
```

Listing 3.4. Wywołanie funkcji *mySwap* dla typu *int* w funkcji *main*



Na rysunku 3.1. przedstawiony został wynik działania kodu z listingu 3.3.

```
przed: 6 8
po: 8 6
```

Rys. 3.1. Wynik działania kodu z listingu 3.3.

Rysunek 3.2. przedstawia działanie szablonu. W momencie kompilacji na podstawie szablonu generowany jest odpowiedni kod.

```
template <typename T>
void mySwap(T& a, T& b)
{
    T temp;
    temp=a;
    a=b;
    b=temp;
}

int main()
{
    int a=6;
    int b=8;
    mySwap<int>(a,b);

    char c='f';
    char d='g';
    mySwap<char>(c,d);
    return 0;
}
```

kompilator generuje kod

```
void mySwap(int& a, int& b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}

void mySwap(char& a, char& b)
{
    char temp;
    temp=a;
    a=b;
    b=temp;
}
```

Rys. 3.2. Przykład działania szablonu

Szablony klas:

Szablony klas, podobnie jak szablony funkcji, są przydatne, gdy klasa definiuje coś, co jest niezależne od typu danych. Listing 3.5. oraz listing 3.6. przedstawiają w jaki sposób tworzyć szablony klasy.

```
1 template <typename T1, typename T2, ... >
2 class ClassName { attributes and methods };
```

Listing 3.5. Tworzenie szablonu klasy

```
1 template <typename T1, typename T2, ... >
2 type ClassName<T1, T2, ... >::functionName(T1 a, arguments) { }
```

Listing 3.6. Odwołanie się do szablonu po za klasą w celu implementacji metody

Listing 3.7. pokazuje w jaki sposób stworzyć obiekt funkcji szablonej. Po nazwie klasy należy w ostrych nawiasach podać z jakimi typami ma zostać użyta klasa a następnie należy podać nazwę obiektu.

```
ClassName<type1,type2, ... > objectName(arguments);
```

Listing 3.7. Stworzenie obiektu klasy szablonej

Na listingach 3.8. – 3.9. przedstawiony został przykład zastosowania klasy szablonej. Została stworzona klasa *Adding*, która przechowuje jedną zmienną typu *T*. W klasie została stworzona metoda *add*, które będzie dodawała wartość przekazaną w argumencie do pola w klasie. Dodawanie dla typu *int* oraz dla typu *string* wygląda dokładnie tak samo, z tą różnicą

że dla typu *int* wartość jest zwiększana o składnik, a dla typu *string* do wartości doklejany jest składnik. Listing 3.8. zawiera:

- linijka 8 – deklaracja zmiennej typu *T*;
- linijka 10 – konstruktor przyjmuje jako argument zmienną typu *T*;
- linijka 11 – metoda *add*, która przyjmuje jako argument zmienną typu *T*;

```
1  #ifndef ADDING_H_INCLUDED
2  #define ADDING_H_INCLUDED
3  using namespace std;

4  template <typename T>
5  class Adding
6  {
7  private:
8      T element;
9  public:
10     Adding(T value);
11     void add(T value);
12     void show();
13 };

14 template <typename T>
15 Adding<T>::Adding(T value)
16 {
17     element=value;
18 }

19 template <typename T>
20 void Adding<T>::add(T value)
21 {
22     element=element+value;
23 }

24 template <typename T>
25 void Adding<T>::show()
26 {
27     cout<<"Element: "<<element<<endl;
28 }

29 #endif // ADDING_H_INCLUDED
```

Listing 3.8. Plik nagłówkowy klasy szablonowej *Adding*

Na listingu 3.9. przedstawiony został plik *main.cpp*, który zawiera tworzenie obiektów i wywoływanie metod klasy szablonowej *Adding*. Listing 3.10. zawiera:

- linijka 6 – stworzenie obiektu klasy *Adding* z typem *int*;
- linijka 9 – stworzenie obiektu klasy *Adding* z typem *string*.


```

1  #include <iostream>
2  #include "Adding.h"
3  using namespace std;
4  int main()
5  {
6      Adding<int> a1(5);
7      a1.add(6);
8      a1.show();

9      Adding<string> a2("Pro");
10     a2.add("gramowanie");
11     a2.show();

12     return 0;
13 }

```

Listing 3.9. Funkcja main – tworzenie obiektów klasy Adding

Na rysunku 3.3. pokazany został wynik działania kodu z listingu 3.9. Obiekt *a1* zawierał typ *int*, do jego konstruktora przekazana została wartość 5. Następnie do metody *add* została przekazana liczba 6 a więc wartość zwiększyła się do 11. Natomiast obiekt *a2* zawierał typ *string*, do jego konstruktora został przekazany ciąg znaków „Pro”, a następnie do metody *add* został przekazany ciąg znaków „gramowanie”, co w konsekwencji spowodowało, że pole *element* w obiekcie *a2* przechowuje słowo „Programowanie”.

```

Element: 11
Element: Programowanie

```

Rys. 3.3. Wynik działania kodu z listingu 3.10.

Specjalizacja szablonów:

Jeśli chcemy zaimplementować inny szablon, gdy określony typ jest przekazywany jako parametr szablonu, możemy zadeklarować specjalizację tego szablonu. Specjalizacja szablonu umożliwia dostosowanie zachowania szablonu do danego typu. Można dokonać pełnej lub częściowej specjalizacji. Częściowa specjalizacja nie nadpisuje ogólnego szablonu, a tylko niektórych jego elementów. Przed specjalizacją szablonu klasy musi być znana deklaracja ogólnego szablonu. Tworząc specjalizację szablonu można zdefiniować inne pola i metody niż w szablonie ogólnym lub można w nim wyspecjalizować tylko wybrane metody. Jeśli specjalizacja dotyczy całej klasy, instrukcję należy umieścić tylko przy deklaracji klasy, a jeśli funkcji – należy umieścić przed deklaracją funkcji. Jako przykład weźmy klasę *Student*, która przechowuje imię oraz ocenę studenta i zawiera metodę, która wyświetla ocenę studenta. Chcemy aby metoda w zależności od przekazanego typu miała inną implementację. W związku z tym musimy dokonać specjalizacji, aby wywołana metoda z typem *int* zachowywała się inaczej niż z typem *string*. Jeśli typ nie będzie ani *int* ani *string* metoda powinna zachować się jeszcze inaczej. Listing 3.10. przedstawia plik nagłówkowy *Student.h*, który zawiera ogólną klasę szablonową i jej specjalizacje. Listing 3.10. zawiera:

- linijki 4 – 32 – definicja ogólnego szablonu klasy *Student*, zawierająca również definicję metod *showMark*;

- linijki 33 – 37 – pełna specjalizacja metody *showMark* dla typu *int*. Definicja metody dla typu ogólnego jak i *int* jest taka sama. Można ale nie jest konieczne tworzenie całej klasy z danym typem;
- linijki 38 – 69 – pełna specjalizacja klasy dla typu *string*. Klasa zawiera metodę *showMark*, która nie pojawiła się w ogólnym szablonie (inny typ zwracany). Metoda *showMark* specjalnie jako typ zwracany ma *int*, aby pokazać, że w takich przypadkach potrzeba jest tworzenia całej klasy z danym typem, a jednej metody.

```
1  #ifndef STUDENT_H_INCLUDED
2  #define STUDENT_H_INCLUDED
3  using namespace std;

4  template <typename T>
5  class Student
6  {
7      private:
8          string name;
9          int mark;
10     public:
11         Student(int mark, string name)
12         {
13             this->mark = mark;
14             this->name = name;
15         }
16         Student()
17         {
18         }
19         void show()
20         {
21             cout <<"imie: "<<name<<" ocena "<<mark<<endl;
22         }
23         void showMark()
24         {
25             cout <<mark<<endl;
26         }
27     };

28     template<>
29     void Student<int>::showMark()
30     {
31         cout <<"Twoja ocena to: "<<mark<<endl;
32     }

33     template<>
34     class Student<string>
35     {
36     private:
37         string name;
38         int mark;
```



```

39 public:
40     Student(int mark, string name)
41     {
42         this->mark = mark;
43         this->name = name;
44     }
45     Student()
46     {
47     }
48     void show()
49     {
50         cout <<"imie: "<<name<<" ocena "<<mark<<endl;
51     }
52     int showMark()
53     {
54         string word [6]= {"jedylnka","dwojka","trojka","czworka",
55                             "piatka","szostka"};
56         cout <<"Twoja ocena to: "<<word[mark-1]<<endl;
57         return mark;
58     }
59 };
60 #endif // STUDENT_H_INCLUDED

```

Listing 3.10. Plik Student.h

Na listingu 3.11. przedstawiony został kod, który trzeba dodać do funkcji *main*. Stworzone zostały trzy obiekty z różnymi typami. Dla typu *int* oraz *string* została dokonana specjalizacja metody *showMark*, natomiast dla typu *float* zostanie wywołana metoda *showMark* z szablonu ogólnego klasy. Należy pamiętać o dołączeniu pliku nagłówkowego *Student.h* do pliku *main.cpp*.

```

1     Student<float> s(5,"Ala");
2     s.showMark();
3     Student<int> s1(5,"Ala");
4     s1.showMark();
5     Student<string> s2(5,"Ala");
6     s2.showMark();

```

Listing 3.11. Plik main.cpp

Rysunek 3.4. prezentuje działanie programu z listingu 3.11.

```

5
Twoja ocena to: 5
Twoja ocena to: piatka

```

Rys. 3.4. Wynik działania kodu z listingu 3.11.

Funkcje szablonowe zawierające tablice obiektów:

W celu zaprezentowania uniwersalności szablonów funkcji stworzona została funkcja która przyjmuje statyczną tablicę dynamicznych obiektów typu *T*. Funkcja ma za zadanie wywołać metodę *showAll* dla każdego obiektu. Funkcja szablonowa będzie wykonywać się dla każdego rodzaju obiektu jeśli w klasie tego obiektu jest metoda *show*. Listing 3.12. prezentuje kod funkcji *showAll*. Funkcja została dodana do pliku *main.cpp*.

```

1  template <typename T>
2  void showAll(T* tab[], int n)
3  {
4      for(int i=0;i<n;i++)
5          tab[i]->show();
6  }
```

Listing 3.12. Funkcja *showAll*

W celu zaprezentowania działania funkcji *showAll*, w funkcji *main* stworzone zostały dwie tablice wykorzystując wcześniej stworzone klasy *Adding* oraz *Student*. Jedna tablica *arrAdd* przechowuje wskaźniki na obiekty klasy *Adding*, a druga – *arrStu* przechowuje wskaźniki na obiekty klasy *Student*. Listing 3.13. przedstawia tworzenie tablic oraz wywołanie funkcji *showAll*. Listing 3.13. zawiera:

- linijki 1 - 5 – tworzenie tablicy obiektów klasy *Adding*;
- linijka 6 – wywołanie funkcji *showAll* dla tablicy przechowującej obiekty z klasy *Adding*;
- linijki 7 – 11 – podobnie dla klasy *Student*;
- linijki 12 – 15 – zwolnienie przydzielonej pamięci.

```

1  Adding<int>* arrAdd[3];
2  for(int i=0;i<3;i++)
3  {
4      arrAdd[i]=new Adding<int>(i);
5  }
6  showAll(arrAdd,3);

7  Student<string>* arrStu[2];
8  for(int i=0;i<3;i++){
9      arrStu[i]=new Student<string>(i+2,"Ala");
10 }
11 showAll(arrStu,3);

12 for(int i=0; i<3; i++)
13     delete arrAdd[i];

14 for(int i=0; i<2; i++)
15     delete arrStu[i];
```

Listing 3.13. Plik *main.cpp*

Na rysunku 3.5. przedstawiony został wynik działania kodu z listingu 3.13.

```
Element: 0
Element: 1
Element: 2
imie: Ala ocena 2
imie: Ala ocena 3
imie: Ala ocena 4
```

Listing 3.5. Wynik działania kodu z listingu 3.13.

Klasa szablonowa `numeric_limits`:

Szablon klasy `numeric_limits` udostępnia specjalizacje dla wszystkich typów podstawowych. Umożliwia na przykład sprawdzenie jaka jest wartość maksymalna dla danego typu i sprawdzenie różnych właściwości typów. Listing 3.14. przedstawia jedno z zastosowań klasy `numeric_limits`. Stworzona została szablonowa funkcja `checkType`, w której sprawdzana jest maksymalna wartość typu `T` za pomocą klasy szablonowej `numeric_limits`.

```
1  #include <iostream>
2  using namespace std;
3  template <typename T>
4  void checkType(T a)
5  {
6      T val=numeric_limits<T>::max();
7      cout<<val<<endl;
8  }
9  int main()
10 {
11     checkType<int>(3);
12     return 0;
13 }
```

Listing 3.14. Przykład zastosowania klasy `numeric_limits`

Zadania do wykonania:

Zadanie 3.1. Przykładowy kod

Uruchom kod przedstawiony i umówiony podczas tego laboratorium.

Zadanie 3.2. Element minimalny

Napisz funkcję szablonową, która przyjmuje jako argument tablicę o typie będącym parametrem szablonu i rozmiar tablicy typu całkowitego. Zadaniem funkcji jest znalezienie elementu minimalnego w tablicy i zwrócenie go.



Zadanie 3.3. Tablica

Stwórz szablon klasy *Array* o jednym parametrze *T*, która będzie przechowywać następujące atrybuty i metody:

- atrybuty prywatne:
 - pole do przechowywania tablicy o typie T^* ,
 - maksymalny rozmiar tablicy,
 - indeks pierwszego wolnego miejsca w tablicy. Zakładamy, że elementy do tablicy są dodawane po kolei;
- konstruktor, który jako argument przyjmie rozmiar tablicy i zaalokuje pamięć;
- konstruktor bezargumentowy ustawiający rozmiar tablicy liczbą 10 oraz przydzielający pamięć;
- destruktor, który zwolni przydzieloną pamięć;
- metodę umożliwiającą sortowanie rosnąco elementów w tablicy;
- metodę zwracającą element maksymalny w tablicy;
- metodę wyświetlającą zawartość tablicy;
- metodę umożliwiającą dodanie jednego elementu do tablicy;
- metodę zwracającą element znajdujący się pod podanym w argumencie indeksem.

Dokonaj specjalizacji klasy *Array* dla typu *string*, w taki sposób aby:

- metoda sortująca sortowała słowa pod względem ich długości;
- metoda zwracająca element maksymalny zwracała najdłuższy napis znajdujący się w tablicy.

W funkcji *main* należy przetestować klasę.



LABORATORIUM 4. STL. KONTENERY SEKWENCYJNE.

Cel laboratorium:

Omówienie elementów biblioteki STL i kontenerów sekwencyjnych.

Zakres tematyczny zajęć:

- biblioteka STL: kontenery, iteratory, algorytmy i funktory,
- kontenery sekwencyjne.

Pytania kontrolne:

1. Co zawiera biblioteka STL?
2. Jaka jest różnica pomiędzy kontenerem a adaptorem?
3. Do czego służy iterator?
4. Co to jest *vector*?
5. Jakie są funkcje umożliwiające pracę z kontenerem *vector*?
6. Jakie znasz algorytmy z biblioteki STL?
7. Co to jest funktor i do czego służy?

Standard Template Library (STL):

Jest to standardowa biblioteka szablonów. W skład biblioteki wchodzi: kontenery, iteratory, algorytmy oraz funktory. Wyróżniane są kontenery sekwencyjne, asocjacyjne oraz ich adaptory. Kontenery to struktury danych, które służą do przechowywania danych. Wszystkie elementy kontenera muszą być tego samego typu. Adapter umożliwia dostosowanie kontenera aby zapewnić odpowiedni interfejs. Iteratory są uogólnieniem wskaźników i umożliwiają pracę z różnymi kontenerami. Iteratory udostępniają pojedynczy element zawarty w kolekcji, można je inkrementować lub dekrementować. Każdy kontener ma zdefiniowany własny iterator. W bibliotece *STL* znajduje się również wiele typowych algorytmów, na przykład, do sortowania albo wyszukiwania danych. Funktory nazywane również obiektami funkcyjnymi są obiektami klasy, która implementuje operator `()`.

Kontenery sekwencyjne:

Kontenery sekwencyjne to kontenery, które zawierają elementy tego samego typu, a elementy są uporządkowane i można do nich uzyskać dostęp w sposób sekwencyjny. Do kontenerów sekwencyjnych należą:

- *array* – tablica o stałym rozmiarze;
- *vector* – tablica dynamiczna, w każdym momencie można zwiększyć jej wielkość;
- *deque* – kolejka dwukierunkowa;
- *list* – lista dwukierunkowa;
- *forward_list* – lista jednokierunkowa.

Kontenery różnią się między sobą na przykład w sposobie przechowywania danych w pamięci, szybkością działania niektórych metod lub w sposobie dostępu do danych ale jednymi z najczęściej używanych funkcji występujących z różnymi argumentami, przy obsłudze kontenerów sekwencyjnych są:

- *push_back* – dodawanie wartości na koniec kontenera;



- *pop_back* – usuwanie z końca kontenera;
- *at* – dostęp do *i-tego* elementu;
- *push_front* – dodawanie elementu na początek;
- *pop_front* – usuwanie elementu z początku;
- *insert* – wstawianie elementu na określoną pozycję;
- *erase* – usuwanie elementu z określonej pozycji;
- *begin* – zwracanie iteratora(wskaźnika) na pierwszy element;
- *end* – zwracanie iteratora(wskaźnika) na element będący za ostatnim elementem znajdującym się w kontenerze;
- *front* – zwracanie elementu znajdującego się na początku kontenera;
- *back* – zwracanie elementu znajdującego się na końcu kontenera;
- *clear* - czyszczenie kontenera;
- *size* – zwracanie ile elementów znajduje się w kontenerze;
- *empty* – sprawdzenie czy kontener jest pusty.

Przykładową różnicą pomiędzy kontenerem *vector* a *deque* jest to, że wstawianie i usuwanie elementów z początku jest bardziej efektywne przy użyciu kontenera *deque*. Jako przykład różnicy między kontenerem *vector* a *list* można podać dostęp do elementów. W kontenerze *list* nie ma dostępu do *i-tego* elementu.

Vector jako przykład kontenera sekwencyjnego:

Vector jest jednym z najbardziej popularnych kontenerów. *Vector* ma charakter dynamiczny, więc jego rozmiar zwiększa się wraz ze wstawianiem elementów. W tablicy mamy dostęp do każdego elementu, jeśli znamy jego pozycję. *Vector* posiada przeciążony operator [], więc do elementów można odwoływać się jak w tablicy. Element można pobrać, zmienić lub usunąć. Elementy można dodawać na dowolną pozycję ale tylko dodawanie na koniec jest realizowane w czasie stałym. Dodawanie na początek lub w środek ma złożoność liniową. Podobnie sytuacja wygląda z usuwaniem. Listing 4.1. pokazuje w jaki sposób można zadeklarować *vector*. *Vector* jest klasą szablonową więc w ostrych nawiasach należy podać jaki typ będzie przechowywać.

```
vector<type>vectorName
```

Listing 4.1. Deklaracja kontenera *vector*

Listing 4.2. przedstawia przykład użycia kontenera *vector*. Listing 4.2. zawiera:

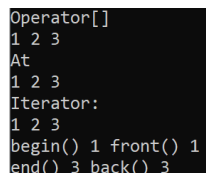
- linijka 2 – dołączenie biblioteki do obsługi kontenera *vector*;
- linijka 6 – deklaracja kontenera *vector* przechowującego liczby całkowite;
- linijki 7 – 9 – dodawanie liczb na koniec;
- linijki 10 – 13 – pierwszy sposób dostępu do elementów kontenera *vector*, za pomocą operatora [];
- linijki 14 – 17 – drugi sposób dostępu do elementów kontenera *vector*, za pomocą metody *at*;
- linijki 18 - 22 – trzeci sposób dostępu do elementów kontenera *vector*, za pomocą iteratora. Zmienna *it* przechowuje adres, a więc, żeby uzyskać wartość, należy użyć operatora wyłuskania czyli *;
- linijki 23 – 24 – uzyskanie dostępu do pierwszego elementu za pomocą funkcji *begin*, która zwraca iterator lub za pomocą funkcji *front*, która zwraca wartość;

- linijki 25 – 26 - uzyskanie dostępu do ostatniego elementu za pomocą funkcji *end*, która zwraca iterator lub za pomocą funkcji *back*, która zwraca wartość.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main()
5  {
6      vector<int> vec;
7
8      vec.push_back(1);
9      vec.push_back(2);
10     vec.push_back(3);
11
12     cout<<"Operator[]"<<endl;
13     for(int i=0; i<vec.size(); i++)
14         cout<<vec[i]<<" ";
15     cout<<endl;
16
17     cout<<"At"<<endl;
18     for(int i=0; i<vec.size(); i++)
19         cout<<vec.at(i)<<" ";
20     cout<<endl;
21
22     cout<<"Iterator: "<<endl;
23     vector<int>::iterator it;
24     for(it=vec.begin(); it != vec.end(); it++)
25         cout <<*it<<" ";
26     cout<<endl;
27
28     cout<<"begin() "<< *vec.begin()<<" front() "
29         <<vec.front()<<endl;
30     cout<<"end() "<< *(vec.end()-1)<<" back() "
31         <<vec.back()<<endl;
32     return 0;
33 }
```

Listing 4.2. Przykład użycia kontenera vector – plik main.cpp

Rysunek 4.1. przedstawia wynik kodu z listingu 4.2.



```
Operator[]
1 2 3
At
1 2 3
Iterator:
1 2 3
begin() 1 front() 1
end() 3 back() 3
```

Rys. 4.1. Wynik działania kodu z listingu 4.2.

W dalszej części przykładów często będzie wyświetlana zawartość kontenerów, więc stworzona została szablonowa funkcja umożliwiająca wyświetlanie zawartości kontenera. Listing 4.3. prezentuje kod tej funkcji. Konieczne jest dodanie słowa *typename* w linijce 4, ponieważ *kontener<type>::iterator* jest typem.

```
1  template <typename T>
2  void show(T &con)
3  {
4      for(typename T::iterator it=con.begin( );
5          it!=con.end( ); it++)
6          cout<<*it<<" ";
7      cout<<endl;
8  }
```

Listing 4.3. Funkcja szablonowa wyświetlająca zawartość kontenera

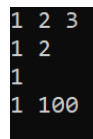
Kontynuując przykład z możliwościami kontenera *vector* dodajmy i przeanalizujmy następujący kod, który został przedstawiony na listingu 4.4. Listing 4.4. zawiera:

- linijka 2 – usunięcie ostatniego elementu;
- linijka 4 – usunięcie elementów z podanego zakresu [a,b), a w tym przypadku [adres początkowy+1, adres początkowy+2), czyli zostanie usunięty drugi element z kontenera;
- linijka 6 – wstawianie elementu pod wskazany adres;
- linijka 8 – wyczyszczenie zawartości kontenera.

```
1  show(vec);
2  vec.pop_back();
3  show(vec);
4  vec.erase(vec.begin()+1,vec.begin()+2);
5  show(vec);
6  vec.insert(vec.begin()+1,100);
7  show(vec);
8  vec.clear();
9  show(vec);
```

Listing 4.4. Kontynuacja przykładu

Rysunek 4.2. przedstawia wynik działania kodu z listingu 4.4.



```
1 2 3
1 2
1
1 100
```

Rys. 4.2. Wynik działania kodu z listingu 4.4.

W przypadku kontenera *vector*, można przy jego deklaracji określić wstępny rozmiar kontenera i wypełnić go liczbami – kod na listingu 4.5. Stworzony został *vector* o długości 5, który został wypełniony liczbami 8.2. Jeśli nie zostanie podana liczba jaką należy wypełnić *vector*, *vector* zostanie wypełniony 0.

```
1  vector<float>vec2 (5, 8.2) ;
2  show (vec2) ;
3  vec2.push_back (3.1) ;
4  show (vec2) ;
```

Listing 4.5. Deklaracja kontenera vector o określony rozmiarze

Rysunek 4.3. przedstawia wynik działania listingu 4.5.

```
8.2 8.2 8.2 8.2 8.2
8.2 8.2 8.2 8.2 8.2 3.1
```

Rys. 4.3. Wynik działania kodu z listingu 4.5.

Algorytmy uogólnione:

Algorytmy w *STL* nazywane algorytmami uogólnionymi realizują popularne operacje takie jak na przykład sortowanie i działają dla różnych typów danych. Działają nie tylko dla kontenerów ale również dla tablic. Większość algorytmów jako dwa pierwsze argumenty pobiera dwa iteratory, które określają zakres, na którym będzie działać algorytm. Zakres zdefiniowany jest jako przedział $[it1; it2)$. Algorytm zaczyna działać od *it1* a kończy na *it2-1*. W celu używania algorytmów należy dołączyć bibliotekę *algorithm*. Algorytmy można podzielić na algorytmy, które modyfikują zawartość kontenera oraz na takie które nie modyfikują. Do algorytmów, które nie modyfikują zawartości kontenera należą między innymi:

- *find* – znajdowanie pierwszego wystąpienia wartości w ciągu;
- *find_if* – znajdowanie, pierwszego wystąpienia wartości w ciągu spełniającej predykat;
- *count* – policzenie liczby wystąpień w ciągu;
- *count_if* – zliczanie wystąpienia w ciągu wartości spełniającej predykat;
- *max_element* – znajdowanie elementu maksymalnego;
- *for_each* – umożliwienie dostępu do elementu oraz wykonanie na nim operacji;
- *binary_search* – sprawdzanie czy element znajduje się w ciągu;

Do algorytmów modyfikujących zawartość kontenera należą między innymi:

- *transform* – zastosowanie operacji do każdego elementu w ciągu;
- *replace* – zastąpienie elementów podaną wartością;
- *fill* – zastąpienie każdego elementu podaną wartością;
- *reverse* – odwrócenie kolejności elementów;
- *sort* – sortowanie elementów;
- *random_shuffle* – losowe ustawienie elementów;
- *merge* – scalanie kontenerów.

Wszystkie algorytmy, które znajdują się w bibliotece *algorithm* można znaleźć na stronie <https://en.cppreference.com/w/cpp/algorithm> Na listingu 4.6. przedstawione zostało użycie niektórych algorytmów na przykładzie kontenera *vector*. Kod można dodać do rozpatrywanego podczas zajęć przykładu. Listing 4.6. zawiera:

- linijka 1 – stworzenie kontenera wypełnionego liczbami;
- linijka 3 – sortowanie;
- linijka 6 – wyszukiwanie liczby 3;
- linijka 8 – zliczanie ile razy występuje liczba 3;
- linijka 10 – odwrócenie kolejności części kontenera;



- linijka 13 – zmiana kolejności elementów w kontenerze;
- linijka 16 – wyszukanie elementu maksymalnego, funkcja zwraca iterator stąd trzeba użyć operatora * aby wyłuskać wartość;
- linijki 17-23 – połączenie dwóch posortowanych kontenerów. Jako piąty argument funkcja *merge* przyjmuje adres spod którego zacznie się łączenie.

```
1  vector<int>num={3,6,3,8,9,1,3};
2  show(num);
3  sort(num.begin(), num.end());
4  show(num);

5  cout<<"binary search"<<endl;
6  cout<<binary_search(num.begin(), num.end(), 3)<<endl;

7  cout<<"count"<<endl;
8  cout<<count(num.begin(), num.end(), 3)<<endl;

9  cout<<"reverse"<<endl;
10 reverse(num.begin(), num.begin()+4);
11 show(num);

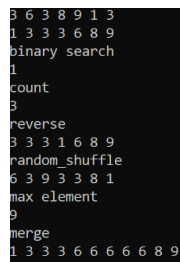
12 cout<<"random_shuffle"<<endl;
13 random_shuffle(num.begin(), num.end());
14 show(num);

15 cout<<"max element"<<endl;
16 cout<<*max_element(num.begin(), num.end())<<endl;

17 cout<<"merge"<<endl;
18 vector<int>num2(4,6);
19 vector<int>res(num.size()+num2.size());
20 sort(num.begin(), num.end());
21 merge(num.begin(), num.end(), num2.begin(),
22        num2.end(), res.begin());
23 show(res);
```

Listing 4.6. Przykład użycia algorytmów z biblioteki *algorithm*

Rysunek 4.4. przedstawia wynik działania kodu z listingu 4.6.



```
3 6 3 8 9 1 3
1 3 3 6 8 9
binary search
1
count
3
reverse
3 3 3 1 6 8 9
random_shuffle
5 3 9 3 3 8 1
max element
9
merge
1 3 3 3 6 6 6 6 8 9
```

Rys. 4.4. Wynik działania kodu z listingu 4.6.



Do niektórych algorytmów mogą być przekazywane funkcje na przykład:

- *for_each* – algorytm wykonuje operację dla każdego elementu. Operacją tą jest funkcja, która umożliwia dostęp do elementu;
- *count_if* – algorytm zlicza liczbę wystąpień jeśli spełniony jest predykat. Predykat to funkcja, która zwraca typ *bool*.

Listing 4.7. przedstawia implementację trzech funkcji, które zostaną użyte w algorytmach *for_each* oraz *count_if* na listingu 4.8.

```
1 void print(int& el)
2 {
3     cout<<el<<"***";
4 }
5 void add10(int &el)
6 {
7     el=el+10;
8 }
9 bool isEven(int x)
10 {
11     if (x%2 == 0)
12         return true;
13     else
14         return false;
15 }
```

Listing 4.7. Funkcje, które zostaną użyte w algorytmach *for_each* oraz *count_if*

Listing 4.8. przedstawia sposób użycia algorytmów *for_each* oraz *count_if*:

- linijka 1 – wywołanie funkcji *for_each*, która zawiera dwa iteratory (początek i koniec kontenera) oraz funkcję *print*. Funkcja *print* zostanie wywołana dla każdego elementu;
- linijka 2 – wywołanie funkcji *for_each*. Dla każdego elementu zostanie wywołana funkcja *add10*;
- linijka 6 – wywołanie funkcji *count_if*, która zawiera dwa iteratory (początek i koniec kontenera) oraz funkcji *isEven*. Algorytm zlicza wystąpienia tych elementów kontenera, dla których funkcja *isEven* zwróci wartość *true*.

```
1 for_each(num.begin(), num.end(), print);
2 for_each(num.begin(), num.end(), add10);
3 cout<<endl;
4 for_each(num.begin(), num.end(), print);
5 cout<<endl;
6 cout<<count_if(num.begin(), num.end(), isEven);
```

Listing 4.8. Funkcje dla algorytmów *for_each* oraz *count_if*

Na rysunku 4.5. przedstawiony został wynik kodu z listingu 4.8.

```
1 3 3 3 6 6 6 6 6 8 9
1***3***3***3***6***8***9***
11***13***13***13***16***18***19***
2
```

Rys. 4.5. Wynik działania kodu z listingu 4.8.

Funktory:

Funktory są obiektami klasy implementującej operator(). Mogą być nazywane również obiektami funkcyjnymi. Funktory działają podobnie jak funkcje. Biblioteka *STL* zawiera kilka zdefiniowanych już funktorów. W zależności od stosowanego kompilatora może zaistnieć potrzeba dołączenia biblioteki *functional*. Można napisać również własne obiekty funkcyjne. W bibliotece znajdują się między innymi funktory, które umożliwiają porównanie:

- *plus<type>()*,
- *less<type>()*,
- *greater<type>()*,
- *equal_to()*.

Adaptory funktorów umożliwiają przekształcenie dwuargumentowego funktora w jednoargumentowy związując jeden z argumentów z konkretną wartością:

- *bind1st(funktor, value)* – wywołanie funkcji dwuargumentowej, której pierwszym argumentem jest *value*;
- *bind2st(funktor, value)* – wywołanie funkcji dwuargumentowej, której drugim argumentem jest *value*;

W standardzie języka C++11 pojawiła się również funkcja *bind*, która jest uogólnieniem dwóch wyżej wymienionych funkcji. Pochodzi ona z biblioteki *Boost* i zostanie przedstawiona w ramach laboratorium numer 6. Na listingu 4.9. pokazane zostały przykładowe zastosowania wbudowanych funktorów. Listing 4.9. zawiera:

- linijka 3 – algorytm *count_if* zlicza ile liczb jest większych od 0.
- linijka 6 – zadeklarowanie zmiennej typu funktor;
- linijki 7 - 8 – algorytm *count_if* zlicza ile liczb jest większych od 0;
- linijka 12 – algorytm *sort* sortuje liczby malejąco wykorzystując porównanie *liczba1>liczba2*.

```

1  vector<int> num3={2, -6, 7, 3, 0, 8, -9, -2};
2  for_each(num3.begin(),num3.end(),print);
3  int howMany=count_if(num3.begin( ), num3.end( ),
4                      bind2nd(greater<int>( ),0));
5  cout<<endl<<"wersja1: wartosc > 0  "<<howMany<<endl;

6  greater<int> f;
7  howMany=count_if(num3.begin( ), num3.end( ),
8                  bind2nd(f,0));
9  cout<<"wersja2: wartosc > 0  "<<howMany<<endl;

10 cout<<"Przed sortowaniem"<<endl;
11 for_each(num3.begin(),num3.end(),print);
12 sort(num3.begin( ), num3.end( ), greater<int>());
13 cout<<endl<<"Po sortowaniu"<<endl;
14 for_each(num3.begin(),num3.end(),print);

```

Listing 4.9. Zastosowanie wbudowanych funktorów

Na rysunku 4.6. przedstawiony został wynik działania kodu z listingu 4.9.

```
2***-6***7***3***0***8***-9***-2***
wersja1: wartosc > 0 4
wersja2: wartosc > 0 4
Przed sortowaniem
2***-6***7***3***0***8***-9***-2***
Po sortowaniu
8***7***3***2***0***-2***-6***-9***
```

Rys. 4.6. Wynik działania kodu z listingu 4.9.

Można również zaimplementować swój własny funktor, z metodą implementującą operator(). Na listingach 4.7. oraz 4.8. zrealizowany został między innymi przykład ze liczeniem ile liczb parzystych jest w kontenerze *vector*. Teraz ten przykład zostanie przedstawiony w wersji z operatorem (). W tym celu stworzona została klasa *Even*, w której zaimplementowany został operator(). Na listingu 4.10. zaprezentowany został plik nagłówkowy *Even.h* a na listingu 4.11. plik źródłowy *Even.cpp*. Klasa *Even* posiada pole *divider* oraz metodę *operator()*. Stworzone zostało pole *divider* aby potem w łatwy sposób można było policzyć nie tylko liczby parzyste ale na przykład podzielne przez 3 itd.

```
1  #ifndef EVEN_H_INCLUDED
2  #define EVEN_H_INCLUDED
3  class Even
4  {
5  private:
6      int divider;
7  public:
8      Even(int divider);
9      bool operator( )(int x);
10 };
11 #endif // EVEN_H_INCLUDED
```

Listing 4.10. Plik nagłówkowy *Even.h*

Listing 4.10. zawiera implementację operatora (). W tej metodzie mówimy co ma zrobić algorytm z każdym elementem. W tym przypadku operator () zwraca *true* lub *false*, ponieważ takiego typu wymagają funkcje implementujące algorytmy.

```
1  #include "Even.h"
2  Even:: Even(int divider)
3  {
4      this->divider=divider;
5  }
6  bool Even::operator( )(int x)
7  {
8      if (x%divider == 0)
9          return true;
10     else
11         return false;
12 }
```

Listing 4.11. Plik źródłowy *Even.cpp*

Do pliku *main.cpp* należy dodać plik *Even.h*, a do funkcji *main* dodać kod z listingu 4.12. Listing 4.12. przedstawia wywołanie funkcji *count_if* na dwa sposoby. Jeśli w klasie został zdefiniowany operator() to obiekt tej klasy z podanymi argumentami spowoduje wywołanie operatora(). W linijce 4 do konstruktora została przekazana liczba 2, aby policzyć ile jest liczb parzystych. Zarówno w jednym jak i w drugim przypadku zostały zwrócone te same wartości.

```
1  for_each(num.begin(), num.end(), print);
2  cout<<endl;
3  cout<<count_if(num.begin(), num.end(), isEven)<<endl;
4  cout<<count_if(num.begin(), num.end(), Even(2))<<endl;
```

Listing 4.12. Wywołanie funkcji *count_if* w dwóch wersjach

Na rysunku 4.7. przedstawiony został wynik działania kodu z listingu 4.12.

```
11***13***13***13***16***18***19***
2
2
```

Rys. 4.7. Wynik działania kodu z listingu 4.12.

Ostatni zaprezentowany przykład będzie dotyczył tego, w jaki sposób posortować dane, jeśli elementem w kontenerze jest obiekt jakiejś klasy. W tym celu została stworzona bardzo prosta klasa *Student*, która zawiera imię oraz ocenę studenta. Klasa posiada konstruktor oraz metodę zwracającą ocenę studenta. Pliki: nagłówkowy oraz źródłowy zostały przedstawione na listingu 4.13. oraz 4.14.

```
1  #ifndef STUDENT_H_INCLUDED
2  #define STUDENT_H_INCLUDED
3  #include<iostream>
4  using namespace std;
5  class Student
6  {
7  private:
8      int mark;
9      string name;
10 public:
11     Student(int mark, string name);
12     int getMark();
13 };
14 #endif // STUDENT_H_INCLUDED
```

Listing 4.13. Plik nagłówkowy *Student.h*

```
1  #include "Student.h"
2  Student::Student(int mark, string name)
3  {
4      this->mark=mark;
5      this->name=name;
6  }
7  int Student::getMark()
```




```
8 {
9     return mark;
10 }
```

Listing 4.14. Plik źródłowy *Student.cpp*

Następnie została stworzona klasa *Compare*, która posiada implementację operatora(). W tym operatorze należy zdefiniować, co oznacza porównanie dwóch studentów. W naszym przypadku chcemy posortować studentów rosnąco względem oceny jaką posiadają. Na listingu 4.15. oraz 4.16. przedstawiony został kod pliku nagłówkowego *Compare.h* oraz pliku źródłowego *Compare.cpp*.

```
1  #ifndef COMPARE_H_INCLUDED
2  #define COMPARE_H_INCLUDED
3  #include "Student.h"
4  class Compare
5  {
6  public:
7      bool operator( ) (Student &s1, Student &s2);
8  };
9  #endif // COMPARE_H_INCLUDED
```

Listing 4.15. Plik źródłowy *Compare.h*

Listing 4.16. zawiera “instrukcję” dla funkcji *sort* w jaki sposób należy dokonać porównania dwóch studentów. Jak widać w linijce 4 porównywane są oceny studentów. Zwrócona wartość operatora() wskazuje, czy element przekazany jako pierwszy argument jest uważany za poprzedzający drugi w określonej kolejności, którą definiuje. W tym przypadku będzie to sortowanie rosnące.

```
1  #include "Compare.h"
2  bool Compare::operator( ) (Student &s1, Student &s2)
3  {
4      return s1.getMark( ) < s2.getMark( );
5  }
```

Listing 4.16. Plik źródłowy *Compare.cpp*

Do pliku *main.cpp* należy dodać nagłówki: *Student.h* oraz *Compare.h*. Dodatkowo została stworzona funkcja *showMark()*, która będzie wykorzystana do wyświetlenia ocen studentów, jej kod znajduje się na listingu 4.17.

```
1  void showMark(Student &s)
2  {
3      cout<<s.getMark()<<" ";
4  }
```

Listing 4.17. Funkcja *showMark*

Na listingu 4.18 przedstawiony został kod, który należy dodać do funkcji *main*. Listing 4.18. zawiera:

- linijki 1 - 4 – stworzenie kontenera *vector*, który przechowuje obiekty klasy *Student*;
- linijka 7 – wywołanie funkcji *sort* z własnym funktorem.

```
1 vector<Student>st;  
2 st.push_back(Student(1,"Ala"));  
3 st.push_back(Student(5,"Ola"));  
4 st.push_back(Student(2,"Piotr"));  
5 for_each(st.begin(),st.end(),showMark);  
6 cout<<endl;  
7 sort(st.begin(),st.end(),Compare());  
8 for_each(st.begin(),st.end(),showMark);
```

Listing 4.18. Porównanie obiektów

Na rysunku 4.8. przedstawiony został wynik działania kodu z listingu 4.18.

```
1 5 2  
1 2 5
```

Rys. 4.8. Wynik działania kodu z listingu 4.18.

Adaptory kontenerów:

Adapter umożliwia dostosowanie kontenera aby zapewnić odpowiedni interfejs. Adaptery kontenerów używa się tak samo jak kontenery. Do adapterów kontenerów należą:

- *stack* – stos,
- *queue* – kolejka,
- *priority_queue* – priorytetowa kolejka.

Zadania do wykonania:

Zadanie 4.1. Przykładowy kod

Uruchom kod przedstawiony podczas laboratorium.

Zadanie 4.2. Losowanie

Napisz program, który losuje dodatnią liczbę całkowitą n a następnie losuje n liczb całkowitych z przedziału $[-100,100]$ i wstawia je do listy, w taki sposób aby na początku listy były elementy dodatnie lub równe 0 a na końcu ujemne. Program powinien wypisać na wyjściu zawartość listy.

Zadanie 4.3. Miasto

Stwórz klasę *Citizen*, która będzie reprezentowała mieszkańca i będzie zawierać następujące pola oraz metody:

- pola prywatne: *name*, *surname* typu *string*, *age* typu *int*, *sex* typu *char* oraz *postal_code* typu *string*;
- konstruktory: bezargumentowy oraz pozwalający na inicjalizację pól klasy;
- metodę *show* wyświetlającą informację o mieszkańcu;

- odpowiednie gettery do pól.

Stwórz klasę *City*, która będzie zawierać następujące pola oraz metody:

- pola prywatne: *citizens* - wektor przechowujący mieszkańców – obiekty z klasy *Citizen*, *city_name* – nazwa miasta;
- konstruktor inicjalizujący nazwę miejscowości;
- metodę *addCitizen*, która umożliwia dodanie mieszkańca do miejscowości. Metoda ma przyjąć obiekt klasy *Citizen*;
- metodę *deleteCitizen*, która usuwa mieszkańca o podanym nazwisku i wieku. Zakładamy, że połączenie nazwiska i wieku jest unikatowe. Nazwisko oraz wiek powinny zostać przekazane jako argumenty metody;
- metodę *show_citizens*, która wyświetla wszystkich mieszkańców miasta;
- metodę *show_city*, która wyświetla informację o nazwie miasta;
- metodę *women()*, która zwraca liczbę kobiet w mieście;
- metodę *city_citizens()*, która zwraca liczbę mieszkańców w mieście;
- metodę *adults()*, która zwraca liczbę pełnoletnich mieszkańców miasta;
- metodę *postal_codes()*, metoda wyświetla statystykę kodów pocztowych swoich mieszkańców, np. : „20-389 -> 3 mieszkańców, 30-678 -> 10 mieszkańców” oraz zwraca liczbę unikatowych kodów pocztowych;

Podpowiedź. Nie wiadomo ile jest unikatowych kodów, więc można stworzyć klasę pomocniczą przechowując kod pocztowy oraz liczbę mieszkańców i odpowiednio zwiększać liczbę mieszkańców. Obiekty tej klasy należy przechowywać w kontenerze. W ramach ćwiczenia wybierz inny kontener niż wektor.

Należy zaimplementować następujące funkcje:

- *void showCities(vector<City> c)*, funkcja wyświetla informacje o miastach;
- *void the_most(vector<City> c, int mode)*, funkcja szuka danych określonych przez parametr *mode* (tryb) oraz wyświetla informacje na konsolę. Każdy z podpunktów powinien zostać zrealizowany w oddzielnej funkcji. Tryb:
 1. Miasto, w którym jest najwięcej różnych kodów pocztowych;
 2. Miasto, w którym mieszka najmniej mieszkańców;
- *void statistic(vector<City> c)*, funkcja wyświetlająca statystykę, dla każdego miasta: nazwę miasta, liczbę mieszkańców tego miasta wraz z podziałem na liczbę kobiet oraz mężczyzn, osób niepełnoletnich i pełnoletnich;
- *void sort_cities(vector<City> &c)*, funkcja sortująca miasta rosnąco pod względem liczby mieszkańców.

W funkcji *main* należy pokazać działanie zaimplementowanych metod oraz funkcji. W zadaniu można używać algorytmów z biblioteki *STL*.

Zadanie 4.4. Sortowanie

Stwórz *vector*, który będzie przechowywał liczby całkowite. Za pomocą funkcji *sort* z biblioteki *STL* należy posortować dane w kontenerze *vector*:

- rosnąco według sumy cyfr w liczbie;
- malejąco według liczby cyfr.

Do wyświetlania zawartości kontenera należy użyć funkcji *for_each*.

LABORATORIUM 5. STL. KONTENERY ASOCJACYJNE.

Cel laboratorium:

Omówienie kontenerów asocjacyjnych.

Zakres tematyczny zajęć:

- biblioteka STL: kontenery, iteratory, algorytmy i funktory,
- kontenery asocjacyjne.

Pytania kontrolne:

1. Co to są kontenery asocjacyjne?
2. Jakie operacje można wykonywać na kontenerze *set*?
3. Z czego składa się element kontenera *map*?
4. Jaka jest różnica między kontenerem *set* a *multiset*?
5. Jaka jest różnica między kontenerem *map* a *multimap*?

Kontenery asocjacyjne:

Kontenery asocjacyjne umożliwiają dostęp do elementów kontenerów przez klucze. Zwykle kontenery asocjacyjne implementowane są w oparciu o drzewa czerwono – czarne dlatego też głównym zadaniem takich kontenerów jest wyszukiwanie na podstawie klucza. Do kontenerów asocjacyjnych należą:

- *set* – zbiór;
- *map* – mapa;
- *multiset* – wielozbiór;
- *multimap* – multimapa;
- *unordered set* – zbiór nieuporządkowany;
- *unordered multiset* – nieuporządkowany multizbiór;
- *unordered map* – nieuporządkowana mapa;
- *unordered multimap* – nieuporządkowana multimapa;

Set:

Set jest kontenerem realizującym zbiór w sensie matematycznym. Wartość przechowywana w zbiorze jest jednocześnie kluczem. W zbiorze nie mogą znajdować się dwa takie same elementy. Wartości przechowywane w zbiorze są uporządkowane, domyślnie rosnąco. Za pomocą funkcyjnego obiektu można do deklaracji kontenera dodać „własny sposób sortowania”. Dla zbioru nie został zdefiniowany operator `[]`. Na zbiorze można wykonywać takie operacje jak:

- suma zbiorów – zbiór zawierający elementy z obydwu zbiorów z wykluczeniem elementów, które się powtarzają;
- iloczyn zbiorów – zbiór zawierający elementy, które powtarzają się w obydwu zbiorach;
- różnica zbiorów – zbiór zawierający elementy, które występują w jednym zbiorze a w drugim nie występują;



Deklaracja zbioru wygląda tak samo jak deklaracja innych kontenerów. Należy pamiętać o dołączeniu biblioteki *set*. Dla kontenera *set* w bibliotece *STL* zostały zaimplementowane między innymi następujące metody:

- *insert* – dodanie elementu do zbioru;
- *begin* – zwrócenie iteratora dla pierwszego elementu;
- *end* – zwrócenie iteratora za ostatnim elementem;
- *erase* – usuwanie elementu ze zbioru;
- *find* – zwrócenie iteratora dla podanego elementu;
- *count* – sprawdzanie czy element jest w zbiorze.

W bibliotece *STL* można znaleźć również algorytmy, które są przeznaczone dla zbioru. Należą do nich:

- *set_union* – suma zbiorów;
- *set_intersection* – część wspólna zbiorów;
- *set_difference* – różnica zbiorów;

Na listingu 5.1. przedstawiony został plik *main.cpp*, który zawiera przykładową obsługę kontenera *set*. Listing 5.1. zawiera:

- linijka 2 – dołączenie biblioteki;
- linijki 4 – 12 – stworzenie funkcji *show*, która umożliwia wyświetlenie zawartości zbioru. W linijce 6 stworzony został iterator w taki sam sposób jak dla kontenerów sekwencyjnych;
- linijka 15 – stworzenie zbioru przechowującego liczby całkowite;
- linijki 16 – 20 dodawanie elementów do zbioru. W linijkach 18 oraz 20 wywołana została metoda *insert* z argumentem 3. Ponowna próba dodania 3 do zbioru nie powiodła się, ponieważ w zbiorze mogą występować tylko unikatowe wartości;
- linijka 22 – usunięcie elementów mniejszych od 2 ze zbioru. Domyślnie elementy w zbiorze są posortowane rosnąco. Metoda jako pierwszy argument przyjmuje iterator wskazujący na początek zbioru, a jako drugi argument funkcja przyjmuje iterator wskazujący za ostatnim elementem, który chcemy usunąć. Metoda *find* zwraca iterator elementu, którego wartość jest równa 2;
- linijka 24 – usunięcie elementu, którego wartość jest równa 2;
- linijka 25 – sprawdzenie czy element, którego wartość jest równa dwa znajduje się w zbiorze.

```
1  #include <iostream>
2  #include <set>
3  using namespace std;
4  void show( set <int> s)
5  {
6      set<int>::iterator itr;
7      for (itr = s.begin(); itr != s.end(); itr++)
8      {
9          cout << *itr<<" ";
10     }
11     cout<<endl;
12 }
13 int main()
14 {
```

```

15     set <int> s;
16     s.insert(1);
17     s.insert(2);
18     s.insert(3);
19     s.insert(7);
20     s.insert(3);

21     show(s);

22     s.erase(s.begin(), s.find(2));
23     show(s);
24     s.erase(2);
25     cout<<s.count(2)<<endl;

26     return 0;
27 }
```

Listing 5.1. Obsługa kontenera set

Na rysunku 5.1. przedstawiony został wynik działania kodu z listingu 5.1. Na początku wyświetlona została zawartość zbioru. Jak widać próba ponownego dodania elementu równego 3 się nie udała. W drugiej linijce wyświetlona jest zawartość zbioru po usunięciu elementów mniejszych od 2, w tym przypadku został usunięty element równy 1. Następnie w linijce 3 wyświetlona została zawartość po usunięciu elementu równego 2. W linijce 4 wyświetlone zostało 0 (*false*) i jest to wynik funkcji *count*, która sprawdzała czy element równy 2 znajduje się w zbiorze.

```

1 2 3 7
2 3 7
3 7
0
```

Rys. 5.1. Wynik działania kodu z listingu 5.1

Na listingu 5.2. zaprezentowane zostały algorytmy, które operują na zbiorach. Do pliku *main.cpp* należy dołączyć bibliotekę *algorithm*. Kod przedstawiony na listingu 5.2. należy dołączyć do funkcji *main*. Listing 5.2. zawiera:

- linijki 1 - 2 – zdefiniowanie dwóch zbiorów i wypełnienie ich wartościami;
- linijki 8 - 12 – wywołanie algorytmów realizujących: sumę, iloczyn oraz różnicę zbiorów. Każdy z tych algorytmów przyjmuje pięć argumentów. Pierwsze cztery argumenty opisują zakres zbiorów (cztery iteratory), a piąty argument to iterator wyjścia. Użyty został iterator *insert_iterator*, który pobiera dwa argumenty: nazwę kontenera w tym przypadku zbioru oraz iterator, który wskazuje pozycję dodania.

```

1  set <int> s1={1,2,3,7};
2  set<int>s2={-2,4,1,7,6};
3  cout<<"s1: ";
4  show(s1);
5  cout<<"s2: ";
6  show(s2);
7  set<int>sUnion,sIntersec,sDiffer;
```



```

8  set_union(s1.begin(), s1.end(), s2.begin(), s2.end(),
9           insert_iterator(sUnion, sUnion.begin()));

10 set_intersection(s1.begin(), s1.end(), s2.begin(), s2.end(),
11                 insert_iterator(sIntersec, sIntersec.begin()));
11 set_difference(s1.begin(), s1.end(), s2.begin(), s2.end(),
12               insert_iterator(sDiffer, sDiffer.begin()));
13 cout<<"sUnion: ";
14 show(sUnion);
15 cout<<"sIntersec: ";
16 show(sIntersec);
17 cout<<"sDiffer: ";
18 show(sDiffer);

```

Listing 5.2. Algorytmy działające na zbiorach

Na rysunku 5.2. przedstawiony został wynik działania kodu z listingu 5.2.

```

s1: 1 2 3 7
s2: -2 1 4 6 7
sUnion: -2 1 2 3 4 6 7
sIntersec: 1 7
sDiffer: 2 3

```

Rys. 5.2. Wynik działania kodu z listingu 5.2.

Przy pracy ze zbiorami przydatna może okazać się klasa *pair*, która traktuje dwie wartości jako pojedynczy element. Listing 5.3. pokazuje w jaki sposób zadeklarować obiekt typu *pair*. W ostrych nawiasach należy podać typy jakie będzie przechowywał obiekt. Klasa *pair* zawiera pola *first* oraz *second*, które umożliwiają dostęp do jej składników.

```
pair < type1, type2 > name
```

Listing 5.3. Deklaracja obiektu typu *pair*

Na listingu 5.4. przedstawiony został przykład użycia obiektu klasy *pair* w kontekście operacji na zbiorze. W przykładzie na listingu 5.4. metoda *insert* zwraca obiekt typu *pair*. Pierwsza składowa (*first*) obiektu typu *pair* zawiera iterator, pod którym znajduje się dodany element, a druga składowa (*second*) zawiera informację w postaci logicznej czy element udało się dodać do zbioru. Kod należy dodać do funkcji *main.cpp*.

```

1  set<int> s3={3,8};
2  pair <set<int>::iterator, bool> res;
3  res=s3.insert(6);
4  cout<<"Dodany element: "<<*(res.first)<<endl;
5  cout<<"Czy element dodany? "<<res.second<<endl;
6  show(s3);

```

Listing 5.4. Użycie metody *insert* i klasy *pair*

Na rysunku 5.3. przedstawiono wynik działania kodu z listingu 5.4.




```
Dodany element: 6
Czy element dodany? 1
3 6 8
```

Rys. 5.3. Wynik działania kodu z listingu 5.4.

Map:

Map jest kontenerem, który przechowuje parę (klucz, wartość) i mapuje klucz na wartość. Klucze muszą być unikatowe. Elementy w kontenerze przechowywane są w domyślnej postaci rosnąco względem kluczy. Za pomocą funkcyjnego obiektu można do deklaracji kontenera dodać „własny sposób sortownia”. Operacje wyszukiwania, wstawiania oraz usuwania wykonywane są na podstawie klucza. Sposób deklarowania kontenera *map* jest taki sam jak innych kontenerów. Aby korzystać z tego kontenera należy dołączyć bibliotekę *map*. Istnieje kilka sposobów tworzenia elementu kontenera *map*. Można to zrobić między innymi za pomocą klasy *pair*, funkcji *make_pair* oraz operatora []. Dla kontenera *map* zostały w bibliotece *STL* zaimplementowane podobne metody jak dla kontenera *set*.

Zaprezentowany przykład będzie dotyczył przechowywania informacji o pensji pracownika. Kluczem będzie nazwisko (zakładamy, że jest unikatowe), a wartością będzie pensja. Listing 5.5. przedstawia przykład obsługi kontenera *map*. Listing 5.5. zawiera:

- linijka 2 – dołączenie biblioteki *map*;
- linijki 4 – 13 – stworzenie funkcji *showMap*, która będzie wykorzystywana do wyświetlania kontenera *map*. Aby wyświetlić kontener należy stworzyć iterator. W celu pobrania wartości i klucza należy użyć pola *first* (dla klucza) oraz *second* (dla wartości);
- linijka 16 – stworzenie kontenera *map*, w którym klucz będzie typu *string* a wartość typu *int*;
- linijka 17 – pierwszy sposób dodawania do kontenera, za pomocą klasy *pair*;
- linijka 18 – drugi sposób dodawania do kontenera, za pomocą funkcji *make_pair*;
- linijka 19 – trzeci sposób dodawania do kontenera, za pomocą operatora [];
- linijka 21 – wyszukiwanie wartości na podstawie klucza;
- linijka 23 – usuwanie elementu w kontenerze na podstawie klucza;
- linijka 25 – sprawdzanie czy element o podanym kluczu znajduje się w kontenerze.

```
1  #include <iostream>
2  #include <map>
3  using namespace std;
4  void showMap( map<string, int> m)
5  {
6      cout<<endl<<"m: "<<endl;
7      map<string, int>::iterator it;
8      for(it=m.begin(); it!=m.end(); ++it)
9      {
10         cout << it->first <<" " <<it->second<<endl;
11     }
12     cout<<endl;
13 }
14 int main()
15 {
16     map<string,int> m;
```

```

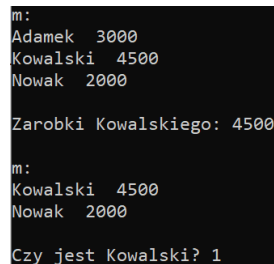
17  m.insert(pair<string,int>("Kowalski",4500));
18  m.insert(make_pair("Nowak",2000));
19  m["Adamek"]=3000;

20  showMap(m);
21  int salary=m.find("Kowalski")->second;
22  cout<<"Zarobki Kowalskiego: "<<salary<<endl;
23  m.erase("Adamek");
24  showMap(m);
25  cout<<"Czy jest Kowalski? "<<m.count("Kowalski")<<endl;
26  return 0;
26 }

```

Listing 5.5. Obsługa kontenera map

Na rysunku 5.4. przedstawiony został wynik działania kodu z listingu 5.5.



```

m:
Adamek 3000
Kowalski 4500
Nowak 2000

Zarobki Kowalskiego: 4500

m:
Kowalski 4500
Nowak 2000

Czy jest Kowalski? 1

```

Rys. 5.4. Wynik działania kodu z listingu 5.5

Elementem kontenera *map* jest element typu *pair*. Na listingu 5.6. przedstawiony został kod prezentujący w jaki sposób uzyskać dostęp do całego elementu przechowywanego w kontenerze *map*. Dla przykładu pobrany został pierwszy element z kontenera.

```

1  pair<string,int> el=*m.begin();
2  cout<<"Klucz: "<<el.first<<" " "<<"wart: "<<el.second<<endl;

```

Listing 5.6. Uzyskanie dostępu do elementu kontenera map

Jeśli chcemy wyszukiwać po wartości a nie po kluczu można zdefiniować następującą funkcję, która została przedstawiona na listingu 5.7. Na listingu 5.7. została zdefiniowana funkcja *searchByValue*, w której stworzony został iterator. Iterator przemieszcza się po kolejnych elementach szukając odpowiedniej wartości. Jeśli wartość została znaleziona, wtedy pętla jest przerywana i zwracany jest iterator wskazujący na element, którego wartość jest równa poszukiwanej wartości. Jeśli wartość nie została znaleziona wtedy zwracany jest iterator równy *m.end()*. Niezbędne jest dodanie referencji do argumentu *m*.

```

1  map<string, int>::iterator
2  searchByValue(map<string, int>& m, int val)
3  {
4      map<string, int>::iterator it;
5      for(it=m.begin();it!=m.end();it++)
6          if(it->second == val)

```

```
7         break;
8
9     return it;
10 }
```

Listing 5.7. Funkcja *searchByValue*

Teraz do funkcji *main* dodajmy wywołanie zaimplementowanej funkcji. Na listingu 5.8. przedstawione zostało wywołanie funkcji *searchByValue*. Jeśli element o podanej wartości został znaleziony, zostanie on wyświetlony. W przeciwnym wypadku zostanie wyświetlony komunikat „Brak elementu”.

```
1 map<string, int>::iterator it = serachByValue(m, 2000);
2 if(it != m.end())
3     cout<<"Element znaleziony: "<<it->first<<" "<<
4         it->second<<endl;
5 else
6     cout<<"Brak elementu"<<endl;
```

Listing 5.8. Wywołanie funkcji *searchByValue*

Multiset oraz multimap:

Kontener *multiset* działa podobnie jak kontener *set* z tą różnicą, że wartości kluczy mogą być zduplikowane. Natomiast kontener *multimap* działa podobnie do kontenera *map*, ale również z tą różnicą, że w kontenerze *multimap* wartości kluczy nie muszą być unikatowe.

Unordered set, unordered multiset, unordered map oraz unordered multimap:

Kontenery nieuporządkowane przechowują elementy w sposób nieuporządkowany, elementy nie są posortowane względem klucza. Zbudowane są w oparciu o technikę haszowania. Dzięki temu można uzyskać szybki dostęp do elementów.

Zadania do wykonania:

Zadanie 5.1. Duplikat

Napisz funkcję, która jako argument przyjmuje kontener *vector* przechowujący liczby całkowite. Zakładamy, że *vector* przechowuje unikatowe wartości oprócz jednej, która została zduplikowana. Zadaniem funkcji jest znalezienie wartości, która została zduplikowana oraz policzenie sumy unikatowych elementów. Funkcja powinna zwrócić obiekt klasy *pair*. W zadaniu należy wykorzystać kontener *set*. W funkcji *main* należy przetestować funkcję.

Zadanie 5.2. Alternatywa wykluczająca

Napisz funkcję szablonową, która jako argument przyjmuje dwa zbiory. Funkcja powinna wyświetlić elementy, które znajdują się tylko w pierwszym lub tylko w drugim zbiorze. Podpowiedź: Należy zwrócić uwagę z jakim typem wywoływane są algorytmy i iteratory.



Zadanie 5.3. Różnica

Napisz funkcję, która dostaje jako argumenty dwa napisy typu string. Drugi napis jest o jeden znak dłuższy od pierwszego. Drugi napis zawiera te same znaki co pierwszy napis tylko w różnej kolejności i dodatkowo jeszcze jeden znak. Zadaniem funkcji jest znalezienie znaku, który został dodany do drugiego napisu i zwrócenie go. W zadaniu należy wykorzystać kontener *map*. W *main* należy przetestować funkcję.

Zadanie 5.4. Słownik

Stwórz klasę *Dictionary*, która będzie reprezentować słownik i będzie posiadała następujące pola i metody:

- *words* – kontener *map*, który będzie przechowywać słowo i jego tłumaczenie. Zakładamy, że jedno słowo ma jedno tłumaczenie;
- konstruktor bezargumentowy;
- metodę umożliwiającą dodanie słowa wraz z tłumaczeniem do słownika. Przed dodaniem należy sprawdzić czy słowo już nie istnieje;
- metodę umożliwiającą usunięcie słowa wraz z jego tłumaczeniem ze słownika;
- metodę wyświetlającą zawartość słownika;
- metodę wyświetlającą tłumaczenie dla podanego słowa;
- metodę wyświetlającą zawartość słownika alfabetycznie od z do a względem tłumaczenia.

Podpowiedź: Aby posortować elementy w kontenerze *map* należy utworzyć pomocniczy kontener np. *vector*. Dodać do niego elementy i wykonać sortowanie używając funkcji *sort*.

W *main* należy stworzyć obiekt klasy *Dictionary* i przetestować działanie metod.

LABORATORIUM 6. PROGRAMOWANIE GENERYCZNE Z WYKORZYSTANIEM BIBLIOTEKI BOOST.

Cel laboratorium:

Omówienie elementów biblioteki *Boost*.

Zakres tematyczny zajęć:

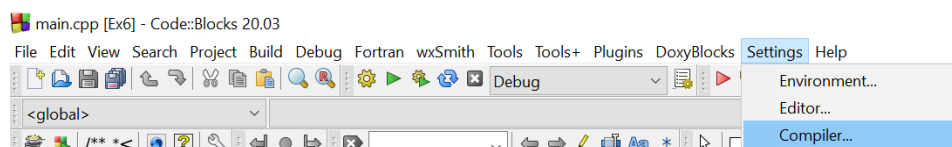
- dodawanie biblioteki *Boost* do projektu,
- elementy biblioteki *Boost*.

Pytania kontrolne:

1. Co zawiera biblioteka *Boost*?
2. Do czego służy *MultiIndex*?
3. Jakie są różnice pomiędzy interfejsami w *MultiIndex*?
4. Do czego służy *Fusion*?
5. Do czego służy *Bind*?

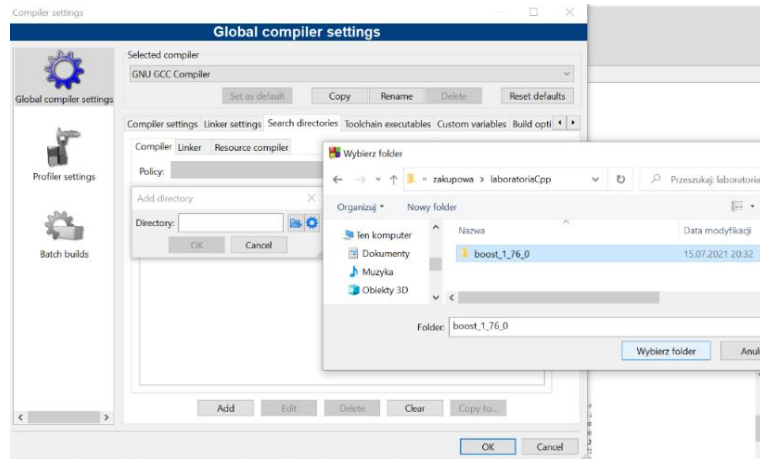
Biblioteka Boost:

Boost zawiera zbiór bibliotek przeznaczonych dla języka C++. Biblioteki te działają w połączeniu ze standardowymi bibliotekami C++. Biblioteka *Boost* zawiera różne moduły, które oparte są o szablony klas, między innymi do obsługi: inteligentnych wskaźników, wyrażeń regularnych, wielowątkowości, obliczeń równoległych. Zawiera również kontenery, iteratory i elementy programowania ogólnego. Biblioteka jest darmowa i należy ją pobrać z oficjalnej strony: <https://www.boost.org/users/download/>. Należy pobrać plik ze strony, rozpakować go i umieścić w dowolnym miejscu na dysku. Rysunki 6.1. – 6.3. pokazują jak dodać bibliotekę *Boost* do projektu w *Code::Blocks*. Pierwszy krok to wybranie z menu *Settings* a następnie *Compiler*.



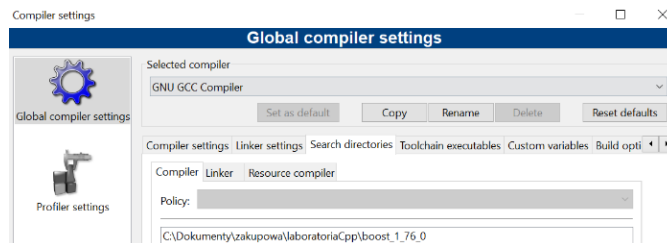
Rys. 6.1. Dodawanie biblioteki Boost – krok 1

Następnie w ustawieniach kompilatora należy wybrać *Search directories* i kliknąć przycisk *Add*, dodając folder z biblioteką *Boost*.



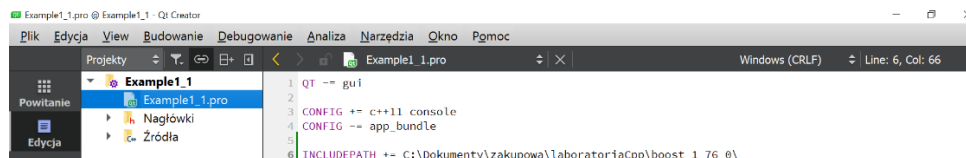
Rys. 6.2. Dodawanie biblioteki Boost – krok 2

Jeśli wszystko zostało poprawnie dodane powinna pojawić się ścieżka do folderu z biblioteką *Boost*.



Rys. 6.3. Dodawanie biblioteki Boost – krok 3

Na rysunku 6.4. przedstawiony został sposób dodania biblioteki *Boost* do projektu w *Qt*. W pliku z rozszerzeniem *pro* należy dodać ścieżkę, do folderu w którym znajduje się biblioteka za pomocą *INCLUDEPATH*. Nie trzeba dodawać folderu z biblioteką.



Rys. 6.4. Dodawanie biblioteki Boost

Jeśli katalog z biblioteką został dołączony już do projektu, należy dołączyć odpowiedni plik nagłówkowy biblioteki *Boost* za pomocą *#include* w zależności od tego, z którego modułu będziemy korzystać. Podczas laboratoriów omówione zostaną wybrane elementy biblioteki *Boost*: *Boost.MultiIndex*, *Boost.Fusion* oraz *Boost.Bind*.

Boost.MultiIndex:

MultiIndex umożliwia definiowanie kontenerów obsługujących dowolną liczbę interfejsów. Może być używany jeśli elementy muszą być dostępne na różne sposoby i w innym przypadku musiałyby być przechowywane w wielu kontenerach. Jako przykład można podać *vector* i *set* z biblioteki *STL*. *Vector* zapewnia interfejs, który obsługuje bezpośredni dostęp do elementów

o podanym indeksie, natomiast *set* zapewnia interfejs, w którym elementy są posortowane. Za pomocą *MultiIndex* można stworzyć kontener, który obsługuje te dwa interfejsy (*vector* oraz *set*), zamiast przechowywać elementy w dwóch kontenerach. *MultiIndex* zawiera następujące interfejsy:

- *hashed_non_unique*, który działa podobnie jak *unordered_multiset* w *STL*;
- *hashed_unique*, który działa podobnie jak *unordered_set* w *STL*;
- *ordered_non_unique*, który działa podobnie jak *multiset* w *STL*;
- *ordered_unique*, który działa podobnie jak *set* w *STL*;
- *sequenced*, który działa podobnie jak *list* w *STL*;
- *random_access*, działa podobnie jak *vector* w *STL*.

Biblioteka *MultiIndex* może przydać się do wyszukiwania wartości względem różnych indeksów, czyli względem różnych zmiennych. Będzie stworzony jeden kontener z różnymi interfejsami, które umożliwią różne wyszukiwania. Na przykład mamy kontener przechowujący informacje o osobach takie jak: imię, nazwisko, pesel oraz numer telefonu. Za pomocą *MultiIndex* można zrealizować wyszukanie osób o podanej informacji w jednym kontenerze, który dla każdego rodzaju wyszukiwania będzie miał zdefiniowany interfejs. Definicja kontenera *MultiIndex* została przedstawiona na listingu 6.1. W podanym przykładzie *typeOfElement* to nazwa struktury/klasy przechowującej informacje o osobie, *nameOfContainer* to nazwa kontenera dla którego tworzony jest interfejs, *typeOfIndex* to typ zmiennej, *nameOfIndex* to nazwa zmiennej na którą nakładany jest indeks, czyli według której ma być możliwe wyszukiwanie. Interfejsów może być zdefiniowanych kilka, na przykład tak jak w przykładzie powyżej wyszukiwanie może być po imieniu lub nazwisku.

```
1 multi_index_container< typeOfElement, indexed_by<
2 nameOfContainer< member< typeOfElement, typeOfIndex,
3 nameOfIndex > >, other interfaces >>
```

Listing 6.1. Tworzenie zmiennej typu *multi_index_container*

W celu korzystania z *MultiIndex* należy dołączyć plik nagłówkowy: *boost/multi_index_container.hpp*. Użycie interfejsu również wymaga dodania odpowiedniego pliku nagłówkowego: *boost/multi_index/nameOfInterface.hpp*. Zamiast *nameOfInterface* należy wstawić: *hashed_index*, *order_index*, *sequenced_index* lub *random_access_index*.

Za pomocą szablonowej metody *get* możliwe jest określenie, z którego interfejsu chcemy korzystać. Jeśli interfejs nie zostanie wybrany domyślnie, program będzie korzystać z pierwszego. W wybranym interfejsie mamy dostęp do wszystkich elementów, ale tylko do jednego nałożonego indeksu. Za pomocą *nth_index* możemy ustalić do jakiego interfejsu się odwołujemy. Na listingach 6.2 – 6.11. przedstawiony został przykład w jaki sposób posługiwać się kontenerem stworzonym za pomocą *MultiIndex*. Jako przykład zostanie zaprezentowany kontener przechowujący osoby. Każda osoba posiada imię oraz wiek. Będzie możliwe wyszukiwanie zarówno po imieniu jak i po wieku, ponieważ na te pola zostaną nałożone indeksy. Listing 6.2. przedstawia klasę *Person*. Klasa ta zawiera dwa pola: *name* oraz *age* i metodę *show* wyświetlającą informacje o osobie. Deklaracja klasy oraz definicja metody zostały umieszczone w jednym pliku nagłówkowym w celu dydaktycznym, aby skupić się na elementach biblioteki *Boost* a nie dołączaniu kolejnych plików. Metoda *show* musi być *const*, ponieważ wszystkie dane przechowywane w kontenerze *MultiIndex* są stałe.




```

1  using namespace std;
2  class Person
3  {   public:
4      string name;
5      int age;
6      void show () const;
7  };
8  void Person::show () const{
9      cout<<"name: "<<name<<" age: "<<age<<endl;
10 }
11 #endif // PERSON_H_INCLUDED

```

Listing 6.2. Klasa Person

Listing 6.3. zawiera przykład użycia *MultiIndex*. W kontenerze, który przechowuje osoby, założone są dwa indeksy: na imię oraz wiek, po których może odbywać się wyszukiwanie. Listing 6.3. zawiera:

- linijki 1 – 3 – dołączenie odpowiednich modułów;
- linijka 6 – dodanie przestrzeni nazw aby za każdym razem nie pisać *boost::multi_index*;
- linijki 8 - 11 stworzenie typu *person_multi* aby przy tworzeniu kontenera nie definiować wszystkiego od początku. Kontener będzie przechowywać obiekty klasy *Person* i będzie posiadać dwa interfejsy. Obydwa interfejsy będą reprezentować multizbiory nieuporządkowane (*multiset unordered*). Dla każdego interfejsu zdefiniowany został index, dzięki któremu będzie możliwe wyszukiwanie;
- linijki 12 - 13 – stworzenie typu dla pierwszego oraz drugiego interfejsu. Pierwszy interfejs odnosi się do imienia a drugi do wieku;
- linijka 16 – stworzenie kontenera;
- linijki 17 -21 – dodawanie kolejnych osób do kontenera;
- linijki 23 - 24– odwołanie się do domyślnego (pierwszego) interfejsu w kontenerze i policzenie ile osób ma na imię Ala;
- linijka 25-26 – odwołanie się do pierwszego interfejsu w kontenerze i policzenie ile osób ma na imię Ala. Metoda *get* umożliwia odwołanie się do dowolnego interfejsu w kontenerze;
- linijka 27 – zapisanie do zmiennej interfejsu wyszukującego po imieniu. Wykorzystany został typ *age_type* zdefiniowany w linijce 12;
- linijki 28 -29 – wykorzystanie interfejsu do wyszukania wszystkich osób, które mają 18 lat;
- linijki 30 – 32 – wyświetlenie wszystkich elementów kontenera za pomocą interfejsu związanego z imieniem (elementy ułożone względem imienia). Metody w klasie *Person* muszą być *const*, aby zapewnić, że żadna wartość indeksu w kontenerze nie zostanie zmieniona. Moduł *MultiIndex* udostępnia funkcje, za pomocą których jest możliwa modyfikacja danych, o tym będzie w dalszej części;
- linijka 33 – realizuje to samo co w linijce 27 tylko z użyciem typu *auto*. Dzięki wprowadzaniu *auto* nie trzeba się martwić o typy;
- linijki 34 – 36 – zastosowanie typu *auto* do zapisania iteratora zwracającego osobę o podanej liczbie lat.

```
1  #include <boost/multi_index_container.hpp>
2  #include <boost/multi_index/hashed_index.hpp>
3  #include <boost/multi_index/member.hpp>
4  #include <iostream>
5  #include "Person.h"
6  using namespace boost::multi_index;
7  using namespace std;

8  typedef multi_index_container<Person, indexed_by<
9  hashed_non_unique<member<Person, string, &Person::name>>,
10 hashed_non_unique<member<Person, int, &Person::age>>
11 >> person_multi;

12 typedef person_multi::nth_index<0>::type name_type;

13 typedef person_multi::nth_index<1>::type age_type;

14 int main()
15 {
16     person_multi persons;

17     persons.insert({"Ala", 40});
18     persons.insert({"Piotr", 10});
19     persons.insert({"Ola", 18});
20     persons.insert({"Ala", 46});
21     persons.insert({"Ula", 46});
22
23     cout<< "Liczba osob o imieniu Ala: "
24          <<persons.count("Ala")<< endl;
25     cout<<"Liczba osob o imieniu Ala: "
26          <<persons.get<0>().count("Ala")<<endl;

27     age_type &age_index = persons.get<1>();
28     cout<< "Liczba osob z wiekiem 18 lat: "
29          <<age_index.count(18)<< endl;

30     for(name_type::iterator it=persons.get<0>().begin();
31         it != persons.get<0>().end(); ++it)
32         it->show();

33     auto &age_indexx = persons.get<1>();
34     auto it = age_indexx.find(46);
35     cout<<"Znaleziona osoba, ktora ma 46 lat "
36          <<it->name<<endl;

37     return 0;
38 }
```

Listing 6.3. Przykład użycia MultiIndex

Na rysunku 6.5. przedstawiony został wynik działania kodu z listingu 6.3. W liniijkach od 3 do 8 wyświetlona została zawartość kontenera z interfejsem dedykowanym imieniu.

```
Liczba osob o imieniu Ala: 2
Liczba osob o imieniu Ala: 2
Liczba osob z wiekiem 18 lat: 1
name: Ala age: 46
name: Ala age: 40
name: Ula age: 46
name: Piotr age: 10
name: Ola age: 18
Znaleziona osoba, ktora ma 46 lat Ula
```

Rys. 6.5. Wynik działania kodu z listingu 6.3.

Zamieńmy linijki 30-32 z listingu 6.3. na linijki przedstawione w listingu 6.4.

```
1   for(age_type::iterator it=persons.get<1>().begin();
2       it != persons.get<1>().end(); ++it)
3       it->show();
```

Listing 6.4. Wyświetlenie zawartości kontenera względem drugiego interfejsu

```
name: Ala age: 40
name: Piotr age: 10
name: Ola age: 18
name: Ula age: 46
name: Ala age: 46
```

Rys. 6.6. Wynik działania kodu z listingu 6.4.

Jak można zauważyć, nie ważne za pomocą jakiego interfejsu, zawsze mamy dostęp do wszystkich elementów, tylko w różnej kolejności. Za pomocą funkcji *modify*, która jest w *MultiIndex*, można modyfikować element znajdujący się w kontenerze. Obiekt, który ma zostać zmodyfikowany jest wskazywany przez iterator jako pierwszy argument. Drugim argumentem funkcji jest funkcja lub obiekt funkcyjny. Na listingu 6.5. przedstawiona została funkcja *UlaToUrszula*, która modyfikuje imię obiektu klasy *Person* i będzie wykorzystana na kolejnym listingu.

```
1 void UlaToUrszula(Person& x)
2 {
3     x.name="Urszula";
4 }
```

Listing 6.5. Funkcja *UlaToUrszula*

Na listingu 6.6. przedstawione zostało użycie funkcji *modify*. Kod należy dodać do funkcji *main*. Listing 6.6. zawiera:

- linijka 1 - pobranie elementów z kontenera za pomocą pierwszego interfejsu;
- linijka 2 – zwrócenie iteratora przez funkcję *find*;
- linijka 3 – modyfikowanie elementu określonego przez iterator za pomocą metody *modify*. Funkcja *modify* jako drugi argument dostaje funkcję, funkcja jest uruchamiana za pomocą funkcji *bind* z biblioteki *Boost*. Funkcja *bind* zostanie opisana w dalszej części tych laboratoriów.

```

1 auto &name_indexx = persons.get<0>();
2 auto itt = name_indexx.find("Ula");
3 name_indexx.modify(itt, boost::bind(UlaToUrszula, _1));

```

Listing 6.6. Funkcja *modify*

Funkcja *modify* modyfikuje tylko jeden element. Na potrzeby kolejnego przykładu stworzona została funkcja *AlaToAlicja*, która modyfikuje imię jeśli dotychczasowe imię to *Ala*. Kod funkcji został przedstawiony na listingu 6.7.

```

1 void AlaToAlicja(Person& x)
2 {
3     if(x.name=="Ala")
4         x.name="Alicja";
5 }

```

Listing 6.7. Funkcja *AlaToAlicja*

Listing 6.8. zawiera kod, dzięki któremu można zmodyfikować wszystkie elementy. Kod należy dodać do funkcji *main*. Listing 6.8. zawiera:

- linijka 2 – stworzenie kontenera *vector*, który będzie zawierać iteratory elementów w kontenerze *persons*;
- linijka 7 – dodawanie iteratorów do *vektor*;
- linijki 9 – 11 – przejście po kontenerze *vector* i przekazanie każdego iteratora z kontenera *vector* do funkcji *modify*. Po każdym wywołaniu metody *modify*, kontener jest aktualizowany, stąd trzeba mieć wcześniej przechowane iteratory, bo później ich kolejność może być zmieniona.

```

1 cout << "Przed modyfikacja: "<<endl;
2 vector<name_type::iterator> elements;
3 for(name_type::iterator it=persons.get<0>().begin();
4     it != persons.get<0>().end(); ++it)
5 {
6     it->show();
7     elements.push_back(it);
8 }

9 for (int i = 0; i<elements.size();i++)
10     name_indexx.modify(elements[i],
11                         boost::bind(AlaToAlicja, _1));

12 cout << "Po modyfikacji"<<endl;
13 for(name_type::iterator it=persons.get<0>().begin();
14     it != persons.get<0>().end(); ++it)
15 {
16     it->show();
17 }

```

Listing 6.8. Modyfikacja wszystkich elementów

Na rysunku 6.7. zaprezentowany został wynik działania kodu z listingu 6.8. Jak można zaobserwować imię „Ala” zostało zmienione na „Alicja” i zmieniła się również kolejność elementów.

```
Przed modyfikacją:
name: Ala age: 46
name: Ala age: 40
name: Piotr age: 10
name: Ola age: 18
name: Urszula age: 46
Po modyfikacji
name: Piotr age: 10
name: Ola age: 18
name: Urszula age: 46
name: Alicja age: 40
name: Alicja age: 46
```

Rys. 6.7. Wynik działania kodu z listingu 6.8.

W celu zaprezentowania kolejnych możliwości *MultiIndex* stworzony zostanie kolejny typ dla kontenera *MultiIndex*, którego kod znajduje się na listingu 6.9. Tym razem kontener będzie przechowywał interfejs zbioru, gdzie elementy mogą się powtarzać (*multiset*), oraz interfejs umożliwiający dostęp, jak do kontenera *vector*. Należy pamiętać o dołączeniu plików nagłówkowych: *boost/multi_index/ordered_index.hpp* oraz *boost/multi_index/random_access_index.hpp*.

```
1 typedef multi_index_container<Person, indexed_by<
2   ordered_non_unique<member<Person, string, &Person::name>>,
3   ordered_non_unique<member<Person, int, &Person::age>>,
4   random_access<>>> person_multi_2;

5 typedef person_multi_2::nth_index<0>::type name_type2;
6
7 typedef person_multi_2::nth_index<1>::type age_type2;
```

Listing 6.9. Definicja kontenera *MultiIndex*

Elementy w zbiorze przechowywane są w uporządkowany sposób, więc bez problemu można wybrać elementy, które należą do zadanego zakresu. Listing 6.10. przedstawia dwa sposoby wyboru elementów z danego zakresu, kod należy dodać do funkcji *main*. Listing 6.10. zawiera:

- linijki 2 – 7 – dodawanie elementu do kontenera, dzięki interfejsowi *vector* możliwe jest użycie metody *push_back*;
- linijka 9 -10 – zastosowanie metody *equal_range*, która zwraca obiekt typu *pair*, gdzie element *first* zawiera iterator pierwszego wystąpienia a element *second* zawiera iterator za ostatnim wystąpieniem;
- linijki 16 – 17 – zastosowanie metod *lower_bound* oraz *upper_bound*, w celu znalezienia iteratora pierwszego wystąpienia oraz iteratora za ostatnim wystąpieniem.

```
1 person_multi_2 persons2;

2 persons2.get<2>().push_back({"Ala", 40});
3 persons2.get<2>().push_back({"Ala", 45});
4 persons2.get<2>().push_back({"Piotr", 10});
5 persons2.get<2>().push_back({"Ola", 18});
```

```
6  persons2.get<2>().push_back({"Aga", 46});
7  persons2.get<2>().push_back({"Ula", 46});

8  auto &name_indexx2 = persons2.get<0>();
9  auto iterStart = name_indexx2.equal_range("Ala");
10 auto iterStop = name_indexx2.equal_range("Piotr");

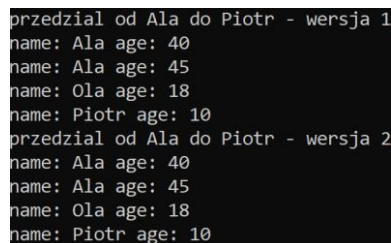
11 cout <<"przedzial od Ala do Piotr - wersja 1"<<endl;
12
13 for (name_type2::iterator it=iterStart.first;
14      it != iterStop.second; ++it)
15     it->show();

16 auto from = name_indexx2.lower_bound("Ala");
17 auto to = name_indexx2.upper_bound("Piotr");

18 cout <<"przedzial od Ala do Piotr - wersja 2"<<endl;
19 for (auto it=from; it != to; ++it)
20     it->show();
```

Listing 6.10. Wybór elementów z danego zakresu

Na rysunku 6.8. przedstawiony został wynik kodu z listingu 6.10. Obydwa zaprezentowane sposoby dają takie same wyniki.



```
przedzial od Ala do Piotr - wersja 1
name: Ala age: 40
name: Ala age: 45
name: Ola age: 18
name: Piotr age: 10
przedzial od Ala do Piotr - wersja 2
name: Ala age: 40
name: Ala age: 45
name: Ola age: 18
name: Piotr age: 10
```

Rys. 6.8. Wynik działania kodu z listingu 6.10.

Na listingu 6.11. przedstawiony został sposób odwoływania się do elementów kontenera, jak do kontenera *vector* za pomocą zaimplementowanego interfejsu *random_access*.

```
1  auto &rand_indexx = persons2.get<2>();
2  cout << rand_indexx[0].name <<endl;
```

Listing 6.11. Odwoływanie się do elementów kontenera

Boost.Fusion:

Fusion jest biblioteka, która umożliwia tworzenie heterogenicznych kontenerów. Na przykład, można utworzyć kontener typu *vector*, którego pierwszym elementem jest *int*, a drugim elementem jest łańcuch znaków. Dodatkowo dostarcza algorytmy do przetwarzania heterogenicznych kontenerów. Za pomocą standardowej biblioteki jest to niemożliwe, wyjątek stanowi typ *tuple*. Biblioteka zawiera takie kontenery jak: *tuple*, *vector*, *set*, *map* oraz *list*. W skład biblioteki wchodzi również funkcje do obsługi tych kontenerów. Aby używać

biblioteki *Fusion* należy dołączyć odpowiednie pliki nagłówkowe: *boost/fusion/container.hpp* oraz *boost/fusion/sequence.hpp*. Dodatkowo należy umieścić nagłówek *boost/mpl/int.hpp*, który potrzebny jest aby odwoływać się za pomocą indeksu do kontenerów.

Na listingu 6.12. przedstawiony został kod klasy *Print*, która zawiera przeciążony *operator()*, który będzie używany w przykładzie przedstawionym na listingu 6.13. Z uwagi na to, że klasa jest bardzo mała, kod z listingu 6.12. został umieszczony w pliku *main.cpp*.

```

1  class Print
2  {
3  public:
4      template <typename T>
5      void operator() (T t)
6      {
7          cout << t <<endl;
8      }
9  };

```

Listing 6.12. Klasa *Print*

Listing 6.13. przedstawia przykład działania kontenerów heterogenicznych na przykładzie kontenera *vector*. Kod należy dodać do funkcji *main*. Jeśli nie chcemy używać za każdym razem odwołania *boost::fusion*, można zadeklarować przestrzeń nazw, tak jak było to w poprzednich przykładach. Tutaj nie została dodana przestrzeń, ponieważ występowała już w przykładach *vector* ze standardowej biblioteki. Jeśli chcemy korzystać w kontenera *vector* z biblioteki *Fusion*, należy to zaznaczyć. Listing 6.13. zawiera:

- linijka 1 – stworzenie kontenera *vector* z wartościami początkowymi. Kontener *vector* przechowuje elementy różnego typu;
- linijka 4 – odwołanie się do elementu w kontenerze *vector* za pomocą funkcji *at*, która jest funkcją szablonową i przyjmuje parametr opakowany w *boost::mpl::int_*;
- linijka 5 – dodanie nowego elementu do kontenera *vector* za pomocą funkcji *push_back*, która zwraca nowy *vector*, nie modyfikując starego;
- linijka 6 – rozmiar kontenera *vector*;
- linijka 8 – pobranie pierwszego elementu;
- linijka 9 – pobranie ostatniego elementu;
- linijka 11 – wyświetlenie całego kontenera. W tym celu stworzona została klasa z zaimplementowaną metodą *operator()*. Za pomocą *for_each* wyświetlana jest zawartość kontenera. Do funkcji został przekazany obiekt funkcyjny, w którym zdefiniowany jest *operator()*.
- linijka 13 – pobranie iteratora pierwszego elementu;
- linijka 14 - pobranie iteratora ostatniego elementu;
- linijka 18 – pobranie iteratora wskazanego elementu.

```

1  boost::fusion::vector<int, string, bool, double> vec{10,
2              "C++", true, 3.14};
3  cout << "Trzeci element w vec:"
4      <<boost::fusion::at<boost::mpl::int_<2>>(vec) <<endl;
5  auto vec2 = push_back(vec, 'M');

```



```

6  cout <<"Liczba elementow w wvec: " << size(vec) <<endl;
7  cout << "Liczba elementow w vec2: " <<size(vec2) <<endl;
8  cout << "Pierwszy element w vec2: " <<front(vec2) <<endl;
9  cout << "Ostatni element w vec2: " <<back(vec2) <<endl;

10 cout<<"Cala zawartosc vec2: " <<endl;
11 boost::fusion::for_each(vec2, Print()) ;
12
13 auto itvStart = begin(vec) ;
14 auto itvStop = end(vec) ;
15 cout<<"Pierwszy element: " <<*itvStart<<endl;
16 cout<<"Drugi element: " <<*next(itvStart)<<endl;
17 cout<<"Trzeci element: "
18    <<*advance<boost::mpl::int_<2>>(itvStart)<<endl;

```

Listing 6.13. Przykład działania biblioteki Fusion

Na rysunku 6.9. przedstawiony został wynik działania kodu z listingu 6.13.

```

Trzeci element w vec:1
Liczba elementow w wvec: 4
Liczba elementow w vec2: 5
Pierwszy element w vec2: 10
Ostatni element w vec2: M
Cala zawartosc vec2:
10
C++
1
3.14
M
Pierwszy element: 10
Drugi element: C++
Trzeci element: 1

```

Rys. 6.9. Wynik działania kodu z listingu 6.13.

Boost.Bind:

Bind i mieszcząca się w tym module funkcja *bind* jest uogólnieniem funkcji w standardowej bibliotece: *bind1st* oraz *bind2st*. W standardzie języka C++ w standardowej bibliotece również pojawiła się funkcja *bind*, która została przeniesiona z biblioteki *Boost* od standardu C++11. Wspiera funkcje oraz obiekty funkcyjne. W celu korzystania w funkcji *bind* należy dołączyć plik nagłówkowy: *boost/bind.hpp*.

Na listingu 6.14. przedstawione zostały przykłady użycia funkcji *bind*. Funkcja *bind* daje możliwość zastępowania argumentów lub kolejności przekazywanych argumentów. Realizowane jest to poprzez symbole *_1*, *_2* aż do *_9* (łącznie jest 9 takich symboli). Symbole zastępują argumenty wejściowe. Jeśli używany jest funktor, który ma więcej niż jedną przeciążoną metodę operator(), wtedy wywołując funkcję *bind* należy w ostrych nawiasach podać typ. Kod przedstawiony na poniższym listingu powinien znaleźć się w pliku *main.cpp*. Listing 6.14. zawiera:

- linijki 5–32 stworzenie funkcji lub funktorów, które będą przekazane do funkcji *bind*;
- linijka 36 – stworzenie kontenera *vector*;
- linijka 39 - użycie funkcji *bind* w celu przekazania argumentu do funkcji *printF*, której elementem jest *vector*;

- linijka 41 – użycie funkcji *bind* w celu przekazania argumentów do funkcji *firstArgMod*. Pierwszy argument to element kontenera a drugi to stała 5;
- linijki 43 – 46 – użycie funkcji *bind* w celu przekazania argumentów do zdefiniowanych funktorów. Pierwszym elementem jest element kontenera a drugi stałą. Funkcja *count_if* zwróci liczbę elementów z przedziału (8,10>;
- linijka 49 – wywołanie funkcji *addition*, która zwraca sumę dwóch elementów;
- linijka 50 – wywołanie tej samej funkcji co w linijce 49 ale za pomocą funkcji *bind*, która jako pierwszy argument przyjmuje 1 a jako drugi 2;
- linijka 54 – wywołanie funkcji *contatination*, za pomocą funkcji *bind*, która jako pierwszy i drugi argument otrzymuje zmienne;
- linijka 56 – stworzenie funktora. Klasa *Operation* zawiera dwa operatory(). Jeden operator() przyjmuje dwa całkowite argumenty i oblicza ich różnicę, a drugi przyjmuje jeden znakowy argument i zwraca dużą literę alfabetu dla podanego znaku;
- linijka 58 – wywołanie funktora z określonym typem – *int* za pomocą funkcji *bind*. Funktor jako argumenty dostaje tą samą wartość zapisaną pod zmienną *x*
- linijka 60 – wywołanie funktora z określonym typem – *char* za pomocą funkcji *bind*.

```
1  include <boost/bind.hpp>
2  #include <iostream>
3  #include<vector>
4  using namespace std;

5  void firstArgMod(int& x, int y)
6  {
7      x=x+y;
8  }
9  void printF(int x)
10 {
11     cout<<x<<" ";
12 }
13 void contatination(int i1,int i2)
14 {
15     cout << i1 << i2<< endl;;
16 }
17 class Operation
18 {
19 public:
20     int operator()(int a, int b)
21     {
22         return a - b;
23     }
24     int operator()(char a)
25     {
26         return (int)a-32;
27     }
28 };
29 int addition(int a, int b)
30 {
```



```

31     return a + b;
32 }
33 int main()
34 {
35     cout<<"vec: "<<endl;
36     vector<int> vec= {1,2,3,4,5};
37     vector<int>::iterator iter1 = vec.begin();
38     vector<int>::iterator iter2 = vec.end();
39     for_each(iter1, iter2, boost::bind(printf, _1));

40     cout<<endl<<"vec - kazdy element zwiekszony o 5: ";
41     for_each(iter1, iter2, boost::bind(firstArgMod, _1, 5));
42     for_each(iter1, iter2, boost::bind(printf, _1));

43     int count=std::count_if(vec.begin(), vec.end(),
44                             boost::bind(std::logical_and<bool>(),
45                             boost::bind(std::greater<int>(),_1,8),
46                             boost::bind(std::less_equal<int>(),_1,10)));

47     cout << endl<<"vec - 8<Elementy<=10  "<<count <<endl;
48
49     cout<<addition(1,2)<<endl;
50     int res=boost::bind(addition,_1,_2) (1,2);
51     cout<<res<<endl;
52
53     int i1=1,i2=2;
54     boost::bind(contatination,_2,_1) (i1,i2);
55
56     Operation ff;
57     int x = 1024;
58     cout<<"Funktor: "<<bind<int>(ff, _1, _1) (x)<<endl;
59     char y='a';
60     cout<<"Funktor: "<<bind<char>(ff, _1) (y)<<endl;
61     return 0;
62 }

```

Listing 6.14. Przykład działania funkcji bind

Na rysunku 6.10. przedstawiony został wynik działania kodu z listingu 6.14.

```

vec:
1 2 3 4 5
vec - kazdy element zwiekszony o 5:
6 7 8 9 10
vec - 8<Elementy<=10  2
3
3
21
Funktor: 0
Funktor: A

```

Rys. 6.10. Wynik działania kodu z listingu 6.14.

Zadania do wykonania:

Zadanie 6.1. Przykładowy kod

Uruchom przykłady przedstawione podczas tego laboratorium.

Zadanie 6.2. Książka teleadresowa

Książka telefoniczna powinna zostać zaimplementowana w oparciu o *MultiIndex* z biblioteki *Boost*. Należy stworzyć klasę *Contact*, która będzie przechowywała informację o jednym kontakcie telefonicznym. Informacje, które powinny być przechowywane to: imię i nazwisko, wiek, numer telefonu – wartość unikatowa oraz ulica.

Następnie należy stworzyć klasę *Contacts*, która będzie umożliwiała:

- przechowywanie książki teleadresowej;
- dodanie nowego kontaktu do książki teleadresowej. Jeśli podany numer telefonu znajduje się w książce, nie należy go dodawać;
- usunięcie kontaktu z książki o podanym numerze telefonu;
- wyszukanie wszystkich osób mieszkających na tej samej ulicy;
- wyszukanie osób z podanego przedziału wiekowego;
- wyszukanie osoby po numerze telefonu;
- zmiana nazwy ulicy dla wszystkich osób mieszkających przy danej ulicy;
- policzenie ile osób w książce telefonicznej, które są pełnoletnie;
- policzenie ile unikatowych nazwisk znajduje się w książce;
- wyświetlenie zawartości książki teleadresowej.

W funkcji *main* należy przetestować stworzone klasy.

Zadanie 6.3. Statystyka

Napisz funkcję szablonową o parametrze *T*, która jako argument będzie zawierać kontener *vector* typu *T*. Zadaniem funkcji jest wyświetlenie:

- elementów mniejszych niż średnia arytmetyczna wszystkich elementów;
- elementów znajdujących się między średnią arytmetyczną a medianą wszystkich elementów;
- elementów dodatnich.

Można skorzystać z elementów biblioteki *functional*. Wykaz operatorów można znaleźć na stronie <http://www.cplusplus.com/reference/functional/> W celu przekazania argumentów do funkcji należy użyć funkcji *bind*.

Zadanie 6.4. Mix

Napisz funkcję, która dostanie jako argument *vector* z biblioteki *Fusion*. O kontenerze wiadomo, że może przechowywać następujące typy: *int*, *double*, *float*, *bool* oraz *char*. Funkcja powinna zwrócić informację ile razy występuje dany typ w kontenerze w postaci mapy, której element będzie stanowić parę <typ, liczba wystąpień tego typu>.

W celu sprawdzenia jaki typ danych jest przechowywany można wykorzystać funkcję *typeid(variableName).name()*.

LABORATORIUM 7. PROGRAMOWANIE GENERYCZNE Z WYKORZYSTANIEM BIBLIOTEKI QT.

Cel laboratorium:

Omówienie elementów biblioteki *Qt*.

Zakres tematyczny zajęć:

- kontenery biblioteki *Qt*,
- algorytmy biblioteki *Qt*.

Pytania kontrolne:

1. Jakie kontenery zawiera biblioteka *Qt*?
2. W jaki sposób można odwołać się do elementów w kontenerze *QVector*?
3. Do czego używana jest metoda *hasNext* w iteratorze?
4. Jaka jest różnica między iteratorami z *STL* a nowym iteratorem w *Qt*?
5. Jakie algorytmy znajdują się w bibliotece *QtAlgorithms*?

Biblioteka Qt:

Qt zawiera zbiór bibliotek przeznaczonych do programowania w C++. Środowisko programistyczne, które posiada wbudowaną bibliotekę *Qt* to *Qt Creator* w związku z tym wszystkie przykłady będą wykonane za pomocą tego środowiska. Biblioteka *Qt* umożliwia między innymi realizację programowania ogólnego, które oparte jest o klasy kontenerowe. W standardowej bibliotece znajdują się klasy kontenerowe *STL*, a w *Qt* znajdują się klasy kontenerowe *QTL*. Funkcje do obsługi kontenerów *Qt* działają podobnie jak te ze standardowej biblioteki. Kontenery *Qt* są zoptymalizowane pod względem pamięciowym. Posiadają takie same iteratory jak dla *STL* oraz nowy typ iteratora, który zostanie przedstawiony w ramach laboratoriów. W *Qt* znajdują się zarówno kontenery sekwencyjne jak i asocjacyjne. Do kontenerów sekwencyjnych należą: *QVector*, *QList*, *QLinkedList*, *QQueue* (adapter *QList*) oraz *QStack* (adapter *QVector*), natomiast do asocjacyjnych należą: *QMap*, *QMultiMap*, *QHash*, *QMultiHash* oraz *QSet*. Każdy z wymienionych kontenerów działa podobnie jak jego odpowiednik w bibliotece *STL*. Biblioteka *Qt* posiada szeroką dokumentację, która umieszczona jest na stronie internetowej: <https://doc.qt.io/qt-5/qtcore-module.html> Jako przykład kontenera sekwencyjnego przedstawiony zostanie kontener *QVector*, a jako przykład kontenera asocjacyjnego zaprezentowany zostanie kontener *QMap*. *QHash* oraz *QMap* działają podobnie, różnica jest tylko w szybkości działania niektórych metod.

QVector:

QVector jest klasą szablonową implementującą dynamiczną tablicę. Odpowiada klasie *Vector* z biblioteki *STL*. Należy dołączyć plik nagłówkowy: *QVector*, aby korzystać z *QVector*. Na stronie internetowej <https://doc.qt.io/qt-5/qvector.html> można znaleźć wykaz metod z klasy *QVector*, która zawiera więcej metod niż *vector* z *STL*. Listing 7.1. prezentuje przykład obsługi kontenera *QVector*. Zaprezentowane metody mogą występować z różnymi argumentami. Listing 7.1. zawiera:

- linijka 7 – 14 – funkcja szablonowa wyświetlająca zawartość kontenera *QVector*. Prezentuje jeden ze sposobów dostępu do elementu kontenera *QVector*;
- linijka 18 – stworzenie kontenera *QVector*, który będzie przechowywał liczby całkowite. Na początku kontener wypełniony jest 0;
- linijki 19 -20 – dodanie/nadpisanie elementu;
- linijka 21 – dodanie elementu na koniec;
- linijka 22 – dodanie elementu na koniec;
- linijka 23 – dodanie elementu na koniec;
- linijka 24 – 26 – pierwszy sposób dostępu do elementów kontenera, za pomocą [];
- linijki 27 – 29 – drugi sposób dostępu do elementów kontenera, za pomocą metody *at*;
- linijki 30 -33 – trzeci sposób dostępu do elementów kontenera, za pomocą iteratora. Tego iteratora nie ma w bibliotece *STL*. Iterator posiada metodę *hasNext*, która zwraca *true* jeśli są jeszcze elementy w kontenerze. Za pomocą metody *next* pobierana jest wartość elementu;
- linijka 35 – wywołanie funkcji *show*, która przedstawia czwarty sposób dostępu do elementów. Ten rodzaj iteratora występuje również w *STL*;
- linijki 36 – 42 – modyfikacja wartości przechowywanych w kontenerze za pomocą iteratora;
- linijka 44 – metoda *insert*, która jako argument przyjmuje pozycję na którą trzeba wstawić element podany jako drugi argument;
- linijka 45 – metoda *insert*, która jako argument dostaje iterator za który należy wstawić element podany jako drugi argument;
- linijka 48 – metoda *lastIndexOf*, która szuka ostatniego wystąpienia wartości przekazanej w argumencie. Jeśli wartości nie ma w kontenerze, metoda zwróci -1;
- linijka 49 – metoda *move*, przenosi wartość spod pozycji przekazanej w argumencie 1 na pozycję przekazaną w argumencie 2;
- linijka 50 – metoda *remove* usuwa element spod wskazanej pozycji.

```
1  #include <boost/bind.hpp>
2  #include <QCoreApplication>
3  #include <iostream>
4  #include <QVector>
5  using namespace std;
6  template<typename T>
7  void show(QVector<T> vec)
8  {
9      cout<<"Zawartosc vec: :";
10     typename QVector<T>::iterator p;
11     for (p = vec.begin(); p != vec.end(); ++p)
12         cout << *p<<" ";
13
14     cout<<endl;
15 }
16 int main(int argc, char *argv[])
17 {
18     QCoreApplication a(argc, argv);
```



```
18  QVector<int> vec(6);
19  vec[0]=1;
20  vec[1]=2;

21  vec.append(2);
22  vec.push_back(4);

23  vec << 5 << 6 ;

24  cout<<"Zawartosc vec: :";
25  for(int i=0; i <vec.count(); ++i)
26      cout<<vec[i]<<" ";

27  cout<<endl<<"Zawartosc vec: :";
28  for(int i =0; i<vec.size(); ++i)
29      cout<< vec.at(i) << " ";

30  cout<<endl<<"Zawartosc vec: :";
31  QVectorIterator<int> i(vec);
32  while(i.hasNext())
33      cout << i.next()<<" ";

34  cout<<endl;
35  show(vec);

36  cout<<"Zawartosc vec: :";
37  QVector<int>::iterator p;
38  for (p = vec.begin(); p != vec.end(); ++p)
39  {
40      *p += 2;
41      cout<<*p<<" ";
42  }

43  cout<<endl;
44  vec.insert(1,666);
45  vec.insert(vec.begin(),777);
46  show(vec);

47  cout<< "Ostatni indeks gdzie 2: "
48      <<vec.lastIndexOf(2)<<endl;
49  vec.move(1,vec.length()-1);
50  vec.remove(3);
51  show(vec);

52  return a.exec();
53 }
```

Listing 7.1. Przykład działania kontenera *QVector*

Na rysunku 7.1. przedstawiony został wynik działania kodu z listingu 7.1.




```

Zawartosc vec: :1 2 0 0 0 0 2 4 5 6
Zawartosc vec: :1 2 0 0 0 0 2 4 5 6
Zawartosc vec: :1 2 0 0 0 0 2 4 5 6
Zawartosc vec: :1 2 0 0 0 0 2 4 5 6
Zawartosc vec: :3 4 2 2 2 2 4 6 7 8
Zawartosc vec: :777 3 666 4 2 2 2 2 4 6 7 8
Ostatni indeks gdzie 2: 7
Zawartosc vec: :777 666 4 2 2 2 4 6 7 8 3

```

Rys. 7.1. Wynik działania kodu z listingu 7.1.

QMap:

QMap jest klasą szablonową implementującą słownik w oparciu o drzewo czerwono - czarne. Odpowiada klasie *map* z biblioteki *STL*. Bardzo podobną klasą do *QMap* jest *QHash*. Aby korzystać z *QMap* należy dołączyć plik nagłówkowy *QMap*. Na stronie internetowej <https://doc.qt.io/qt-5/qmap.html> można znaleźć wykaz metod z klasy *QMap*, która zawiera więcej metod niż *map* z *STL*. Na listingu 7.2 zostały przedstawione niektóre z nich. Kod należy dodać do funkcji *main*. W przykładzie używany jest kontener *QList*, który wymaga dołączenia pliku *QList*. Zaprezentowane metody mogą występować z różnymi argumentami. Listing 7.2. zawiera:

- linijka 1 – stworzenie mapy, która przechowuje parę klucz typu *int* oraz wartość typu *QString*. *QString* jest odpowiednikiem *String* ze standardowej biblioteki;
- linijka 2 – pierwszy sposób dodania elementu do kontenera;
- linijka 4 – drugi sposób dodania elementu do kontenera;
- linijki 5 - 11 – stworzenie iteratora dla kontenera oraz wyświetlenie wszystkich elementów. Metoda *next()* przechodzi do kolejnego elementu. Aby odwołać się do klucza należy użyć metody *key*, a do wartości metody *value*. W prezentowanym przykładzie wartość jest typu *QString* więc trzeba użyć metody *toStdString* aby móc wyświetlić za pomocą *cout* wartość na konsolę;
- linijka 12 – pobranie wszystkich kluczy. Metoda *keys* zwraca listę;
- linijka 13 – pobranie wszystkich wartości;
- linijki 19-20 – użycie metody *equal_range*, która zwraca parę iteratorów. Pierwszy iterator zawiera pierwsze wystąpienie elementu o podanym kluczu a drugi zawiera adres elementu za ostatnim wystąpieniem. W przypadku kiedy wartości kluczy są unikatowe, w kontenerze może wystąpić tylko jeden element o podanej wartości klucza;
- linijka 21 – wyświetlenie wartości dla podanego klucza.

```

1  QMap<int,QString> shops;
2  shops[111]= "CCC";
3  shops[222]= "Decathlon";
4  shops.insert(333,"Reserved");

5  QMapIterator<int,QString> j(shops);
6  while(j.hasNext())
7  {
8      j.next();
9      cout<<"Id: "<<j.key()<<" sklep:  "
10         << j.value().toStdString()<<endl;
11  }

12  QList<int> keys=shops.keys();

```



```

13  QList<QString> values=shops.values();
14  cout<<"Klucze: "<<endl;
15  for(int i=0;i<keys.size();i++){
16      cout<<keys[i]<<" ";
17  }
18  cout<<endl;
19  QMap<int,QString>::iterator, QMap<int,QString>::
20      iterator> range=shops.equal_range(111);
21  cout<<"Wartosc, dla klucza 111: "
22      <<range.first->toString()<<endl;

```

Listing 7.2. Przykład działania kontenera *QMap*

Na rysunku 7.2. przedstawiony został wynik działania kodu z listingu 7.2.

```

Id: 111 sklep: CCC
Id: 222 sklep: Decathlon
Id: 333 sklep: Reserved
Klucze:
111 222 333
Wartosc, dla klucza 111: CCC

```

Rys. 7.2. Wynik działania kodu z listingu 7.5.

Iteratory:

Biblioteka *Qt* zawiera takie same iteratory jak te które są w bibliotece *STL* oraz dodatkowo nowy typ iteratora. Iterator ten działa w inny sposób niż pozostałe. Iterator wskazuje na miejsce pomiędzy dwoma elementami a nie na element. Kontener przekazywany jest do iteratora a następnie za pomocą metody *hasNext/hasPrevious* odbywa się sprawdzenie czy za tą pozycją znajduje się jeszcze element. Jeśli jest jeszcze jakiś element wtedy za pomocą metody *next/previous* iterator przechodzi za ten element i zwraca wartość „przez którą przeszedł”. Do pracy z iteratorami mogą być przydatne następujące funkcje:

- *toFront* – iterator przesuwany jest na początek;
- *toBack* – iterator przesuwany jest na koniec.

Iterator, który pozwala na modyfikację, usunięcie elementu w kontenerze musi mieć specyfikator *Mutable*. Listing 7.3 przedstawia przykład użycia nowego iteratora oraz porównanie iteratora *QVectorIterator* i *QMutableVectorIterator*. W przykładzie użyty został kontener *QVector* z poprzednich przykładów. Kod należy dodać do funkcji *main*. Listing 7.3. zawiera:

- linijka 2 – stworzenie iteratora;
- linijki 3 – 5 – wyświetlenie wszystkich elementów w kontenerze;
- linijka 6 – stworzenie iteratora *Mutable*;
- linijki 7 – 11 – przejście po elementach kontenera i zmiana ich wartości na -1.

```

1  cout<<endl<<"Zawartosc vec: ";
2  QVectorIterator<int> it(vec);
3  while(it.hasNext())
4      cout << it.next()<<" ";
5  cout<<endl;

```

```

6  QMutableVectorIterator<int> itM(vec);
7  while(itM.hasNext())
8  {
9      itM.next();
10     itM.setValue(-1);
11 }

12 itM.toFront();
13 cout<<endl<<"Zawartosc vec: ";
14 while(itM.hasNext())
15     cout << itM.next()<<" ";

```

Listing 7.3. Przykład użycia iteratora

Rysunek 7.3. przedstawia wynik działania kodu z listingu 7.3.

```

Zawartosc vec: 777 666 4 2 2 2 4 6 7 8 3
Zawartosc vec: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

```

Rys. 7.3. Wynik działania kodu z listingu 7.3.

Na stronie <https://doc.qt.io/qt-5/qmutablevectoriterator.html> zaprezentowane zostały wszystkie funkcjonalności iteratora *Mutable* dla *QVector*.

Algorytmy:

Dla kontenerów z biblioteki *Qt* można używać algorytmy ze standardowej biblioteki *STL*. Dodatkowo biblioteka *Qt* zawiera bibliotekę *QtAlgorithms*, w której zdefiniowane są algorytmy. Na stronie <https://doc.qt.io/qt-5/qtalgorithms-obsolete.html> zamieszczone są wszystkie dostępne algorytmy w *Qt*. Listing 7.4. przedstawia klasę *Compare*, która będzie wykorzystana na następnym listingu. Klasa posiada przeciążony *operator()*.

```

1  class Compare{
2  public:
3      bool operator()(int s1, int s2){
4          if(s1%2>s2%2)
5              return true;
6          else
7              return false;
8      }
9  };

```

Listing 7.4. Klasa *Compare*

Na listingu 7.5. zaprezentowane zostały niektóre z algorytmów. Jeśli kod z listingu 7.5. będzie dołączany do poprzednich przykładów należy pamiętać, aby zakomentować modyfikację elementu w kontenerze *QVector*, ponieważ wtedy będą się w nim znajdowały tylko -1. Listing 7.5. zawiera:



- linijka 3 – algorytm *qCount* zlicza ile razy podana wartość występuje w kontenerze. W czwartym argumencie przekazywana jest zmienna, do której zapisany będzie wynik zliczania;
- linijka 6 – algorytm *qFind* zwraca iterator znalezionego elementu. Jeśli element nie został znaleziony, zwrócony zostanie iterator odnoszący się do elementu za ostatnim elementem (*end*);
- linijka 10 – algorytm *qSort*, który jako trzeci argument posiada gotowy funktor – sortowanie malejąco;
- linijka 13 – algorytm *qSort*, który jako trzeci argument posiada zdefiniowany przez użytkownika funktor. Sortowanie odbywa się w następujący sposób: najpierw będą liczby nieparzyste a następnie parzyste.

```
1 show(vec);
2 int howMany=0;
3 qCount(vec.begin(),vec.end(),2,howMany);
4 cout<<"Liczba wystapien 2: "<<howMany<<endl;

5 QVector<int>::iterator it2=qFind(vec.begin(),
6                                 vec.end(),-2);
7 if(it2==vec.end())
8     cout<<"Element nie zostal znaleziony"<<endl;

9 qSort(vec.begin(), vec.end(), qGreater<int>());
10 show(vec);

11 qSort(vec.begin(), vec.end(), Compare());
12 show(vec);
```

Listing 7.5. Przykład działania algorytmów z QtAlgorithms

Na rysunku 7.4. został przedstawiony wynik działania kodu z listingu 7.5.

```
Zawartosc vec: :777 666 4 2 2 2 4 6 7 8 3
Liczba wystapien 2: 3
Element nie zostal znaleziony
Zawartosc vec: :777 666 8 7 6 4 4 3 2 2 2
Zawartosc vec: :3 7 777 2 2 4 4 2 666 6 8
```

Rys. 7.4. Wynik działania kodu z listingu 7.5.

Zadania do wykonania:

Zadanie 7.1. Katalog samochodów

Napisz aplikację okienkową, która będzie obsługiwała katalog samochodów i umożliwi:

- przechowywanie informacji o samochodzie:
 - marka i model samochodu,
 - rok produkcji,
 - numer VIN.
- dodanie nowego samochodu do katalogu;
- wyświetlenie samochodów z katalogu;

- policzenie samochodów starszych niż podany rok produkcji;
- usunięcie samochodu o podanym numerze VIN;
- wyświetlenie samochodów z katalogu w następującym porządku:
 - malejąco względem roku produkcji
 - alfabetycznym od a do z względem marki samochodu

Do przechowywania samochodów należy wykorzystać kontener z biblioteki *Qt*.

Zadanie 7.2. Pangram

Napisz funkcję, która jako argument dostaje napis. Zadaniem funkcji jest sprawdzenie czy napis jest pangramem. Pangram to napis, w którym występują wszystkie litery alfabetu (wielkość liter nie ma znaczenia). W zadaniu bierzemy pod uwagę tylko alfabet łaciński. Przykład pangramu to: The quick brown fox jumps over the lazy dog. W zadaniu należy wykorzystać kontener *QSet*. W funkcji *main* należy przetestować stworzoną funkcję.

Zadanie 7.3. Kraje

Napisz aplikację okienkową, która będzie obsługiwała informację o kraju oraz liczbie ludności. Aplikacja ma umożliwiać:

- dodanie nowego kraju wraz z liczbą ludności;
- wyświetlenie krajów wraz z liczbą ludności;
- wyświetlenie samych krajów;
- wyświetlenie krajów z podanego zakresu;
- usunięcie kraju o podanej nazwie;
- wyświetlenie krajów wraz z liczbą mieszkańców rosnąco lub malejąco względem liczby mieszkańców.

Podpowiedź: Do sortowania można użyć *QVector*, którego element będzie miał typ *QPair*.

Do przechowywania informacji o kraju należy wykorzystać *QMap* lub *QHash*.

LABORATORIUM 8. KOŁOKWIUM 1.

Cel laboratorium:

Weryfikacja nabytych umiejętności dotyczących polimorfizmu statycznego oraz dynamicznego, szablonów funkcji i klas oraz bibliotek *STL*, *Boost* oraz *Qt*.

Wytyczne do kolokwium 1:

- zakres laboratoriów 1-7,
- próg zaliczeniowy 51%,
- pozostałe wytyczne i sposób zaliczenia kolokwium ustala prowadzący zajęcia.

LABORATORIUM 9. OBSŁUGA WE/WY. OBSŁUGA BŁĘDÓW I WYJĄTKÓW.

Cel laboratorium:

Omówienie obsługi wejścia i wyjścia oraz obsługi błędów i wyjątków.

Zakres tematyczny zajęć:

- zapis i odczyt do/z pliku,
- manipulatory,
- obsługa błędów,
- obsługa wyjątków.

Pytania kontrolne:

1. Jakie strumienie znajdują się w bibliotece *iostream*?
2. Jakie kroki należy wykonać aby czytać lub zapisywać z/do pliku?
3. W jakich trybach może zostać otwarty plik?
4. Do czego służą napisy strumieniowe?
5. Do czego służą manipulatory?
6. Na czym polega mechanizm wyjątków?

Operacje WE/WY:

Operacje WE/WY w języku C++ realizowane są za pomocą strumieni. Strumienie mogą być wejściowe i wyjściowe. Strumień wyjściowy oznacza, że przekazywana jest informacja z programu do wyjścia. Wyjściem może być na przykład: terminal czy plik. Natomiast strumień wejściowy oznacza, że przekazywana jest informacja z wejścia do programu. Wejściem może być na przykład klawiatura czy plik. Obsługę strumieni umożliwia klasa *iostream* dziedzicząca po klasie *istream*, która obsługuje strumień wejściowy oraz dziedzicząca po klasie *ostream*, która obsługuje strumień wyjściowy. Do obsługi plików należy użyć bardziej wyspecjalizowanej klasy *fstream*, która umożliwia dostęp do klasy *ifstream* oraz *ofstream*. Klasa *ifstream* umożliwia czytanie z pliku a klasa *ofstream* umożliwia zapis do pliku. Biblioteka *iostream* zawiera definicję czterech strumieni:

- *cin* – standardowy strumień wejściowy;
- *cout* – standardowy strumień wyjściowy;
- *cerr* – standardowy strumień komunikatów o błędach (niebuforowany);
- *clog* – standardowy strumień komunikatów o błędach (buforowany).

Zalecane jest stosowanie strumienia komunikatów o błędach zamiast strumienia wyjściowego do wyświetlania komunikatów w sytuacji kiedy chcemy wyświetlić informację o niepowodzeniu jakiejś operacji. Wtedy kod jest czytelny i łatwo znaleźć, gdzie wyświetlana jest informacja o błędzie.

Zapis oraz odczyt może być formatowany lub nieformatowany. Formatowany zapis/odczyt oznacza, że informacja jest interpretowana/formatowana, na przykład można przeczytać liczbę, opuszczając białe znaki. Nieformatowany zapis/odczyt oznacza, na przykład, że czytane są bajty (znaki, 1 znak = 1 bajt) bez żadnej interpretacji/formatowania. Użycie strumienia umożliwia odczyt formatowany natomiast użycie poniższych funkcji dotyczy zapisu i odczytu niesformatowanego. Czytać można na przykład z klawiatury lub z pliku, a zapisywać można

na konsolę lub do pliku. Do funkcji umożliwiających zapis i odczyt nieformatowany należą między innymi:

- *get* – czytanie jednego bajtu. Jeśli czytanie nie powiodło się to zostanie zwrócony znak końca *EOF*;
- *getline* – czytanie do przekazanego bufora całej linijki, domyślnie do końca linijki (ale bez znaku końca). Można również określić rozmiar bufora i znak do którego ma się odbywać czytanie;
- *read* – umożliwia przeczytanie określonej liczby bajtów do bufora. Używana jest głównie do plików binarnych;
- *ignore* – wczytuje określoną liczbę znaków, domyślnie jeden, i nigdzie ich nie zapisuje;
- *put* – wstawia znak do strumienia;
- *write* – umożliwia zapis określonej liczby bajtów do strumienia. Używana jest głównie do plików binarnych;

Manipulatory:

Manipulatory to specjalne funkcje lub obiekty funkcyjne, które umożliwiają formatowanie strumieni: wejściowych i wyjściowych. W celu korzystania z manipulatorów argumentowych należy użyć biblioteki *iostream*. Manipulatory mogą być argumentowe lub bezargumentowe. Manipulatory bezargumentowe wstawiane są do strumienia bez nawiasów. Natomiast manipulatory argumentowe wywołuje się podobnie jak zwykłe funkcje podając argumenty w nawiasach. Niektóre z manipulatorów wprowadzają zmiany na trwałe, więc należy pamiętać, aby po skorzystaniu, do strumienia wstawić odpowiedni manipulator lub wywołać metodę resetującą odpowiednią flagę, przywracając stan sprzed operacji. Wszystkie dostępne flagi zostały podane na stronie internetowej: https://www.cplusplus.com/reference/ios/ios_base/fmtflags/. Zarówno flagi jak i manipulatory umożliwiają formatowanie strumieni. Przy pracy z flagami przydatne mogą okazać się również funkcje: *setf* oraz *unsetf*, za pomocą których można ustawić lub zresetować daną flagę. Flagi oraz manipulatory mają takie same nazwy. Na stronie internetowej <https://en.cppreference.com/w/cpp/io/manip> można znaleźć wykaz wszystkich manipulatorów. Do manipulatorów bezargumentowych należą między innymi:

- *hex*, *oct*, *dec* – umożliwiają wyświetlenie liczby w systemie 16, 8 lub 10;
- *left*, *internal*, *right* – umożliwiają wyrównanie do lewej, środka i prawej strony;
- *scientific* – umożliwiają wyświetlenie liczby w postaci naukowej;
- *endl* – wysłanie do strumienia wyjściowego znaku końca linii;
- *showbase*, *noshowbase* – umożliwiają wyświetlenie liczby z podstawą (0 lub 0x)
- *showpos* – umożliwiają wyświetlenie liczby dodatniej ze znakiem +;

Z kolei do manipulatorów argumentowych należą między innymi:

- *setw* – umożliwia ustawienie szerokości pola dla najbliższej operacji w strumieniu;
- *setfill* – umożliwia ustawienie znaku, którym będą wypełnione puste miejsca, jeśli szerokość pola jest większa niż liczba znaków;
- *setprecision* – umożliwia ustawienie precyzji;
- *setiosflags* – ustawienie flagi;
- *resetiosflags* – wyłączenie flagi.

Listing 9.1. prezentuje przykład działania manipulatorów. Listing 9.1. zawiera:

- linijka 3 – wyświetlenie liczby w postaci dziesiętnej (domyślnie);
- linijka 4 – wyświetlenie liczby w postaci szesnastkowej;



- linijka 5 – wyświetlenie liczby w postaci szesnastkowej z 0x z przodu;
- linijka 6 – wyświetlenie liczby w postaci ósemkowej z 0 z przodu;
- linijka 7 – wyświetlenie liczby w postaci ósemkowej bez 0 z przodu;
- linijka 8 – wyświetlenie liczby w postaci dziesiętnej ze znakiem +;
- linijka 9 – wyświetlenie liczby w notacji naukowej;
- linijka 10 – ustawienie szerokości okna, domyślne wyrównanie do prawej;
- linijki 11-12 – ustawienie szerokości okna, wyrównanie zawartości do lewej strony oraz wypełnienie pozostałych znaków w oknie *;
- linijka 13 – wyłączenie flagi, która umożliwia zapis liczby w postaci naukowej;
- linijka 14 – wyświetlenie liczby z zadaną dokładnością.

```
1  int a = 223;
2  float b=21.365;
3  cout << "dec: " << dec << a << endl;

4  cout << "hex bez showbase: " << hex << a << endl;
5  cout << "hex z showbase:  " << showbase << a << endl;
6  cout << "oct z showbase:  " << oct << a << endl;
7  cout << "oct bez showbase: " << noshowbase << a << endl;

8  cout << "dec z showpos: " << dec << showpos << a << endl;

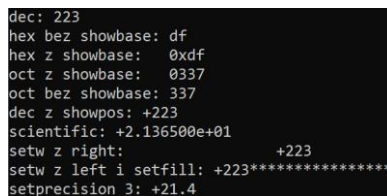
9  cout << "scientific: " << scientific << b << endl;

10 cout << "setw z right: " << setw(20) << a << endl;
11 cout << "setw z left i setfill: "
12     << setw(20) << left << setfill('*') << a << endl;

13 cout << "setprecision 3: " << resetiosflags(ios::scientific)
14     << setprecision(3) << b << endl;
```

Listing 9.1. Zastosowanie manipulatorów

Na rysunku 9.1. przedstawiony został wynik działania kodu z listingu 9.1.



```
dec: 223
hex bez showbase: df
hex z showbase: 0xdf
oct z showbase: 0337
oct bez showbase: 337
dec z showpos: +223
scientific: +2.136500e+01
setw z right: +223
setw z left i setfill: +223*****
setprecision 3: +21.4
```

Rys. 9.1. Wynik działania kodu z listingu 9.1.

Strumienie napisowe:

Strumienie napisowe umożliwiają wykonywanie operacji na napisach traktując je jak strumienie. Strumienie napisowe tworzą trzy klasy:

- *ostream* – strumień umożliwiający zapis;
- *istream* – strumień umożliwiający odczyt;
- *stringstream* – strumień umożliwiający zapis i odczyt.

Aby korzystać z tych klas należy dołączyć plik nagłówkowy *sstream*. Za pomocą tych klas możliwa jest konwersja różnych typów na napisy, formatowanie napisów oraz parsowanie napisów. Na listingu 9.2. przedstawione zostały przykłady zastosowania strumieni napisowych. Listing 9.2. zawiera:

- linijki 3 -6 – konwersja z typu *string* na *int*;
- linijki 7 – 10 – konwersja z typu *int* na *string*;
- linijki 11 – 14 – formatowanie – tworzenie wynikowego napisu połączonego z różnych typów;
- linijki 15 – 30 – parsowanie podanego napisu – rozdzielanie danych. Przy parsowaniu często mogą pojawiać się błędy więc wskazane jest użycie metody *fail*, za pomocą której można sprawdzić czy udało się poprawnie dokonać parsowania. Metodę *fail* można użyć dla każdej klasy strumieni napisowych;
- linijki 31-34 – parsowanie podanego napisu – wyłuskiwanie oddzielnych napisów za pomocą funkcji *getline*. Do konstruktora obiektu *ss* przekazany jest napis. Następnie na podstawie tego napisu dokonywane jest wyodrębnienie części napisu do podanego znaku, w tym przypadku do spacji.

```

1  string l1,l2,l3="Ala Nowak";
2  int num;
3  cout<<"Konwersja: "<<endl;
4  istringstream isstream("123");
5  isstream >> num;
6  cout<<num<<endl;

7  ostreamstream osstream;
8  osstream << 123;
9  l1=osstream.str();
10 cout << l1<< endl;

11 ostreamstream osstream1;
12 osstream1<< "Laboratorium " << 9 <<
13     ", zostalo jeszcze "<<6<<endl;
14 cout << "Formatowanie: "<<osstream1.str();

15 string stringEx;
16 int numberIntEx;
17 float numberFloatEx;
18 istringstream isstream1("Ala 10 3.45");
19 isstream1 >> stringEx >> numberIntEx >> numberFloatEx;
20 if(isstream1.fail())
21 {

```



```

22     cout << "Parsowanie nie udalo sie" << endl;
23 }
24 else
25 {
26     cout<< "Parsowanie udalo sie:" << endl;
27     cout<<"Napis: " << stringEx << endl;
28     cout<<"Liczba int: " << numberIntEx << endl;
29     cout<<"Liczba float: " << numberFloatEx << endl;
30 }
31 istringstream ss(13);
32 getline(ss, l1, ' ');
33 getline(ss, l2, ' ');
34 cout<<l1<<" "<<l2<<endl;

```

Listing 9.2. Zastosowanie strumieni napisowych

Na rysunku 9.2. zaprezentowany został wynik działania kodu z listingu 9.2. Na rysunku 9.2. liczby zostały wyświetlone ze znakiem + ponieważ na listingu 9.1. za pomocą manipulatora *showpos* została ustawiona taka opcja i nigdzie nie została zresetowana.

```

Konwersja:
+123
123
Formatowanie: Laboratorium 9, zostalo jeszcze 6
Parsowanie udalo sie:
Napis: Ala
Liczba int: +10
Liczba float: +3.45
Ala Nowak

```

Rys. 9.2. Wynik działania kodu z listingu 9.2.

Zapis i odczyt do/z pliku:

W celu zapisu do pliku lub odczytu z pliku należy stworzyć „uchwyt” do pliku, następnie powiązać go z konkretnym plikiem i otworzyć go. Jeśli zostanie podana tylko sama nazwa pliku to plik zostanie stworzony lub musi się znajdować w katalogu projektu. Plik można również umieścić w innym miejscu na dysku, wtedy należy podać pełną ścieżkę do niego. Po skończeniu pracy z plikiem należy pamiętać, aby zamknąć strumień. Plik można otworzyć, określając tryb otwarcia, na przykład:

- *ios::in* – odczyt ze strumienia;
- *ios::out* – zapis do strumienia. Jeśli pliku nie ma, zostanie utworzony nowy, w przeciwnym wypadku zawartość pliku zostanie nadpisana;
- *ios::app* – zapis do strumienia dołączając do istniejącej zawartości.

Przy pracy z plikami warto znać ich strukturę. Znaczenie ułatwia to dopasowanie sposobu odczytu lub zapisu danych. Najczęściej mamy do czynienia z plikami tekstowymi z rozszerzeniem *txt* lub *csv*. Pliki *csv* można wygenerować na przykład z arkusza kalkulacyjnego. W pliku *csv* dane są odseparowane od siebie separatorem. Najczęściej separatorem jest: spacja, tabulator czy średnik. Aby wygenerować plik z rozszerzeniem *csv* należy wybrać odpowiednią opcję przy zapisie danych w arkuszu kalkulacyjnym. Można również stworzyć taki plik samodzielnie na przykład w programie *Notatnik*. Na rysunku 9.3. przedstawiony został widok z arkusza kalkulacyjnego oraz pliku *csv* utworzonego w *Notatniku* dla tego samego pliku. W arkuszu kalkulacyjnym w każdej kolumnie znajdują się jakieś dane, tak samo jest w *Notatniku* tylko w *Notatniku* „kolumnę tworzą dwa średniki”.

	A	B	C	D	E	F
1	nrporzadkowy	imie	nazwisko	plec(KM)	nrtelefonu	email
2	1	Agata	Kowalska	K	818803491	abc@op.pl
3	2	Pawel	Abacki	M	888803491	abc1@op.pl
4	3	Piotr	Cabacki	M	838803491	abc2@wp.pl
5	4	Mateusz	Babacki	M	228803491	abc3@op.com
6	5	Olga	Nowak	K	228803491	abc4@gmail.pl
7	6	Anna	Adamek	K	238803491	abc5@op.pl
8	7	Anna	Adamczyk	K	678803491	abc6@op.pl
9	8	Monika	Mazur	K	818803491	abc7@outlook.com

nrporzadkowy;imie;nazwisko;plec(KM);nrtelefonu;email
1;Agata;Kowalska;K;818803491;abc@op.pl
2;Pawel;Abacki;M;888803491;abc1@op.pl
3;Piotr;Cabacki;M;838803491;abc2@wp.pl
4;Mateusz;Babacki;M;228803491;abc3@op.com
5;Olga;Nowak;K;228803491;abc4@gmail.pl
6;Anna;Adamek;K;238803491;abc5@op.pl
7;Anna;Adamczyk;K;678803491;abc6@op.pl
8;Monika;Mazur;K;818803491;abc7@outlook.com

Rys. 9.3. Fragment danych w arkuszu kalkulacyjnym oraz Notatniku.

Listing 9.3. przedstawia w jaki sposób można zapisać dane do pliku. Listing 9.3. zawiera:

- linijka 2 – dołączenie biblioteki umożliwiającej pracę z plikami;
- linijka 6 – stworzenie uchwytu, za pomocą którego będzie możliwy zapis do pliku;
- linijka 7 – powiązanie uchwytu z plikiem. Domyślny tryb to *ios::in*. Jeśli chcemy zapisać dane do pliku używając innego trybu należy go podać jako drugi argument funkcji *open*.
- linijka 8 – sprawdzenie czy plik został poprawnie otwarty. Nie jest konieczne używanie funkcji *is_open* ale wskazane;
- linijki 11-15 – zapisywanie danych do pliku. Zamiast *cout* podany jest uchwyt *fileOf*;
- linijka 16 – zamknięcie strumienia;
- linijka 20 – wyświetlenie komunikatu o błędzie. Można również użyć *cout*, ale użycie *cerr* jest bardziej wskazane.

```

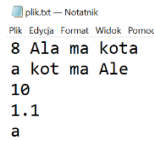
1  #include <iostream>
2  #include <fstream>
3  using namespace std;

4  int main()
5  {
6      ofstream fileOf;
7      fileOf.open("plik.txt");
8      if(fileOf.is_open())
9      {
10         cout<<"Plik otworzyl sie poprawnie"<<endl;
11         fileOf<<"8 Ala ma kota"<<endl;
12         fileOf<<"a kot ma Ale"<<endl;
13         fileOf<<10<<endl;
14         fileOf<<1.1<<endl;
15         fileOf<<'a'<<endl;
16         fileOf.close();
17     }
18     else
19         cerr<<"Blad przy otwieraniu pliku!"<<endl;
20     return 0;
21 }
```

Listing 9.3. Zapis do pliku



Na rysunku 9.4. przedstawiona została zawartość pliku, która stworzyła się po uruchomieniu kodu z listingu 9.3.



```
płik.txt - Notatnik
Plik  Edycja  Format  Widok  Pomoc
8 Ala ma kota
a kot ma Ale
10
1.1
a
```

Rys. 9.4. Zawartość pliku po uruchomieniu kodu z listingu 9.3.

Linijki 6 -7 z listingu 9.3. można zastąpić jedną linią przedstawioną na listingu 9.4. Tworząc uchwyt od razu do konstruktora można przekazać, z którym plikiem ma być skojarzony. Jeśli chcemy otworzyć plik z innym trybem niż domyślny to jako drugi argument konstruktora należy podać odpowiedni tryb.

```
ofstream fileOf("zapis.txt");
```

Listing 9.4. Zapis do pliku

Przykład użycia innego trybu niż domyślny przedstawia listing 9.5. Na tym listingu plik otwierany jest w trybie, który umożliwia dopisanie danych, jeśli plik już istnieje.

```
ofstream fileOf("zapis.txt", ios::app);
```

Listing 9.5. Zapis do pliku z dołączeniem danych

Istnieją różne sposoby odczytu danych z pliku. Można na przykład czytać zawartość całych linii albo znak po znaku. Znajomość zawartości pliku umożliwia dopasowanie najwygodniejszej metody odczytu danych. Do zaprezentowania możliwości czytania z pliku użyty zostanie plik stworzony podczas zapisu danych (rysunek 9.4.). Listing 9.6. przedstawia różne sposoby czytania z pliku bez sprawdzenia czy plik został otwarty, aby skupić się na czytaniu. Na listingu 9.6. przedstawione zostały sposoby odczytu danych z pliku. Listing 9.6. zawiera:

- linijka 1 – stworzenie uchwytu do odczytu danych z pliku;
- linijka 2 – skojarzenie uchwytu z plikiem, domyślny tryb czytania to *ios::in*;
- linijki 3 – 4 – czytanie jednej liczby;
- linijki 5 – 7 – czytanie jednego napisu (w tym przypadku do białego znaku);
- linijki 8 – 12 – czytanie całej linii do znaku nowej linii;
- linijki 13 – 15 – czytanie jednego bajtu – znaku;
- linijki 16 – 20 – czytanie całych linii do momentu wystąpienia końca pliku. Funkcja *eof()* zwraca *true* jeśli został osiągnięty *End-of-File*;
- linijka 21 – zamknięcie strumienia.

```
1  ifstream fileIf;
2  fileIf.open("plik.txt");

3  int number;
4  fileIf>>number;

5  string line;
6  fileIf>>line;
```



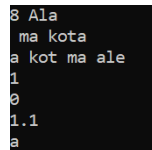
```
7      cout<<number<<" "<<line<<endl;
8      string line1;
9      string line2;
10     getline(fileIf,line1);
11     getline(fileIf,line2);
12     cout<<line1<<endl<<line2<<endl;

13     char c;
14     c=fileIf.get();
15     cout<<c<<endl;

16     while(!fileIf.eof())
17     {
18         getline(fileIf,line1);
19         cout<<line1<<endl;
20     }

21     fileIf.close();
```

Listing 9.6. Odczyt z pliku



```
8 Ala
ma kota
a kot ma ale
1
0
1.1
a
```

Rys. 9.5. Wynik działania kodu z listingu 9.6.

Linijki 19 – 23 z listingu 9.6. można zastąpić następującym kodem przedstawionym na listingu 9.7., który realizuje dokładnie to samo.

```
1 while(getline(fileIf,line1)){
2     cout<<line1<<endl;
3 }
```

Listing 9.7. Odczyt z pliku razem ze sprawdzeniem końca pliku

Obsługa błędów:

Błędy można podzielić na dwie grupy: błędy czasu wykonania oraz błędy kompilacji. Błędy czasu kompilacji związane są z wykryciem błędu w czasie kompilacji przez kompilator. Błędy czasu wykonania są związane z wykonywaniem programu i w większości przypadków są trudniejsze do zidentyfikowania. Jako przykłady błędów wykonania można podać: próba dzielenia przez 0, próba otworzenia nieistniejącego pliku, próba wyjścia poza tablicę.

W celu zapobiegania błędom można zastosować różne podejścia, na przykład: funkcje walidacyjne, sprawdzanie poprawności danych wejściowych, wyjątki. Zadaniem funkcji walidacyjnych jest sprawdzenie poprawności danych, które będą dalej używane, na przykład, w obliczeniach. Taka funkcja powinna zwrócić informację czy dane są poprawne. Listing 9.8. przedstawia przykład funkcji walidacyjnej, która zostanie wykorzystana do obliczenia pierwiastka z liczby. Listing 9.8. zawiera:

- linijki 4 – 10 – stworzenie funkcji, która będzie sprawdzała czy argument jest dodatni. Funkcja zwraca wartość *true* lub *false*;
- linijka 14 – wywołanie funkcji *valid1*. Jeśli funkcja zwróci *true*, oznacza to, że można liczyć pierwiastek z liczby rzeczywistej.

```
1  #include <iostream>
2  #include <math.h>
3  using namespace std;
4  bool valid1( int val )
5  {
6      if ( val>=0)
7          return true ;
8      else
9          return false ;
10 }
11 int main()
12 {
13     int num=-7;
14     if(valid1(num) )
15         cout<<sqrt(num)<<endl;
16     else
17         cout<<"Wartosc ujemna"<<endl;
18     return 0;
19 }
```

Listing 9.8. Funkcja walidacyjna

Kolejnym sposobem obsługi błędów może być zastąpienie błędnych argumentów, na przykład, domyślnymi argumentami. Na listingu 9.9. zaprezentowany został taki przykład. Listing 9.9. zawiera definicję funkcji a na listingu 9.10. przedstawione zostało jej wywołanie.

```
1  int valid2(int val)
2  {
3      if ( val<0)
4          return abs(val) ;
5  }
```

Listing 9.9. Funkcja walidacyjna

```
1  num=valid2(num) ;
2  cout<<sqrt (num) <<endl;
```

Listing 9.10. Wywołanie funkcji walidacyjnej

W celu zapobiegania błędom można również badać aktualny stan strumienia za pomocą metod wywoływanych na rzecz strumieni lub bezpośrednio w warunkach logicznych. Jeśli dla strumienia wywołana została metoda *fail*, oznacza to, że stan strumienia jest niewłaściwy. Wtedy metoda *fail* zwraca *true* (strumień zwraca *false*). W przeciwnym przypadku metoda *fail* zwróci *false* (strumień zwraca *true*). W poniższym przykładzie bezpośrednio strumień będzie umieszczony w warunku logicznym.

Kod przedstawiony na poniższym listingu należy dodać do funkcji *main*. Listing 9.11. przedstawia przykład, w którym obliczana jest wartość pierwiastka kwadratowego, jeśli użytkownik podał liczbę dodatnią lub równą 0. Jeżeli użytkownik nie poda wartości liczbowej lub wartość ta będzie ujemna, to wtedy całe wyrażenie będzie miało wartość logiczną *false*.

```
1  cout<<"Podaj liczbe: "<<endl;
2  if ((cin>>num) &&num>=0)
3      cout<<sqrt (num)<<endl;
4  else
5      cout<<"Wartosc ujemna"<<endl;
```

Listing 9.11. Właściwości strumienia *cin*

Na listingu 9.12. zaprezentowany został kolejny przykład sprawdzenia poprawności wprowadzonych danych. Kod znajdujący się na listingu należy umieścić pod poprzednimi przykładami. Metody *clear* i *ignore* zostały zastosowane w celu wymazania znacznika błędu i wyczyszczenia buforu (ważna jest kolejność wywołania). Jeśli w przykładzie na listingu 9.11. zostanie wczytana poprawna liczba wtedy przy pierwszym obrocie pętli trzeba wcisnąć *enter*.

```
1  do
2      {
3          cin.clear();
4          cin.ignore();
5          cout<<"Podaj liczbe: "<<endl;
6          cin>>num;
7          cout<<"Blednie wczytano!"<<endl;
8      }
9  while(!cin);
```

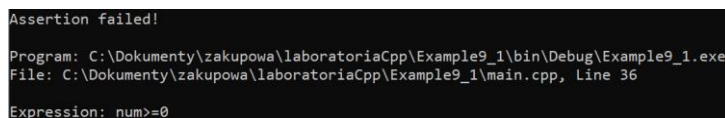
Listing 9.12. Wczytywanie za pomocą *cin*

Kolejnym sposobem na sprawdzenie poprawności danych jest użycie funkcji *assert*. Aby korzystać z tej funkcji należy dodać nagłówek *assert.h*. Do funkcji *assert* przekazywany jest warunek, jeśli warunek nie jest spełniony wtedy *assert* wyświetla warunek, który nie został spełniony, podaje nazwę pliku oraz numer wiersza. Na listingu 9.13. przedstawiony został przykład użycia funkcji *assert*.

```
1  num=-7;
2  assert (num>=0);
3  cout<<sqrt (num)<<endl;
```

Listing 9.13. Przykład użycia funkcji *assert*

Na rysunku 9.6. przedstawiony został wynik działania kodu z listingu 9.13.



```
Assertion failed!
Program: C:\Dokumenty\zakupowa\laboratoriaCpp\Example9_1\bin\Debug\Example9_1.exe
File: C:\Dokumenty\zakupowa\laboratoriaCpp\Example9_1\main.cpp, Line 36
Expression: num>=0
```

Rys. 9.6. Wynik działania kodu z listingu 9.13.

Obsługa wyjątków:

Obsługa wyjątków pozwala na zarządzanie błędami. Mechanizm wyjątków polega na tym, że w sytuacji, kiedy wystąpił błąd, wysyłany jest sygnał przy pomocy instrukcji *throw*. W sekcji *try* umieszczane są instrukcje, w których mogą wystąpić wyjątki (błędy). Natomiast w sekcjach *catch* następuje przechwytywanie rzucanych wyjątków i ich obsługa, na przykład: próba naprawienia błędu lub zignorowanie. Jeśli dany rodzaj wyjątku nie został opisany w sekcji *catch* a pojawi się w programie, wtedy wywoływana jest funkcja *terminate()*, która powoduje wywołanie funkcji *abort()*. Wyjątki można podzielić na systemowe, które są już zdefiniowane oraz na zdefiniowane samodzielnie. Wykaz wszystkich wyjątków systemowych można znaleźć na stronie internetowej <https://en.cppreference.com/w/cpp/error/exception> Listing 9.14. przedstawia przykład tworzenia i obsługi wyjątku. Listing 9.14. zawiera:

- linijka 7 – rzucenie wyjątku typu *int*. Typ, który jest rzucany przez wyjątek w żaden sposób nie odnosi się do zwracanego typu przez funkcję;
- linijki 13 – 17 – sekcja *try*, w której umieszczone zostało wywołanie funkcji *sqrt_function*. Wywołanie funkcji *sqrt_function* z argumentem -5 spowoduje, że zostanie rzucony wyjątek. Wtedy program nie wykonuje linii umieszczonych pod wywołaniem funkcji *sqrt_function*, tylko przechodzi do sekcji *catch*;
- linijki 18 – 21 – sekcja *catch*. Może być kilka rodzajów sekcji *catch*, ważne aby w każdej sekcji łapany był inny rodzaj wyjątku. W funkcji *sqrt_function* został rzucony wyjątek typu *int* więc zostanie on w sekcji *catch* złapany;
- linijki 22 – 25 – domyślna sekcja *catch*. Nie jest ona obowiązkowa ale dzięki niej możemy złapać wyjątek, który został rzucony a nie został uwzględniony w innych sekcjach *catch*, dlatego też należy domyślną sekcję *catch* umieścić na końcu.

```

1  #include <iostream>
2  #include <math.h>
3  using namespace std;
4  double sqrt_function(int a)
5  {
6      if(a<0)
7          throw -1;
8      else
9          return sqrt(a);
10 }
11 int main()
12 {
13     try
14     {
15         double result=sqrt_function(-5);
16         cout<<"Wynik to: "<<result<<endl;
17     }
18     catch(int w1)
19     {
20         cout<<"Zlapany wyjatek int "<<w1<<endl;
21     }
22     catch(...)
23     {

```



```

24         cout<<"Brak zdefiniowanego wyjatku"<<endl;
25     }
26     cout<<"Po sekcji try"<<endl;
27     return 0;
28 }
    
```

Listing 9.14. Obsługa wyjątku – przykład 1

Na rysunku 9.7. przedstawiony został wynik kodu z listingu 9.14. Jak można zauważyć wyjątek został złapany a następnie wykonała się instrukcja umieszczona po sekcjach *try-catch*.

```

Złapany wyjątek int -1
Po sekcji try

Process returned 0 (0x0)
Press any key to continue.
    
```

Rys. 9.7. Wynik działania kodu z listingu 9.14.

Teraz zobaczymy jaki będzie wynik jeśli usuniemy sekcje *try-catch* i wywołamy funkcję *sqr_function*. Jak widać na rysunku 9.8, program zakończył się z błędem i instrukcja umieszczona za wywołaniem funkcji nie została zrealizowana.

```

terminate called after throwing an instance of 'int'

Process returned 3 (0x3)   execution time : 2.219 s
Press any key to continue.
    
```

Rys. 9.8. Wynik działania kodu z listingu 9.14 po usunięciu sekcji *try-catch*

Kolejny przykład będzie przedstawiał stworzenie własnej klasy wyjątku. Najpierw należy zacząć od tworzenia klasy. Z uwagi na dydaktyczny wymiar laboratorium deklaracja oraz definicja klasy zostanie umieszczona w pliku *main.cpp*. Do już istniejącego pliku można dodać poniższe kody. Stworzona została abstrakcyjna klasa *Error* oraz jedna klasa potomna *SquareRootError*, która będzie dotyczyła błędu przy obliczaniu pierwiastka kwadratowego z liczby. Dzięki takiemu rozwiązaniu w łatwy sposób będzie można stworzyć kolejną klasę potomną dla kolejnych rodzajów błędu. Następnie za pomocą mechanizmu polimorfizmu będzie możliwe łapanie wyjątku za pomocą klasy bazowej. Na listingu 9.15. przedstawiona została klasa *Error* oraz *SquareRootError*. Klasa *Error* posiada metodę czysto wirtualną *error*, której definicja znajdzie się w klasach potomnych. W klasie *ErrorSqaureRoot* znajduje się definicja metody *error* z odpowiednim komunikatem.

```

1  class Error
2  {
3  public:
4      virtual void error()=0;
5      virtual~Error() {};
6  };

7  class SquareRootError: public Error
8  {
9  public:
10     void error()
    
```



```

11      {
12          cout<<"Pierwiastek z liczby ujemnej"<<endl;
13      }
14  };

```

Listing 9.15. Klasa *Error* i *SquareRootError*

W celu zaprezentowania przykładu zostały stworzone dwie funkcje. Ich zadaniem jest obliczenie pierwiastka kwadratowego z liczby. W przypadku, kiedy jest to niemożliwe funkcje rzucają wyjątek. Funkcja *sqr_function1* rzuca obiekt klasy *SquareRootError* a funkcja *sqr_function2* rzuca wskaźnik na obiekt *SquareRootError*. Listing 9.16. prezentuje kody funkcji.

```

1  double sqr_function1(int a)
2  {
3      if(a<0)
4          throw SquareRootError();
5      else
6          return sqrt(a);
7  }
8  double sqr_function2(int a)
9  {
10     if(a<0)
11         throw new SquareRootError();
12     else
13         return sqrt(a);
14 }

```

Listing 9.16. Funkcje *sqr_function1* oraz *sqr_function2*

Listing 9.17. przedstawia wywołanie funkcji *sqr_function1* oraz *sqr_function2* w sekcji *try*. Listing 9.17. zawiera:

- linijki 6 – 9 – przechwytywanie wyjątku typu *SquareRootError*. W funkcji *sqr_function1* został rzucony obiekt klasy *SquareRootError*. Za pomocą obiektu *p* wywoływana jest metoda *error*;
- linijki 15 – 19 – przechwytywanie wyjątku typu *Error**. W funkcji *sqr_function2* rzucany jest wskaźnik na obiekt klasy *SquareRootError*, a więc mamy do czynienia z polimorfizmem. Dzięki temu można mieć kilka klas potomnych i obiektem klasy rodzica można złapać wszystkie klasy potomne. Dzięki temu, że dla każdej z klas potomnych jest zaimplementowana klasa *error*, wywoła się odpowiednia metoda. Należy pamiętać, że przy rzuceniu wyjątku używany był operator *new*, więc w sekcji *catch* należy użyć *delete*;
- linijki 25 – 28 – przechwytywanie wyjątku referencji do klasy *Error*. W funkcji *sqr_function1* rzucany jest wyjątek typu *SquareRootError*. Podobnie jak w przypadku linijek 15 – 19.

```

1      try
2      {
3          double result=sqr_function1(-5);

```



```

4         cout<<"Wynik to: "<<result<<endl;
5     }
6     catch(SquareRootError p)
7     {
8         p.error();
9     }

10    try
11    {
12        double result=sqrt_function2(-5);
13        cout<<"Wynik to: "<<result<<endl;
14    }
15    catch(Error *b)
16    {
17        b->error();
18        delete b;
19    }

20    try
21    {
22        double result=sqrt_function1(-5);
23        cout<<"Wynik to: "<<result<<endl;
24    }
25    catch(Error &b)
26    {
27        b.error();
28    }

```

Listing 9.17. Wywołanie funkcji *sqrt_function1* oraz *sqrt_function2*

We wszystkich przypadkach na konsoli wyświetli się komunikat „Pierwiastek z liczby ujemnej”. Ostatni zaprezentowany przykład będzie dotyczył wyjątków systemowych. W tym celu została stworzona funkcja, której kod został przedstawiony na listingu 9.18. W funkcji zostanie podjęta próba przydzielenia pamięci. Próba zakończy się rzuceniem wyjątku systemowego *bad_alloc* ze względu na rozmiar tablicy.

```

1 void function_bad_alloc1()
2 {
3     int* tab=new int [10000000000000];
4 }

```

Listing 9.18. Kod funkcji *function_bad_alloc1*

Listing 9.19. prezentuje wywołanie funkcji *function_bad_alloc1* w sekcji *try*. Listing 9.19. zawiera:

- linijki 5 – 9 – funkcja *function_bad_alloc1* rzuca wyjątek *bad_alloc*. W sekcji *catch* obiekt tej klasy jest łapany. Za pomocą gotowej metody *what* możliwe jest wyświetlenie komunikatu o błędzie;
- linijki 15 – 18 – w sekcji *catch* łapany jest obiekt klasy *Exception*. Jest to nadrzędna klasa wyjątków. Zachodzi tutaj polimorfizm podobnie jak w poprzednim przykładzie.

```

1      try
2      {
3          function_bad_alloc1();
4      }
5      catch (bad_alloc &ba)
6      {
7          cout<<"function1 bad alloc: "<<ba.what()<<endl;
8      }
9
10     try
11     {
12         function_bad_alloc1();
13     }
14     catch (exception &e)
15     {
16         cout<<"function1 exception: "<<e.what()<<endl;
17     }

```

Listing 9.19. Wywołanie funkcji `function_bad_alloc1` w sekcji `try-catch`

W obydwu przypadkach metoda `what` zwróci tekst: `std::bad_alloc`.

Zadania do wykonania:

Zadanie 9.1. Kod programu

Napisz program, który wyświetli kod programu w języku C++ usuwając komentarze, które są poprzedzone `//` wraz z tymi znakami. Aby usunąć komentarze można użyć metody `ignore`. Należy skorzystać z pliku `kod.txt`. W programie należy obsłużyć sytuację, kiedy plik nie będzie istniał poprzez złapanie odpowiedniego wyjątku.

Podpowiedź: Metoda `peek` umożliwia podgląd kolejnego znaku. Można ją użyć aby sprawdzić czy występują po sobie dwa znaki `//`.

Zadanie 9.2. Oczko

Napisz program, w którym użytkownik będzie podawał dodatnią liczbę całkowitą do momentu aż suma podanych przez użytkownika liczb będzie mniejsza niż oczko (21). Jeśli suma będzie równa 21 wtedy należy wyświetlić odpowiedni komunikat i zakończyć program. Jeśli suma przekroczy 21 należy rzucić wyjątek informujący, że suma została przekroczona. Wyjątek należy obsłużyć umożliwiając ponowne podanie ostatniej liczby. W programie powinna zostać obsłużona sytuacja, kiedy użytkownik zamiast liczby podaje inny typ danych. Wtedy użytkownik powinien zostać poproszony o ponowne podanie liczby. Program powinien wyświetlić po ilu próbach (udanych i nieudanych) udało się osiągnąć oczko.

Zadanie 9.3. Kartoteka studentów

Napisz program do obsługi kartoteki studentów. Dane do zadania znajdują się w pliku „dane.csv”, gdzie w pierwszej linijce zapisane są nagłówki poszczególnych kolumn: imię, nazwisko, płeć, ocena oraz opcjonalnie email. W kolejnych linijkach znajdują się dane studentów. Każda linijka zawiera informację o jednej osobie. Dane rozdzielone są między sobą średnikiem. Użytkownik podaje ścieżkę do pliku w celu wczytania danych do programu (książka adresowa) i ma do wyboru następujące opcje:

1. Wyświetlenie książki adresowej na konsolę w postaci tabelki bez konturów z podziałem na imię, nazwisko, płeć, ocenę oraz ewentualnie email. Powinny zostać zastosowane stałe odstępki (na przykład 20) między kolejnymi kolumnami, puste miejsca powinny zostać wypełnione znakiem _;
2. Zapisanie do książki adresowej kolejnej osoby. Imię oraz nazwisko mają zaczynać się dużą literą alfabetu, w imieniu i nazwisku mogą występować tylko litery alfabetu łacińskiego. Płeć to K lub M, a email musi zawierać znak @. Należy obsłużyć sytuację w której którakolwiek z danych jest niepoprawna;
3. Wyświetlenie osób o podanym nazwisku na konsolę;
4. Stworzenie plików *k.csv* oraz *m.csv*. Plik *k.csv* będzie zawierać dane kobiet a plik *m.csv* dane mężczyzn. Należy pamiętać aby do plików dodać nagłówek;
5. Wyświetlenie X pierwszych studentów, liczbę X podaje użytkownik. Należy obsłużyć sytuację, w której jest mniej studentów niż podana liczba X;
6. Posortowanie studentów względem oceny rosnąco;
7. Wyjście z programu.

Należy zapewnić kontrolę błędów, jakie mogą pojawić się przy wprowadzaniu danych przez użytkownika z konsoli. Dodatkowo należy rozpatrzyć przypadek, że plik o podanej nazwie nie istnieje.

Wczytane z pliku dane studentów należy przechowywać w wybranym przez siebie kontenerze z biblioteki STL. Informacje o pojedynczym studencie należy przechowywać jako obiekt klasy. Po zakończeniu programu dane wszystkich dopisanych studentów powinny zostać dopisane do istniejącego pliku „dane.csv”.

LABORATORIUM 10. KLASA STRING I ELEMENTY BIBLIOTEK STL I BOOST. WYRAŻENIA REGULARNE.

Cel laboratorium:

Omówienie elementów klasy *string* oraz wyrażeń regularnych.

Zakres tematyczny zajęć:

- klasa *string*,
- wyrażenia regularne,
- elementy biblioteki *Boost*.

Pytania kontrolne:

1. Jakie operacje można wykonywać na napisach?
2. Jakie algorytmy znajdują się w bibliotece *Boost Algorithms*, które umożliwiają obsługę napisów?
3. Jaka jest różnica pomiędzy *regex_match* oraz *regex_search*?
4. Jakie są metaznaki?

Klasa *string*:

Klasa *string* została stworzona na potrzeby obsługi łańcuchów znaków. Klasa *string* jest zdefiniowanym typem więc można powiedzieć, że *string* to kolejny typ danych. Umieszczona jest w przestrzeni nazw *std::*. Aby wykonywać operacje na łańcuchach znaków należy dołączyć odpowiednią bibliotekę – bibliotekę *string*. Typ *string* jest specjalizacją szablonu klasy *basic_string<>* dla typu *char*. Zmienna *string* jest sekwencyjnym kontenerem przechowującym znaki i można go porównać do kontenera *vector*. W klasie *string* znajduje się sporo metod, które umożliwiają pracę z napisami. Są to między innymi:

- *begin* – iterator zwracający adres początku napisu;
- *end* – iterator zwracający adres znajdujący się za ostatnim elementem w napisie;
- *length* – zwraca długość napisu;
- *size* – zwraca długość napisu;
- *resize* – zmiana rozmiaru napisu;
- *front* – zwraca pierwszy element w napisie;
- *back* – zwraca ostatni element w napisie;
- *append* – dodaje napis do napisu;
- *push_back* – dodaje element na koniec napisu;
- *assign* – przypisuje nową wartość napisowi;
- *insert* – wstawia element do napisu;
- *erase* – usuwa element z napisu;
- *replace* – zastępuje część ciągu drugim;
- *pop_back* – usuwa ostatni element w napisie;
- *find* – znajduje pierwsze wystąpienie podanego napisu;
- *rfind* – znajduje ostatnie wystąpienie podanego napisu;
- *find_first_of* – znajduje pierwsze wystąpienie podanych liter;
- *find_last_of* – znajduje ostatnie wystąpienie podanych liter;



- *find_first_not_of* – znajduje pierwsze wystąpienie liter, które nie zostały podane;
- *find_end_not_of* – znajduje ostatnie wystąpienie liter, które nie zostały podane;
- *substr* – ucina napis.

Większość z wymienionych funkcji występuje w kilku wersjach, a niektóre funkcje mogą przyjmować jako argumenty zarówno numery pozycji jak i iteratory. Na listingu 10.1. zaprezentowane zostały przykładowe operacje na łańcuchach znaków. Listing 10.1. zawiera:

- linijki 5 – 6 - stworzenie i zainicjalizowanie zmiennej typu string;
- linijka 9 – wczytywanie jednego słowa do zmiennej typu *string*;
- linijki 10 – 11 – przygotowanie strumienia do kolejnego wczytywania;
- linijka 13 – wczytywanie całego zdania do momentu naciśnięcia znaku *enter*;
- linijki 15 – 18 – porównanie dwóch napisów;
- linijki 19 – 20 – konkatencja dwóch napisów (łączenie napisów);
- linijka 21 – odwołanie się do pojedynczego znaku;
- linijki 22 – 26 – odwołanie się do wszystkich znaków w napisie;
- linijki 27 – 28 – dodanie elementów na koniec istniejącego napisu;
- linijki 29 - 33 – pierwsze wyszukanie wystąpienia podanego znaku. Można podać również napis oraz miejsce od którego powinno rozpocząć się wyszukiwanie;
- linijki 34 – 36 – wyszukanie ostatniego wystąpienia litery, która nie należy do podanych;
- linijki 37 – 38 – obcięcie napisu;
- linijka 39 – wstawienie znaku na określoną pozycję;
- linijka 40 – dodanie znaku na koniec;
- linijki 42 – 44 – odwołanie się do wszystkich znaków w napisie poprzez użycie iteratora;
- linijki 45 – 48 – usunięcie fragmentu napisu;

```
1  #include <iostream>
2  using namespace std;

3  int main()
4  {
5      string line1="Programowanie";
6      string line2("Programowanie");
7
8      cout<<"Podaj slowo: "<<endl;
9      cin>>line1;
10     cin.clear();
11     cin.ignore();
12     cout<<"Wczytane slowo: "<<line1<<endl;
13     getline(cin, line2);
14     cout<<"Zdanie: "<<line2<<endl;

15     if(line1==line2)
16         cout<<"Napisy sa takie same"<<endl;
17     else
18         cout<<"Napisy roznia sie"<<endl;

19     line1=line1+" "+line2;
```

```
20     cout<<"Napis: "<<line1<<endl;
21     cout<<"Druga litera: "<<line1[1]<<endl;

22     cout<<"Napis w for: "<<endl;
23     for(int i=0;i<line1.length();i++){
24         cout<<line1[i];
25     }
26     cout<<endl;

27     line1.append(" i psa");
28     cout<<"Napis po append: "<<line1<<endl;

29     int ind=line1.find('a');
30     cout<<"Pierwsze wystapienie znkau 'a' "<<ind<<endl;
31     ind=line1.find('a',7);
32     cout<<"Pierwsze wystapienie znkau 'a' po 7 pozycji "
33         <<ind<<endl;

34     ind=line1.find_last_not_of("kgtoa");
35     cout<<"Ostatnie wystepienie litery ktora nie jest
36         kgtoa: "<<ind<<endl;

37     line2=line1.substr(0,3);
38     cout<<"Substr(0,3): "<<line2<<endl;

39     line1.insert(line1.begin(),'*');
40     line1.push_back('*');
41     cout<<"Napis po insert i push_back: "<<line1<<endl;

42     for (string::iterator it = line1.begin();
43         it != line1.end(); it++)
44         cout << *it ;

45     string::iterator del;
46     del=line1.erase(line1.begin()+0,line1.begin()+5);
47     cout<<*del<<endl;
48     cout<<"Napis po erase: "<<line1<<endl;

49     return 0;
50 }
```

Listing 10.1. Przykłady operacji na łańcuchach znaków

Na rysunku 10.1. zaprezentowany został wynik działania kodu z listingu 10.1.



```

Podaj slowo:
Ala
Wczytane slowo: Ala
ma kota
Zdanie: ma kota
Napisy roznia sie
Napis: Ala ma kota
Druga litera: l
Napis w for:
Ala ma kota
Napis po append: Ala ma kota i psa
Pierwsze wystapienie znkau 'a' 2
Pierwsze wystapienie znkau 'a' po 7 pozycji 10
Ostatnie wystapienie litery ktora nie jest kgtoa: 15
Substr(0,3): Ala
Napis po insert i push_back: *Ala ma kota i psa*
*Ala ma kota i psa*m
Napis po erase: ma kota i psa*
    
```

Rys. 10.1. Wynik działania kodu z listingu 10.1.

Do zmiennych typu *string* można zastosować algorytmy z biblioteki *STL*. Należy wtedy dołączyć plik nagłówkowy *algorithm*. Większość z tych algorytmów została omówiona podczas laboratorium 4 i 5. Na listingu 10.2. przedstawione zostały kody dwóch funkcji, które zostaną użyte na listingu 10.3.

```

1 void print(char x){
2     cout<<x<<"|";
3 }
4 char toUpper(char ch){
5     if((ch>='a' and ch<='z'))
6         return ch-32;
7 }
    
```

Listing 10.2. Funkcja print oraz toUpper

Listing 10.3. przedstawia przykład użycia algorytmów. Kod jest kontynuacją listingu 10.1. Listing 10.3. zawiera:

- linijki 1 – 3 – zastosowanie algorytmu *transform*, do zamiany wszystkich liter na duże litery. Algorytm *transform* kopiuje zwracaną wartość do elementu z zakresu wyjściowego. Funkcja *toUpper* dla każdej małej litery alfabetu zwraca jej duży odpowiednik;
- linijki 5 – 7 – zastosowanie algorytmu *find* do znalezienia wystąpienia litery *K*;
- linijki 8 – 9 – zastosowanie algorytmu *for_each*. Funkcja *print* umożliwia wyświetlenie każdej litery oddzielonej znakiem |.

```

1 cout<<"Transform"<<endl;
2 transform(line1.begin(),line1.end(),line1.begin(),
3           toUpper);
4 cout<<line1<<endl;

5 cout<<"find"<<endl;
6 del=find(line1.begin(),line1.end(),'K');
7 cout<<*del<<endl;

8 cout<<"for_each"<<endl;
9 for_each(line1.begin(),line1.end(),print);
    
```

Listing 10.3. Przykłady operacji na łańcuchach znaków



Na rysunku 10.2. przedstawiony został wynik działania kodu z listingu 10.3.

```
Transform
MA KOTA I PSA*
find
K
for_each
M|A| |K|O|T|A| |I| |P|S|A|*|
```

Rys. 10.2. Wynik działania kodu z listingu 10.3.

W bibliotece *Boost* znajdują się również algorytmy, które można zastosować dla łańcuchów znaków. Zostaną one omówione w następnej części.

Zmienna string w bibliotece Boost:

Biblioteka *Boost String Algorithms* udostępnia algorytmy związane z ciągami znaków, które rozszerzają bibliotekę *algorithms STL*. W celu korzystania z biblioteki należy dołączyć plik nagłówkowy: *boost/algorithm/string.hpp*. W bibliotece *Boost String Algorithms* można znaleźć między innymi następujące algorytmy:

- *to_upper* – zamiana wszystkich liter ciągu znaków na duże litery;
- *to_lower* – zamiana wszystkich liter ciągu znaków na małe litery;
- *erase_first* – usunięcie pierwszego wystąpienia podanego podciągu;
- *erase_nth* – usunięcie *n* – tego wystąpienia podanego podciągu;
- *erase_all* – usunięcie wszystkich wystąpień podanego podciągu;
- *replace_first* – zastąpienie pierwszego wystąpienia podciągu, drugim podciągiem;
- *replace_last* – zastąpienie ostatniego wystąpienia podciągu, drugim podciągiem;
- *join* – połączenie napisów wraz z dodaniem separatora;
- *trim* – usunięcie spacji/tabulatorów na początku i końcu napisu;
- *trim_left* – usunięcie spacji/tabulatorów z początku napisu;
- *starts_with* – sprawdzenie czy dany ciąg znaków zaczyna się podanym podciągiem;
- *ends_with* – sprawdzenie czy dany ciąg znaków kończy się podanym podciągiem;
- *contains* – sprawdzenie czy dany ciąg znaków zawiera podany podciąg;
- *split* – podzielenie podanego ciągu znaków według podanego separatora;

Większość algorytmów występuje w dwóch wersjach: ze słowem *copy* oraz bez tego słowa. Różnica pomiędzy dwoma wersjami algorytmu jest następująca. Algorytm ze słowem *copy* nie modyfikuje przekazanego argumentu, tylko zwraca nową wartość. Natomiast wersja algorytmu bez słowa *copy*, modyfikuje przekazaną zmienną. Na listingu 10.4. zaprezentowane zostały przykłady użycia algorytmów z biblioteki *Boost Algorithms*. Listing 10.4. zawiera:

- linijka 2 – dołączenie odpowiedniej biblioteki;
- linijka 9 – zamiana liter w napisie na duże litery alfabetu. Zmienna *line* zostanie zmodyfikowana;
- linijka 11 – zamiana liter alfabetu w napisie na małe litery alfabetu. Algorytm zwraca nowy wyraz;
- linijki 13 -14 – usunięcie pierwszego wystąpienia litery *o* w napisie;
- linijka 15 – usunięcie wszystkich wystąpień litery *o*;
- linijki 16 – 17 – zastąpienie pierwszego wystąpienia małej litery *p* dużą literą *P*;
- linijki 18 – 19 – zastąpienie wszystkich wystąpień małej litery *o* podciągiem *-O-*;

- linijki 20 – 21 – połączenie napisów przechowywanych w kontenerze *vector* w jeden napis, w którym poszczególne napisy oddzielone są spacjami;
- linijki 22 – 24 – usunięcie spacji z przodu i z tyłu napisu;
- linijki 25 – 28 – usunięcie z napisu elementów z przodu i z tyłu jeśli spełniają warunek, w tym przypadku jeśli elementy są znakiem -.
- linijki 29 – 32 – usunięcie z napisu elementów z przodu i z tyłu jeśli są liczbą;
- linijki 33 – 36 – sprawdzenie czy dany ciąg zawiera podany podciąg. W pierwszym przypadku na końcu, a w drugim czy w ogóle podany podciąg znajduje się z napisie;
- linijki 37 – 43 – podział napisu względem podanego separatora na oddzielne napisy i zapisanie ich do kontenera *vector*. Napisy zostaną podzielone względem spacji.

```
1  #include <iostream>
2  #include <boost/algorithm/string.hpp>
3  using namespace boost::algorithm;
4  using namespace std;

5  int main()
6  {
7      string line="programowanie";
8      cout<<line<<endl;
9      to_upper(line);
10     cout<<"To_upper: "<<line<<endl;
11     line=to_lower_copy(line);
12     cout<<"To_lower: "<<line<<endl;

13     cout<<"Erase_first: "
14         <<erase_first_copy(line,"o")<<endl;
15     cout<<"Erase_all:"<<erase_all_copy(line,"o")<<endl;
16     cout<<"Replace_first: "
17         <<replace_first_copy(line,"p","P")<<endl;
18     cout<<"replace_all: "
19         <<replace_all_copy(line,"o","-O-")<<endl;

20     vector<string> vec{"Programowanie", "w", "C++"};
21     cout << "Join: "<<join(vec, " ") << endl;

22     string s = "    Programowanie w C++    ";
23     cout<<s<<endl;
24     cout << "Trim: "<<"_" << trim_copy(s) << "_"<<endl;

25     s = "---Programowanie w C++---";
26     cout<<s<<endl;
27     cout <<"Trim_if is_any_of: "
28         <<trim_copy_if(s, is_any_of("-")) << endl;

29     s = "123Programowanie w C++12";
30     cout<<s<<endl;
31     cout << "Trim if digit: "
```



```

32         <<trim_copy_if(s, is_digit()) <<endl;

33     s = "Programowanie w C++";
34     cout<<s<<endl;
35     cout << "Ends_with: " <<ends_with(s, "C++") <<endl;
36     cout <<"Contains: " << contains(s, "gram") <<endl;

37     s = "Programowanie w C++";
38     cout<<s<<endl;
39     vector<string> vec1;
40     split(vec1, s, is_space());
41     cout<<"Zawartosc vectora: " <<endl;
42     for(int i=0; i<vec1.size(); i++)
43         cout<<vec1[i]<<endl;

44     return 0;
45 }
    
```

Listing 10.4. Przykłady użycia algorytmów z biblioteki *Boost Algorithms*

Na rysunku 10.3. przedstawiony został wynik działania kodu z listingu 10.4.

```

programowanie
To_upper: PROGRAMOWANIE
To_lower: programowanie
Erase_first: prgramowanie
Erase_all:prgramwanie
Replace_first: Programowanie
replace_all: pr-O-gram-O-wanie
Join: Programowanie w C++
      Programowanie w C++
Trim: _Programowanie w C++_
---Programowanie w C++---
Trim_if is_any_of: Programowanie w C++
123Programowanie w C++12
Trim if digit: Programowanie w C++
Programowanie w C++
Ends_with: 1
Contains: 1
Programowanie w C++
Zawartosc vectora:
Programowanie
w
C++
    
```

Rys. 10.3. Wynik działania kodu z listingu 10.4.

Kolejną biblioteką przedstawioną w ramach laboratorium będzie biblioteka *Boost Tokenizer*. Umożliwia dzielenie łańcuchów znaków na podciągi. Metoda podziału łańcucha znaków może być ustalona przez programistę za pomocą parametryzacji funktora. Jeśli metoda podziału nie zostanie zdefiniowana wtedy zostanie użyta domyślna wersja, która podzieli napis ze względu na spacje i punktuery. W celu korzystania z tej biblioteki należy dołączyć odpowiedni plik nagłówkowy: *boost/tokenizer.hpp*. Na listingu 10.5. przedstawiony został sposób użycia biblioteki. Poniższy kod należy dołączyć do kodu przedstawionego na listingu 10.4. Listing 10.5. zawiera:

- linijki 1 – 2 – stworzenie typu o nazwie *tokenizer*, aby w następnych linijkach posługiwać się krótszą nazwą;
- linijka 4 – stworzenie separatora, którym będzie spacja;

- linijka 5 - utworzenie domyślnego obiektu typu *tokenizer*, w którym ciąg znaków *string* dzielony jest na podstawie spacji;
- linijki 7 – 9 – uzyskanie dostępu do elementów *tokenizera* realizowane jest za pomocą odpowiedniego iteratora.

```
1  typedef boost::tokenizer
2  <boost::char_separator<char>> tokenizer;
3  s = "Programowanie w C++";
4  boost::char_separator<char> sep{" "};
5  tokenizer tok{s,sep};
6  cout<<s<<endl;
7  for (tokenizer::iterator it = tok.begin();
8      it != tok.end(); ++it)
9      cout << *it <<endl;
```

Listing 10.5. Przykłady użycia biblioteki Boost Tokenizer

Jako wynik działania kodu z listingu 10.5. trzy oddzielne słowa zostały wyświetlone.

Wyrażenia regularne:

Wyrażenie regularne to wzorzec do którego da się dopasować tylko pewien tekst. Wzorzec wyrażany jest za pomocą specjalnej składni. Wyrażenia regularne używa się do sprawdzenia czy napis pasuje do podanego przez wyrażenie wzorca. We wzorcu można stosować znaki specjalne i za ich pomocą tworzyć wyrażenia. Za pomocą metaznaków można stworzyć konstrukcję, która umożliwi stworzenie wzorca, w którym, na przykład, określone znaki mogą pojawić się kilka razy. Do metaznaków należą:

- . – dowolny pojedynczy znak;
- {} – licznik;
- [] – klasa znaków;
- () – grupa;
- * - powtórzenie 0 lub więcej;
- + - powtórzenie 1 lub więcej;
- ? – 0 lub 1 wystąpienie;
- | - alternatywna (lub);
- ^ - negacja;
- \$ - koniec wiersza.

Jeśli znak specjalny poprzedzony jest odwrotnym ukośnikiem \, to wtedy jest traktowany jako zwykły znak a nie specjalny. Wśród klasy znaków można wyróżnić takie klasy, które zostały już zdefiniowane, są to na przykład:

- [:alpha:] – dowolny znak alfabetu;
- [:digit:] – dowolna cyfra dziesiętna, w skrócie \d;
- [:lower:] – dowolna mała litera, w skrócie \l;
- [:upper:] – dowolna duża litera, w skrócie \u;
- [:punct:] – dowolny znak interpunkcyjny;
- [:alnum:] – dowolny znak alfanumeryczny, w skrócie \w;

Należy pamiętać, że odwrócony ukośnik jest znakiem specjalnym w C++, a więc należy go zamaskować. Maskowanie w C++ odbywa się po przez dodanie odwróconego ukośnika więc

w tym przypadku będą to dwa odwrócone ukośniki. Można również stworzyć własne przedziały tak zwane klasy znaków, na przykład:

- [a-z] – małe litery od a do z;
- [a-zA-Z] – małe i duże litery alfabetu;
- [^abc] – dowolny znak oprócz znaków a, b, c;
- [aA] – mała lub duża litera a.

Większość znaków specjalnych umieszczonych w klasach znaków traktowana jest jako zwykłe znaki. W celu skorzystania z wyrażeń regularnych należy dołączyć plik nagłówkowy *regex*. Biblioteka *regex* zawiera metody:

- *regex_match*, która sprawdza czy podana sekwencja pasuje do wyrażenia regularnego;
- *regex_search*, która sprawdza, czy sekwencja znaków częściowo pasuje do wyrażenia regularnego;
- *regex_replace*, która umożliwia zastępowanie sekwencji znaków, które pasują do wyrażenia regularnego.

Różnica pomiędzy *regex_match* oraz *regex_search* jest taka, że *regex_match* sprawdza, czy wyszukiwany tekst pasuje do wzorca wyrażenia regularnego, a *regex_search* sprawdza, czy wyszukiwany tekst zawiera podciąg, który pasuje do wzorca wyrażenia regularnego. Funkcja *regex_match* jest używana zwykle do sprawdzenia poprawności danych takich jak: email, adres, numer telefonu itd. W celu uzyskania szczegółów dotyczących dopasowań należy przekazać obiekt *match_results<>*. Dla łańcuchów znaków (*string*) będzie używana klasa *smatch*. Na obiekcie tej klasy można wywołać między innymi następujące metody aby dowiedzieć się więcej szczegółów na temat dopasowań:

- *size* – liczba dopasowań;
- *length* – długość dopasowania;
- *position* – numer indeksu, od którego zaczyna się dopasowanie;
- *str* – tekst, który spełnił dopasowanie;
- *prefix* – tekst znajdujący się przed pierwszym znakiem dopasowania;
- *suffix* – tekst znajdujący się za ostatnim znakiem dopasowania

Do iterowania po wszystkich dopasowaniach wyrażenia regularnego używane są iteratory typu *regex_iterator<>*. Podczas pracy z wyrażeniami regularnymi mogą wystąpić błędy związane z ich obsługą. Klasa *regex_error* umożliwia obsługę wyjątków związanych z wyrażeniami regularnymi.

Na listingu 10.6. przedstawione zostały przykłady użycia funkcji *regex_match*, *regex_search* oraz *regex_replace*. Listing 10.6. zawiera:

- linijka 2 – dołączenie biblioteki;
- linijki 7 – 10 – sprawdzenie czy zmienna *line* pasuje do wzorca (*Prog*)(.*). We wzorcu musi znaleźć się słowo *Prog* oraz dowolny znak powtórzony 0 lub więcej razy. Funkcja *regex_match* zwraca *true* jeśli udało się znaleźć dopasowanie, i *false* w przeciwnym wypadku;
- linijki 11 – 15 – druga wersja, obiekt typu *regex* został stworzony wcześniej;
- linijki 16 – 19 – trzecia wersja, zastosowanie iteratora, za pomocą którego można określić zasięg;
- linijki 20 - 27 – czwarta wersja, zastosowanie obiektu *smatch*, który zawiera informację o dopasowaniu. Na obiekcie *res* typu *smatch* można wywołać różne metody aby dowiedzieć się o szczegółach dopasowania. W przypadku metody *regex_match*, wartość *res.size()* może być równa 0 lub 1. Jeśli znaleziono dopasowanie wtedy można wyświetlić dopasowany tekst za pomocą *res[0]*;



- linijki 37 – 45 – wywołanie funkcji *regex_serach*, za pomocą której sprawdzone zostało czy wyszukiwany tekst zawiera podciągi wzorca (zapisane w nawiasach). Za pomocą metod: *size*, *length*, *suffix*, *prefix* itd. można wyciągnąć bardziej szczegółowe informacje o dopasowaniu. Jeśli nie odwołujemy się za pomocą konkretnego indeksu to domyślnie brany jest pod uwagę indeks 0, a więc cały wzorzec, a nie jego część;
- linijki 46 – 54 – wyświetlenie informacji o wszystkich znalezionych dopasowaniach;
- linijki 55 – 60 – druga wersja, z użyciem iteratora;
- linijki 61 – 66 – użycie funkcji *regex_replace*. W celu zaprezentowania zapisu wyrażenia regularnego wzorzec (C++) został zapisany w następujący sposób: `C\\+{2}`, oznacza to, że musi wystąpić litera C oraz znak + powtórzony dwa razy. Znak + poprzedzony jest dwoma ukośnikami, dodanie jednego ukośnika przed znakiem + powoduje, że znak + nie jest już znakiem specjalnym a drugi ukośnik maskuje pierwszy ukośnik, który w języku C++ jest znakiem specjalnym. Jeśli zastosowana byłby klasa znaków, wtedy nie trzeba byłoby maskować.

```
1  #include <iostream>
2  #include <regex>
3  using namespace std;

4  int main()
5  {
6      string line="Programowanie w C++";
7      if (regex_match (line, regex("(Prog)(.*)") ))
8          cout << "Znaleziono"<<endl;
9      else
10         cout<<"Nie znaleziono!"<<endl;

11     regex regPattern("(Prog)(.*)");
12     if (regex_match (line, regPattern ))
13         cout << "Znaleziono"<<endl;
14     else
15         cout<<"Nie znaleziono!"<<endl;

16     if (regex_match (line.begin(), line.end(),regPattern ))
17         cout << "Znaleziono"<<endl;
18     else
19         cout<<"Nie znaleziono!"<<endl;

20     smatch res;
21     regPattern="(Prog)(.*)";
22     regex_match(line,res,regPattern);
23     cout<<"Regex_match: "<<endl;
24     cout << "Czy znaleziony? " << res.size()<<endl;
25     if(res.size()!=0)
26         cout<<"Wzorzec: "<<res[0]<<
27         " zostal znaleziony"<<endl;

28     cout<<endl<<"Porownanie match i search"<<endl;
```



```

29     line="*****Programowanie w C++*****";
30     regPattern="(Prog)(.*)";
31     regex_match(line,res,regPattern);
32     cout<<"Regex_match: "<<endl;
33     cout << "Czy znaleziony? " << res.size()<<endl;
34     if(res.size()!=0)
35         cout<<"Wzorzec: "<<res[0]<<
36         "   zostal znaleziony"<<endl;

37     regex_search(line,res,regPattern) ;
38     cout<<"Regex_search: "<<endl;
39     cout << "Liczba dopasowan:" << res.size()<<endl;
40     cout << "res.str(): " << res.str() << endl;
41     cout << "res.length(): " << res.length() << endl;
42     cout << "res.position(): " << res.position() << endl;
43     cout<<"res.prefix: "<<res.prefix().str()<<endl;
44     cout<<"res.suffix: "<<res.suffix().str()<<endl;
45     cout << endl;

46     cout<<"Podgrupy ver1:"<<endl;
47     for (int i=0; i<res.size(); ++i)
48     {
49         cout << "[" << res[i] << "]" ";
50         cout << "res.str(): " << res.str(i) << endl;
51         cout << "res.position(): "<<res.position(i)<< endl;
52         cout<<"res.dl: "<<res.length(i)<<endl;
53         cout << endl;
54     }

55     cout << "Podgrupy ver2:" << endl;
56     for (auto pos = res.begin(); pos != res.end(); pos++)
57     {
58         cout << "[" << *pos << "]" ";
59         cout << "dl: " << pos->length() << endl;
60     }

61     cout<<endl<<"Replace: "<<endl;
62     line="*****Programowanie w C++*****";
63     regPattern="(C\\+{2})";
64     cout<<"Przed replace: "<<line<<endl;
65     line=regex_replace(line, regPattern, "Java");
66     cout<<"Po replace: "<<line<<endl;
67     return 0;
68 }

```

Listing 10.6. Przykłady użycia biblioteki regex

Na rysunku 10.4. przedstawiony został wynik działania kodu z listingu 10.6. Jak można zauważyć, na rysunku pod napisem „Podgrupy ver1”, zostały wyświetlone trzy sekcje.

Pierwsza sekcja dotyczy dopasowania całego wzorca ((Prog)(.*)) do fragmentu/całego tekstu, druga sekcja dotyczy pierwszej grupy ((Prog)), a trzecia sekcja dotyczy drugiej grupy((.*)).

```
Znaleziono
Znaleziono
Znaleziono
Regex_match:
Czy znaleziony? 3
Wzorzec: Programowanie w C++ zostal znaleziony

Porownanie match i search
Regex_match:
Czy znaleziony? 0
Regex_search:
Liczba dopasowan:3
res.str(): Programowanie w C++*****
res.length(): 27
res.position(): 7
res.prefix: *****
res.suffix:

Podgrupy ver1:
[Programowanie w C++*****] res.str(): Programowanie w C++*****
res.position(): 7
res.dl: 27

[Prog] res.str(): Prog
res.position(): 7
res.dl: 4

[ramowanie w C++*****] res.str(): ramowanie w C++*****
res.position(): 11
res.dl: 23

Podgrupy ver2:
[Programowanie w C++*****] dl: 27
[Prog] dl: 4
[ramowanie w C++*****] dl: 23

Replace:
Przed replace: *****Programowanie w C++*****
Po replace: *****Programowanie w Java*****
```

Rys. 10.4. Wynik działania kodu z listingu 10.6.

Poniżej zaprezentowane zostały przykłady wyrażeń regularnych:

- `s[a-z]+` – słowo zaczynające się na s i zawierające przynajmniej jedną literę z zakresu a - z;
- `(moj|twoj) kot` – wyrażenie mój kot lub twój kot;
- `[0-9]{2}-[0-9]{3}` – kod pocztowy: dwie cyfry następnie myślnik i kolejne trzy cyfry;
- `r.k` – litera r następnie jeden dowolny znak i potem litera k, na przykład: rok, rak;
- `kot(y|ki)` – słowa koty lub kotki;
- `r[ao]k` – słowo rak lub rok;
- `r[^ao]k` – słowo, które nie będzie zawierało liter a ni o jako drugiej litery w słowie, na przykład: ryk;
- `ko+t` – słowo, które będzie zawierało przynajmniej jedną literę o lub więcej, na przykład: kot, koooot;
- `\d{2}` – liczba dwucyfrowa;

Zadania do wykonania:

Zadanie 10.1. Liczba zmiennoprzecinkowa

Zdefiniuj wyrażenie regularne, które pozwoli na sprawdzenie czy w danym łańcuchu znaków znajduje się liczba zmiennoprzecinkowa ze znakiem (liczba całkowita od części ułamkowej oddzielona jest kropką, np. +6.789, -7.234). W funkcji *main* należy przetestować stworzone wyrażenia regularne.

Zadanie 10.2. Godzina

Zdefiniuj wyrażenie regularne do rozpoznawania godziny. Zakładamy, że zapis godziny jest następujący: *hh:mm:ss* lub *hh:mm*, gdzie *hh* to godziny (0-23), *mm* - liczba minut (0-59), *ss* – to liczba sekund (0-59). Podawanie sekund jest opcjonalne. Liczby (*hh,mm,ss*) zawsze zapisane są za pomocą dwóch cyfr. W funkcji *main* należy przetestować stworzone wyrażenia regularne.

Zadanie 10.3. Adres zamieszkania

Zdefiniuj wyrażenia regularne pozwalające na sprawdzenie:

- ulicy (nazwa może składać się tylko z liter alfabetu łacińskiego oraz znaku spacji);
- kodu pocztowego (w formacie *XX-XXX*, gdzie *X* to cyfra);
- numeru domu (liczba lub liczba i litera, np. 28, 28B);
- numer mieszkania

W funkcji *main* należy przetestować stworzone wyrażenia regularne.

Zadanie 10.4. Kartoteka

Zdefiniuj wyrażenia regularne pozwalające na sprawdzenie:

- imienia (imię może składać się tylko z liter alfabetu łacińskiego);
- nazwisko (nazwisko może składać się z liter alfabetu łacińskiego oraz myślnika przy nazwiskach łączonych, na przykład: Nowak-Kowalska);
- wieku (liczba od 0 do 99);
- numeru telefonu komórkowego (numer telefonu składa się z 9 cyfr, przy czym pierwsza cyfra nie może być 0);
- emaila (email postaci *u@x.y*, gdzie *u* to nazwa użytkownika, która może składać się z liter alfabetu łacińskiego, cyfr oraz znaków: podkreślenie, myślnik, kropka oraz nie może zaczynać się od cyfry, znaku podkreślenia, kropki oraz myślnika, *x.y* to nazwa domeny, która może składać się z liter alfabetu łacińskiego oraz cyfr);

Napisz program, który będzie wczytywał od użytkownika: imię, nazwisko, wiek, numer telefonu komórkowego oraz email i sprawdzał za pomocą zdefiniowanych wyrażeń regularnych czy podane dane są prawidłowe. Jeśli użytkownik podał prawidłowe dane to należy zapisać je do pliku. Jeśli plik istnieje to dane należy dopisać na koniec. W jednej linijce powinna zostać zapisana informacja o jednej osobie, gdzie dane powinny zostać rozdzielone średnikiem.

Zadanie 10.5. Statystyka

Wykorzystaj plik stworzony w zadaniu 10.4. i na jego podstawie wyświetl następujące informacje:

- nazwy unikatowych domen;
- numery telefonów, które kończą się liczbą parzystą;
- nazwiska łączone;
- statystykę imion (imię + liczba wystąpień).

LABORATORIUM 11. ZARZĄDZANIE PAMIĘCIĄ. STOSOWANIE INTELIGENTNYCH WSKAŹNIKÓW.

Cel laboratorium:

Omówienie inteligentnych wskaźników.

Zakres tematyczny zajęć:

- zarządzanie pamięcią,
- inteligentne wskaźniki.

Pytania kontrolne:

1. Jakie są błędy związane w zarządzaniem pamięcią?
2. Na czym polega technika RAII?
3. Co to są inteligentne wskaźniki?
4. Jaka jest różnica pomiędzy *unique_ptr* a *shared_ptr*?
5. Kiedy trzeba stosować *weak_ptr*?

Zarządzanie pamięcią:

W języku C++ można pracować ze wskaźnikami. Do najczęstszych błędów związanych z pracą ze wskaźnikami zaliczane są wycieki pamięci (ang. memory leak). Wyciek pamięci pojawia się wtedy, kiedy wszystkie wskaźniki, które pokazywały na fragment pamięci przestały istnieć, a pamięć nie została zwolniona. Kolejnym błędem, który pojawia się, jest odwołanie do pamięci lub próba zwolnienia pamięci, której adres jest nieprawidłowy, na przykład: użycie wskaźnika po zwolnieniu nieprzydzielonej mu pamięci lub ponowne zwolnienie pamięci, która już została zwolniona. Technika RAII pomaga w rozwiązywaniu wspomnianych problemów.

Technika RAII:

RAII (ang. Resource acquisition in initialization) czyli inicjalizowanie przy pozyskaniu zasobu. Jest wzorcem projektowym i polega na łączeniu przejęcia i zwolnienia zasobu z inicjowaniem i usuwaniem zmiennych. Destruktor wywoływany jest automatycznie więc, gdy zmienna wyjdzie po za swój zasięg, zasób zostanie zwolniony od razu, również w przypadku pojawienia się wyjątków. Technika RAII umożliwia pisanie kodu odpornego na błędy. Dynamicznie przydzielona pamięć za pomocą operatora *new* może być kontrolowana za pomocą techniki RAII przy użyciu inteligentnych wskaźników (ang. smart pointers).

Inteligentne wskaźniki:

Zadaniem inteligentnych wskaźników jest zwolnienie przydzielonej pamięci obiektowi w momencie kiedy zostanie usunięty ostatni wskazujący na obiekt wskaźnik. Klasy do obsługi inteligentnych wskaźników wymagają dołączenia pliku nagłówkowego *memory*. Po zainicjalizowaniu inteligentnego wskaźnika, inteligentny wskaźnik jest właścicielem surowego wskaźnika (ang. raw pointer). Inteligentny wskaźnik jest odpowiedzialny za usunięcie pamięci określonej przez wskaźnik surowy. Destruktor inteligentnego wskaźnika wywołuje usunięcie (operator *delete*). Użycie inteligentnych wskaźników jest prawie takie samo jak używanie

surowych wskaźników. Różnica występuje przy tworzeniu inteligentnego wskaźnika. Istnieje kilka implementacji inteligentnych wskaźników:

- *unique_ptr* – implementuje wyłączną własność obiektu. Zapewnia istnienie dokładnie jednego wskaźnika do obiektu. Pozwala na przenoszenie własności;
- *shared_ptr* – implementuje współdzieloną własność obiektu. Umożliwia na istnienie wielu wskaźników do tego samego obiektu. Posiada licznik referencji, który jest zwiększany kiedy tworzona jest kopia oraz zmniejszany kiedy kopia jest niszczone. Zwalnia obiekt w momencie zwalniania ostatniego wskaźnika;
- *weak_ptr* – implementuje współdzielenie dostępu do obiektu ale bez odpowiedzialności za jego zwolnienie. Klasa ta używana jest w szczególnych przypadkach kiedy nie może zostać użyta klasa *shared_ptr*.

Klasa *unique_ptr*:

Przy pracy ze wskaźnikami *unique_ptr* pomocne mogą okazać się następujące funkcje:

- *get* – zwraca adres wskaźnika;
- *move* – przenosi wszystkie dane z obiektu źródłowego, a obiekt źródłowy pozostawia pusty;
- *reset* – niszczy obiekt.

Wskaźniki *unique_ptr* obsługują również typ tablicowy oraz kontenery STL. Posiadają przeciążony operator []. Jeśli klasa ma do czynienia z tablicą wtedy w destruktorze wywoła się nie *delete* a *delete []*. Klasa *unique_ptr* zapewnia istnienie dokładnie jednego wskaźnika do obiektu. Nie pozwala na kopiowanie. Używając wskaźników *unique_ptr* można zdefiniować własny delokator, który będzie odpowiedzialny za zwolnienie pamięci. Na listingu 11.1. zaprezentowany został przykład użycia *unique_ptr*. Listing 11.1. zawiera:

- linijka 2 – dołączenie biblioteki;
- linijki 4 – 7 – funkcja, która modyfikuje oryginalną wartość;
- linijka 10 – stworzenie wskaźnika na liczbę całkowitą typu *unique_ptr*;
- linijka 11 – odwołanie się do przechowywanej wartości poprzez operator *;
- linijka 13 – pozyskanie adresu na jaki wskazuje wskaźnik;
- linijki 15 – 21 – przeniesienie całego obiektu do drugiego. Pierwszy obiekt wskazuje na *NULL* ponieważ, dwa obiekty nie mogą wskazywać na ten sam obiekt;
- linijki 22-23 – modyfikacja wartości, na którą wskazuje wskaźnik. Do funkcji należy przekazać oryginał, ponieważ *unique_ptr* uniemożliwia kopiowanie;
- linijki 24 – 26 – niszczenie obiektu.

```
1  #include <iostream>
2  #include <memory>
3  using namespace std;
4  void funcUniqueModify(unique_ptr<int> &uptr)
5  {
6      *uptr=112;
7  }
8  int main()
9  {
10     unique_ptr<int> uptr1(new int);
11     *uptr1=12;
12     cout<<"Wartosc uptr1: "<<*uptr1<<endl;
```



```

13  cout<<"Adres uptr1: "<<uptr1.get()<<endl;
14  unique_ptr<int> uptr2;
15  cout<<"Przed move"<<endl;
16  cout<<"Adres uptr1: "<<uptr1.get()<<endl;
17  cout<<"Adres uptr2: "<<uptr2.get()<<endl;
18  uptr2=move(uptr1);
19  cout<<"Po move: "<<endl;
20  cout<<"Adres uptr1: "<<uptr1.get()<<endl;
21  cout<<"Adres uptr2: "<<uptr2.get()<<endl;

22  funcUniqueModify(uptr2);
23  cout<<"Wartosc uptr2: "<<*uptr2<<endl;

24  cout<<"Przed reset: Adres uptr2: "<<uptr2.get()<<endl;
25  uptr2.reset();
26  cout<<"Po reset: Adres uptr2: "<<uptr2.get()<<endl;
27  return 0;
28 }

```

Listing 11.1. Przykłady użycia *unique_ptr*

Jak można zauważyć w powyższym kodzie, nigdzie nie został jawnie wywołany operator *delete*, za to są odpowiedzialne inteligentne wskaźniki. Na rysunku 11.1. przedstawiony został wynik działania kodu z listingu 11.1. Jak można zauważyć, zmienna *uptr1* ma adres 0x1c1770 przed wywołaniem metody *move*, a potem wskazuje na *NULL*. Natomiast zmienna *uptr2* przed wywołaniem metody *move* wskazuje na *NULL* a potem wskazuje na adres, na który wskazywała wcześniej zmienna *uptr1*. Dzieje się tak, ponieważ *move* przenosi wszystkie dane z obiektu źródłowego, a obiekt źródłowy pozostawia pusty. Po wywołaniu metody *reset* zmienna *uptr2* wskazuje na *NULL*.

```

Wartosc uptr1: 12
Adres uptr1: 0x1c1770
Przed move
Adres uptr1: 0x1c1770
Adres uptr2: 0
Po move:
Adres uptr1: 0
Adres uptr2: 0x1c1770
Wartosc uptr2: 112
Przed reset: Adres uptr2: 0x1c1770
Po reset: Adres uptr2: 0

```

Rys. 11.1. Wynik działania kodu z listingu 11.1.

Na listingu 11.2. przedstawione zostały funkcje, które zostaną użyte na listingu 11.3. Listing 11.2. zawiera:

- linijki 1 – 5 – funkcja, która będzie pełniła funkcje własnego delokatora;
- linijki 6 – 12 – funkcja, która tworzy tablicę, wypełnia ją liczbami i zwraca wskaźnik do stworzonej tablicy.

```

1  void funUniqueDeleter(int* p)
2  {
3      delete[] p;
4      cout<<"funUniqueDeleter: usunieta tablica"<<endl;

```



```

5  }
6  unique_ptr<int[]> funUniqueArray(int n)
7  {
8      unique_ptr<int[]> arr(new int[n]);
9      for(int i=0; i<n; i++)
10         arr[i]=i+7;
11
12     return arr;
13 }

```

Listing 11.2. Funkcje, które zostaną użyte na listingu 11.3.

Listing 11.3. przedstawia, w jaki sposób stworzyć tablicę. Listing 11.3. zawiera:

- linijka 1 – stworzenie tablicy przechowującej 4 liczby całkowite;
- linijka 8 – stworzenie tablicy przechowującej 5 liczb całkowitych oraz wypełnienie jej liczbami. Funkcja *funUniqueArray* zwraca wskaźnik typu *unique_ptr*;
- linijka 12 – stworzenie tablicy przechowującej 10 liczb całkowitych. Jako drugi argument do konstruktora została przekazana funkcja pełniąca rolę funktora *funUniqueDeleter*, który, usuwając tablicę, wyświetli dodatkowo napis „funUniqueDeleter: usunięta tablica”. Przy pracy ze wskaźnikami *unique_ptr* należy podawać również jawne dookreślenia drugiego parametru szablonu, w tym przypadku jest to *void(*)int(*)*;
- linijka 14 – stworzenie kontenera *vector*, przechowującego wskaźniki *unique_ptr*;
- linijka 17 – dodanie elementu do kontenera, metoda *move* jest niezbędna ponieważ, kopiowanie jest zabronione;
- linijka 18 – odwołanie się do wartości znajdującej się pod indeksem 0.

```

1  unique_ptr<int[]> tab1(new int[4]);
2  cout<<"Elementy tablicy tab1:"<<endl;
3  for(int i=0; i<4; i++)
4  {
5      tab1[i]=i+7;
6      cout<<tab1[i]<<endl;
7  }
8  unique_ptr<int[]> tab3=funUniqueArray(5);
9  cout<<"Elementy tablicy tab3:"<<endl;
10 for(int i=0; i<5; i++)
11     cout<<tab3[i]<<endl;
12 unique_ptr<int[],void(*) (int*)>
13     tab2(new int[10],funUniqueDeleter);
14
15 vector<unique_ptr<int>> vec;
16 unique_ptr<int> i1(new int);
17 *i1=3;
18 vec.push_back(move(i1));
19 cout<<"Element w wektorze: "<<*vec[0]<<endl;

```

Listing 11.3. Przykłady tablic



Klasa `shared_ptr`:

Przy pracy ze wskaźnikami `shared_ptr` pomocne mogą okazać się następujące funkcje:

- `get` – zwraca adres wskaźnika;
- `reset` – niszczy obiekt i może utworzyć nowy;
- `use_count` – zwraca liczbę wskaźników aktualnie wskazujących na daną wartość.

Klasa `shared_ptr` umożliwia istnienie wielu wskaźników do tego samego obiektu używając licznik do zliczania ile wskaźników aktualnie wskazuje na daną wartość. Kiedy ostatni ze wskaźników przestanie istnieć i licznik będzie równy 0 wtedy następuje zwolnienie pamięci. Operacje kopiowania są dozwolone. Klasa `shared_ptr` umożliwia przechowywanie wskaźników w kontenerach STL. Natomiast klasa wspiera tablice dopiero od standardu C++17, wcześniej było to niemożliwe. Podczas tworzenia obiektu konstruktor tworzy licznik referencji i inicjalizuje go wartością 1. Klasa `shared_ptr` umożliwia implementację własnego delokatora. Listingi 11.4. – 11.6. przedstawiają przykład użycia klasy `shared_ptr`. Zostanie zaprezentowany następujący przykład. W biurze pracują dwie osoby przez 5 dni w tygodniu. Dla poszczególnych dni zostanie zapisana informacja, kto pierwszy przyszedł danego dnia do biura. Zostanie stworzona klasa `Person` oraz kontener przechowujący obiekty klasy `Person`. Listing 11.4. zawiera definicję klasy `Person`. Klasę `Person` należy dodać do pliku `main.cpp`, w celach dydaktycznych aby nie tworzyć kolejnych plików: nagłówkowego i źródłowego dla klasy `Person`. Klasa `Person` zawiera dwa pola, jeden konstruktor oraz jedną metodę.

```

1  class Person{
2  private:
3      string name;
4      int age;
5  public:
6      Person(string name1, int age1){
7          name=name1;
8          age=age1;
9      }
10     void info(){
11         cout<<name<<" "<<age<<endl;
12     }
13 };

```

Listing 11.4. Klasa `Person`

Listing 11.5. zawiera funkcję, która będzie potrzebna na listingu 11.6.

```

1  void funDeleter(Person* p)
2  {
3      cout<<"usunieta osoba: ";
4      p->info();
5      delete p;
6  }

```

Listing 11.5. Funkcja `funDeleter`

Na listingu 11.6. przedstawione zostały przykłady użycia wskaźników `shared_ptr`. Kod należy dodać do funkcji `main`. Listing 11.6. zawiera:

- linijki 1 – 2 – stworzenie dwóch wskaźników wraz z wywołaniem konstruktora;
- linijka 3 – wywołanie metody;
- linijki 4 – 5 – wyświetlenie wartości liczników;
- linijka 6 – stworzenie kontenera *vector*, który będzie przechowywał wskaźniki *shared_ptr* na obiekty klasy *Person*;
- linijki 7 – 11 – dodanie obiektów do kontenera;
- linijki 15 – 16 – wywołanie metody *info* dla wszystkich obiektów w kontenerze;
- linijka 18 – zmiana liczby elementów w kontenerze;
- linijki 22 – 23 – stworzenie wskaźnika wraz z przekazaniem własnego funktora/funkcji, który będzie pełnić funkcję delokatora. Przy pracy z *shared_ptr* nie trzeba doprecyzowywać typu tak jak było to przy pracy z *unique_ptr*;

```

1  shared_ptr < Person > sptr1( new Person("Ola", 32) );
2  shared_ptr < Person > sptr2( new Person("Ula", 52) );
3  sptr1->info();
4  cout<<"Licznik sptr1: "<<sptr1.use_count()<<endl;
5  cout<<"Licznik sptr2: "<<sptr2.use_count()<<endl;

6  vector<shared_ptr<Person>> firstInOffice;
7  firstInOffice.push_back(sptr1);
8  firstInOffice.push_back(sptr2);
9  firstInOffice.push_back(sptr2);
10 firstInOffice.push_back(sptr1);
11 firstInOffice.push_back(sptr2);

12 cout<<"Po dodaniu do kontenera"<<endl;
13 cout<<"Licznik sptr1: "<<sptr1.use_count()<<endl;
14 cout<<"Licznik sptr2: "<<sptr2.use_count()<<endl;

15 for (shared_ptr<Person> ptr : firstInOffice)
16     ptr->info();
17 cout << endl;

18 firstInOffice.resize(3);

19 cout<<"Po resize"<<endl;
20 cout<<"Licznik sptr1: "<<sptr1.use_count()<<endl;
21 cout<<"Licznik sptr2: "<<sptr2.use_count()<<endl;

22 shared_ptr <Person> sptr3(new
23     Person("Magda", 22), funDeleter);
24 sptr3->info();
    
```

Listing 11.6. Przykłady użycia *shared_ptr*

Na rysunku 11.2. przedstawiony został wynik działania kodu z listingu 11.6. Jak można zauważyć, w momencie, kiedy tworzony jest obiekt, wartość licznika ustawiana jest na 1. Następnie dwa razy została dodana zmienna *sptr1* do kontenera a zmienna *sptr2* została dodana trzy razy więc odpowiednio po tym, w licznikach znalazły się liczby 3 oraz 4. Po zmniejszeniu



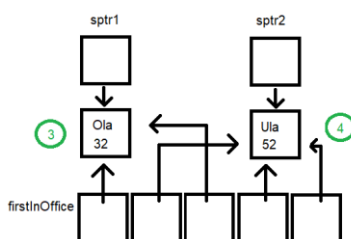
kontenera do trzech elementów, dwa ostatnie zostały usunięte, więc zmieniły się wartości przechowywane w licznikach na odpowiednio 2 i 3. Przy tworzeniu zmiennej *sptr3* podana została funkcja, która pełniła funkcję delokatora. Przy usuwaniu obiektu *sptr3* wywołany został podany funktor.

```
Ola 32
Licznik referencji sptr1: 1
Licznik referencji sptr2: 1
Po dodaniu do kontenera
Licznik referencji sptr1: 3
Licznik referencji sptr2: 4
Ola 32
Ula 52
Ula 52
Ola 32
Ula 52

Po resize
Licznik referencji sptr1: 2
Licznik referencji sptr2: 3
Magda 22
usunieta osoba: Magda 22
```

Rys. 11.2. Wynik działania kodu z listingu 11.6.

Rysunek 11.3. przedstawia interpretację graficzną prezentowanego przykładu. Na zielono zostały zaznaczone liczniki zmiennych *sptr1* oraz *sptr2*.



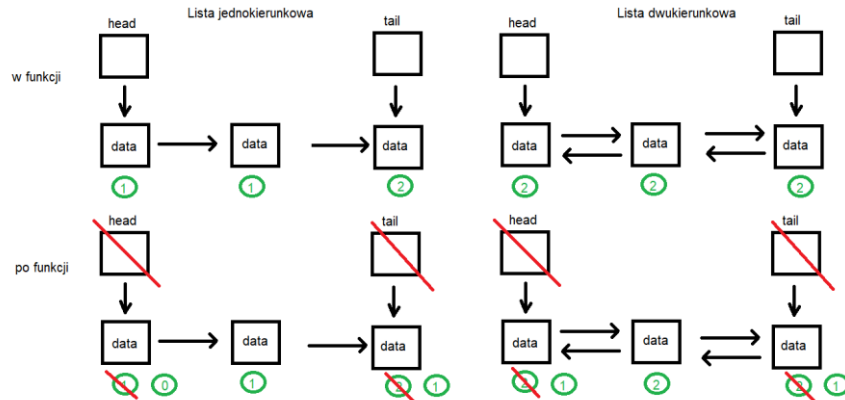
Rys. 11.3. Graficzna prezentacja przykładu z listingu 11.6.

Klasa *weak_ptr*:

Wskaźniki *weak_ptr* używane są po to aby zapobiec cyklicznemu odwołaniu. Taka sytuacja występuje na przykład przy implementacji listy dwukierunkowej: dwa obiekty odwołują się do siebie nawzajem i nigdy nie dojdzie do zwolnienia zasobów tych obiektów. Przy liście jednokierunkowej nie będzie takiego problemu. Obiekty klasy *weak_ptr* nie uczestniczą w zliczaniu referencji. Przed użyciem wskaźnika *weak_ptr* należy sprawdzić, czy wartość na którą wskazuje dalej istnieje. Wskaźnik *weak_ptr* nie jest właścicielem obiektu a tylko obserwatorem. Kiedy ostatni wskaźnik *shared_ptr* przestanie istnieć, usunięty zostanie także obiekt a *weak_ptr* nie będzie zawierał wskaźnika na zwolnioną pamięć i będzie zawierać adres *NULL*. Aby uzyskać dostęp do wskazywanego obiektu konieczna jest konwersja do *shared_ptr*. Można to wykonać na dwa sposoby: albo użyć metody *lock* albo wykorzystać konstruktor *shared_ptr*.

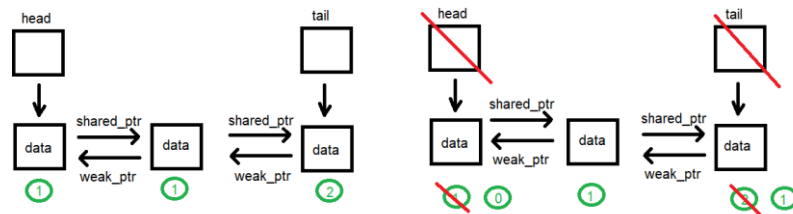
Rysunek 11.4. przedstawia problem oraz brak problemu cyklicznego odwołania. Załóżmy, że obie listy tworzone są w jakiejś funkcji. Obie listy zawierają wskaźniki *head* i *tail* które są typu *shared_ptr*. Wskaźnik *head* zawiera początek listy, a *tail* koniec. Na zielono zostały zaznaczone wartości liczników dla każdego elementu. Po wyjściu z funkcji zmienne *head* i *tail* są usuwane, co powoduje zmianę wartości liczników. Przeanalizujemy sytuację dla listy jednokierunkowej: dla pierwszego elementu licznik jest równy 0 więc element zostanie usunięty, a więc i kolejny element zostanie usunięty bo dla kolejnego elementu wartość licznika

zostanie zdekrementowana i będzie równa 0, co w konsekwencji spowoduje dekrementację licznika dla kolejnego ostatniego elementu. W konsekwencji cała lista zostanie usunięta. Teraz zobaczmy jak sytuacja wygląda dla listy dwukierunkowej: dla pierwszego elementu licznik przechowuje wartość 1 a więc nie może być on usunięty, co blokuje kolejne usuwanie.



Rys. 11.4. Problem i brak problemu cyklicznego odwołania

Na rysunku 11.5. przedstawione zostało rozwiązanie problemu z rysunku 11.4. dla listy dwukierunkowej. Rozwiązaniem problemu cyklicznego odwołania jest wprowadzenie jednego wskaźnika *shared_ptr* a drugiego *weak_ptr*. Na rysunku wskaźnik wskazujący na następne pole jest typu *shared_ptr* a wskaźnik wskazujący na poprzednie pole jest typu *weak_ptr*. Jak wiemy obiekt klasy *weak_ptr* nie uczestniczy w zliczaniu i dzięki temu sytuacja sprowadza się do tej, która była przy liście jednokierunkowej.



Rys. 11.5. Rozwiązanie problemu cyklicznego odwołania

Listing 11.7. przedstawia sytuację powodującą cykliczne odwołanie. Klasa *Elem* zawiera dwa wskaźniki: *next* oraz *prev*, które są typu *shared_ptr*. Pola są publiczne aby nie kompilkować przykładu, poprzez dodawanie getterów i seterów do pól.

```

1  class Elem {
2  public:
3      shared_ptr<Elem> next;
4      shared_ptr<Elem> prev;
5      ~Elem() {
6          cout << "Destruktor Elem" << endl;
7      }
8  };

```

Listing 11.7. Klasa Elem

Kod przedstawiony na listingu 11.8. należy dodać do funkcji *main*. Obiekt *el1* wskazuje na obiekt *el2*, a obiekt *el2* wskazuje na obiekt *el1*.

```
1    shared_ptr<Elem> el1 (new Elem);
2    shared_ptr<Elem> el2 (new Elem);
3    el1->next = el2;
4    el2->prev = el1;
```

Listing 11.8. Przykład cyklicznego odwołania

Po uruchomieniu programu na konsoli nie wyświetla się komunikat umieszczony w destruktorze. Jeśli obiekt byłby zniszczony wtedy wywołał by się destruktor. Oznacza to, że mamy do czynienia z cyklicznym odwołaniem. Teraz zmienimy w linijce 4 z listingu 11.7. *shared_ptr* na *weak_ptr* resztę pozostawiając bez zmiany. Po uruchomieniu programu na konsoli dwa razy wyświetli się komunikat z destruktora. Aby mieć dostęp do przechowywanej wartości, należy skonwertować *weak_ptr* do *shared_ptr*. Na listingu 11.9. zostało przedstawione w jaki sposób można to zrobić.

```
1    shared_ptr<Elem> tempEl(el2->prev);
2    cout <<"Adres: " <<tempEl<< endl;
```

Listing 11.9. Odwołanie się do zmiennej typu *weak_ptr*

Zadania do wykonania:

Zadanie 11.1.Bufor

W zadaniu wykorzystaj inteligentne wskaźniki i pamiętaj, aby wszystkie zasoby zostały poprawnie zwolnione.

Zdefiniuj klasę abstrakcyjną *Buffer*. Klasa posiada:

- funkcję czysto wirtualną *void add(int a)*;
- funkcję czysto wirtualną *void write()*;
- konstruktor bezparametrowy wypisujący na ekranie napis "Konstruktor Buffer"
- destruktor wypisujący na ekranie napis "Destruktor Buffer"

Stwórz dwie klasy dziedziczące publicznie po tej klasie: *BufferArr* oraz *BufferFile*.

Klasa *BufferArr* posiada:

- prywatne pola do reprezentowania jednowymiarowej tablicy elementów o rozmiarze podanym w parametrze;
- konstruktor jednoargumentowy alokujący tablicę o rozmiarze podanym w parametrze;
- odpowiednie gettery oraz settery do pól;
- metodę *add(int a)*, która dodaje przekazany w parametrze element na kolejną pozycję w tablicy, w przypadku gdy tablica jest pełna i nie można dodać do niej elementu, na ekranie powinien pojawić się odpowiedni komunikat;
- napisz metodę *write()*, która wyświetla wszystkie dodane elementy na ekranie;

Konstruktory oraz destruktor dodatkowo wypisują na ekranie napis identyfikujący je: „Konstruktor klasy *BufferArr*” oraz „Destruktor klasy *BufferArr*”.



Klasa *BufferFile* posiada:

- prywatne pole do przechowywania ścieżki do pliku;
- konstruktor jednoargumentowy pozwalający na inicjalizację pola w klasie przyjmujący nazwę ścieżki jako parametr;
- metodę *add(int a)*, która dopisuje do pliku kolejną liczbę. W jednej linijce znajduje się jedna liczba;
- metodę *write()*, która wyświetla wszystkie liczby z pliku na konsolę;

Konstruktor oraz destruktor dodatkowo powinny wypisywać na ekranie napis identyfikujący je: „Konstruktor klasy *BufferFile*” oraz „Destruktor klasy *BufferFile*”. Uchwyty pozwalające na otworzenie pliku powinny być wskaźnikami.

Napisz program, w którym przetestujesz polimorficzne wywołanie metod wirtualnych. Napisz funkcję *main()*, która posiada 6 „buforów” umieszczonych w kontenerze lub tablicy.

Zadanie 11.2. Sklepy

W zadaniu wykorzystaj inteligentne wskaźniki. Zdefiniuj klasę *Warehouse*, która będzie reprezentować magazyn. Klasa będzie posiadać:

- prywatne pola przechowujące nazwę towaru, który jest przechowywany w magazynie (dla uproszczenia na przykład: książki, komputery, łóżka, itd.) oraz liczbę sztuk;
- konstruktor inicjalizujący pola w klasie;
- odpowiednie setery oraz gettery do pól;
- metodę wyświetlającą informację o magazynie (nazwa towaru oraz liczba sztuk towaru).

Następnie stwórz klasę *Shop*, która będzie reprezentować sklep i posiadać:

- prywatne pole: nazwa sklepu, listę magazynów (*vector*), z których korzysta sklep;
- konstruktor jednoparametrowy pozwalający na inicjalizację nazwy sklepu;
- metodę dodającą magazyn do listy;
- metodę, która umożliwi sprzedaż towaru, którego nazwa i liczba sztuk została podana w argumencie (zakładamy, że nazwa artykułu jest taka sama jak nazwa towaru w magazynie). Jeśli w danym magazynie znajduje się odpowiednia liczba sztuk danego towaru, to towar powinien zostać sprzedany;
- metodę wyświetlającą z jakich magazynów korzysta sklep;
- metodę wyświetlającą nazwę sklepu.

W destruktorach stworzonych klas należy wyświetlić komunikat identyfikujący te destruktory.

W funkcji *main* stwórz listę unikatowych sklepów. Sklepy mogą korzystać z tych samych magazynów. Przetestuj stworzone klasy.

Zadanie 11.3. Lista dwukierunkowa

Napisz program, który w oparciu o inteligentne wskaźniki będzie realizować uproszczoną listę dwukierunkową. Operacje, które będzie można wykonać na liście, to dodawanie na początek oraz na koniec listy, usuwanie z początku i z końca, wyświetlenie zawartości listy, sprawdzenie czy jest pusta. Program należy przetestować. W destruktorach stworzonych klas należy wyświetlić komunikat identyfikujący te destruktory.

LABORATORIUM 12. WYRAŻENIA I FUNKCJE LAMBDA. KROTKI W C++.

Cel laboratorium:

Omówienie wyrażeń lambda oraz krotek.

Zakres tematyczny zajęć:

- wyrażenia lambda,
- krotki.

Pytania kontrolne:

1. Co to jest krotka?
2. W jaki sposób dodać nowy element do krotki?
3. Co to jest wyrażenie lambda?
4. Jakie są sposoby przechwytywania w funkcjach lambda?

Krotki:

Krotka (ang. tuple) to kontener, który może zawierać elementy różnych typów danych, na przykład liczby całkowite, znaki, napisy, obiekty innych klas. Można powiedzieć, że krotki są uogólnieniem szablonu *pair*, który umożliwia przechowanie dwóch wartości. Elementy w krotce przechowywane są w kolejności dodania. Krotki umożliwiają łatwe zwracanie wielu wartości z funkcji oraz wiązanie ich ze zmiennymi. W celu korzystania z krotek należy dołączyć plik nagłówkowy *tuple*. Poniżej wymienione zostały funkcje, które mogą się przydać przy pracy z krotkami:

- *make_tuple* – tworzenie krotki;
- *forward_as_tuple* – przekazanie krotki do funkcji;
- *tuple_cat* – dodawanie krotek (konkatenacja);
- *tie* – powiązanie zmiennych z elementami krotki;
- *get* – pobranie elementu z krotki.

Dodatkowo biblioteka *tuple* udostępnia dwie klasy pomocnicze:

- *tuple_size* – umożliwia sprawdzenie, ile elementów znajduje się w krotce;
- *tuple_element* – umożliwia pobranie elementu z krotki i zapisanie do zmiennej.

Listing 12.1. przedstawia przykład użycia krotki. Listing 12.1. zawiera:

- linijka 2 – dołączenie pliku nagłówkowego;
- linijki 6 – 7 – stworzenie krotki;
- linijki 8 – 9 – wyświetlenie liczby elementów znajdujących się w krotce;
- linijki 10 - 12 - dostęp do poszczególnych elementów krotki. Przekazany indeks musi być *const*, a więc nie jest możliwe wyświetlenie elementów w pętli;
- linijki 13 – 18 – zapisanie do zmiennej elementu znajdującego się w krotce;
- linijki 19 – 22 – modyfikacja elementu znajdującego się w krotce;
- linijki 23 – 32 – drugi sposób tworzenia krotek. Zamiana elementów krotek między sobą. Krotki muszą przechowywać elementy o odpowiadających sobie typach;
- linijki 33 – 38 – trzeci sposób tworzenia krotki, za pomocą typu *auto*. Dodawanie nowego elementu lub elementów do istniejącej krotki odbywa się za pomocą łączenia

dwóch krotek. Aby dodać nowy element do krotki trzeba stworzyć pomocniczą krotkę następnie użyć funkcji do łączenia;

- linijki 39 – 48 – zapisanie do zmiennych wartości z krotek. Za pomocą *ignore* można pominąć niektóre elementy.

```

1  #include <iostream>
2  #include <tuple>
3  using namespace std;
4  int main()
5  {
6      tuple <char, int, float> t1;
7      t1 = make_tuple('a', 8, 21.5);

8      cout << "Liczba elementow: "
9           << tuple_size<decltype(t1)>::value << endl;

10     cout <<"get<0>: "<< get <0>(t1) << endl;
11     cout <<"get<1>: "<< get <1>(t1) << endl;
12     cout <<"get<2>: "<< get <2>(t1) << endl;

13     tuple_element<0,decltype(t1)>::type
14         first = get<0>(t1);
15     tuple_element<1,decltype(t1)>::type
16         second = get<1>(t1);
17     cout << "Pierwszy element t1: " << first << endl;
18     cout << "drugi element t1: " << second << endl;

19     cout<<"Przed modyfikacja: get<1>: "
20         <<get<1>(t1)<<endl;
21     get<1>(t1)=2;
22     cout<<"Po modyfikacji: get<1>: "<<get<1>(t1)<<endl;

23     tuple <char,int,float> t2('p',45,7.5);
24     cout<<"Przed swap t2"<<endl;
25     cout <<"get<0>: "<< get <0>(t2) << endl;
26     cout <<"get<1>: "<< get <1>(t2) << endl;
27     cout <<"get<2>: "<< get <2>(t2) << endl;
28     t2.swap(t1);
29     cout<<"Po swap t2"<<endl;
30     cout <<"get<0>: "<< get <0>(t2) << endl;
31     cout <<"get<1>: "<< get <1>(t2) << endl;
32     cout <<"get<2>: "<< get <2>(t2) << endl;

33     tuple <int,char> t3(2,'a');
34     auto t4=make_tuple(3,'b');
35     auto t5 = tuple_cat(t3,t4);
36     cout<<"Zawartosc t5: "<<endl;
37     cout<<get<0>(t5)<<" "<<get<1>(t5)<<" "<<get<2>(t5)
38         <<" "<<get<3>(t5)<<endl;
    
```

```

39     int i_val;
40     char ch_val;
41     float f_val;
42     tuple <int,char,float> t6(2,'R',3.5);
43     tie(i_val,ch_val,f_val) = t6;
44     cout << "Pobranie wszystkich zmiennych z tupli: ";
45     cout << i_val << " " << ch_val << " " << f_val<<endl;
46     tie(i_val,ignore,f_val) = t6;
47     cout << "Pobranie wybranych zmiennych z tupli: ";
48     cout << i_val << " " << f_val<<endl;
49
50     return 0;
51 }

```

Listing 12.1. Przykład użycia krotek

Na rysunku 12.1. przedstawiony został wynik działania kodu z listingu 12.1.

```

Liczba elementow: 3
get<0>: a
get<1>: 8
get<2>: 21.5
Pierwszy element t1: a
drugi element t1: 8
Przed modyfikacja: get<1>: 8
Po modyfikacji: get<1>: 2
Przed swap t2
get<0>: p
get<1>: 45
get<2>: 7.5
Po swap t2
get<0>: a
get<1>: 2
get<2>: 21.5
Zawartosc t5:
2 a 3 b
Pobranie wszystkich zmiennych z tupli: 2 R 3.5
Pobranie wybranych zmiennych z tupli: 2 3.5

```

Rys. 12.1. Wynik działania kodu z listingu 12.1

Funkcje lambda:

Funkcje lambda są to anonimowe funkcje, które można definiować lokalnie. Z założenia funkcje lambda tworzone są na krótkotrwały użytek. Ich składnia jest następująca: *[przechwytywane nazwy](parametry funkcji) -> typ zwracany {ciało funkcji}* gdzie:

- przechwytywane nazwy mówią w jaki sposób funkcja będzie miała dostęp do zmiennych, jeśli nawiasy będą puste([]) będzie to oznaczać, że wszystkie dane funkcja dostaje poprzez argumenty wywołania (ma dostęp tylko do przekazanych zmiennych). Przekazywanie zmiennych z otoczenia do wyrażenia lambda nazywane jest ich przechwytywaniem. Przechwytywać zmienne (które są w zasięgu widzialności funkcji lambda) można na dwa sposoby: przez referencję lub przez wartość. Zmienne, które zostały przekazane przez wartość są traktowane jako stałe. Zapis [&] oznacza, że wszystko przechwycone zostało przez referencję, a zapis [=], że wszystko przechwycone zostało przez wartość. Można również łączyć sposoby przechwytywania;

- parametry funkcji to lista parametrów przekazywana w taki sam sposób jak do zwykłej funkcji;
- typ zwracany to typ jaki zwraca funkcja, można go pominąć jeśli występuje jeden zwracany typ;
- ciało funkcji to kod funkcji;

Funkcje lambda pozwalają zdefiniować obiekt funkcyjny w miejscu użycia. Wyrażenia lambda są stosowane głównie przy wykorzystaniu biblioteki *algorithms*, na przykład: *find_if*, *count_if*. Można je wykorzystać wszędzie tam, gdzie potrzebna jest funkcja/obiekt funkcyjny, który sprawdza kryterium. Listing 12.2. prezentuje przykład działania funkcji lambda dla algorytmów z biblioteki *algorithms*. Listing 12.2. zawiera:

- linijki 5 – 8 – stworzenie funkcji, która będzie pełniła funkcję komparatora, tak, jak było to dotychczas;
- linijki 9 – 15 – funkcja umożliwiająca wyświetlenie zawartości kontenera *vector*;
- linijka 19 – algorytm *sort*, którego trzecim argumentem jest funkcja *fn*, dane zostaną posortowane malejąco;
- linijka 20 – realizacja tego, co jest w linijce 19 ale za pomocą funkcji lambda. Funkcja lambda przyjmuje dwie liczby i porównuje je ze sobą zwracając wartość logiczną. Funkcja nie ma dostępu do innych zmiennych niż te, które zostały przekazane jako parametry ponieważ nawiasy kwadratowe są puste;
- linijka 21 – realizuje to samo co linijka 20 ale w tym przypadku zwracany typ nie został jawnie określony, robi to kompilator;
- linijki 25 – 26 – algorytm *transform*, każdy element kontenera jest zwiększany dwukrotnie;
- linijki 28 – 29 – algorytm *for_each* wyświetla element kontenera;
- linijka 30 – algorytm *for_each*, za pomocą, którego każdy element w kontenerze jest zwiększany dwukrotnie. W związku z tym jako argument funkcji lambda przekazana została referencja;

```
1  #include <iostream>
2  #include<vector>
3  #include <algorithm>
4  using namespace std;
5  bool fn( int l, int r )
6  {
7      return l > r;
8  }
9  void printVec(vector < int > num)
10 {
11     cout<<endl<<"Zawartosc num: ";
12     for( auto it = num.begin(); it != num.end(); ++it )
13         cout << *it << " ";
14     cout<<endl;
15 }
16 int main()
17 {
18     vector < int > num= {1,2,3,4,5,6,7,8,9};
19     sort(num.begin(),num.end(), fn );
20     sort(num.begin(),num.end(),
```



```

21         []( int l, int r )->bool { return l > r ;});
22     sort(num.begin(),num.end() ,
23         []( int l, int r ){ return l > r ;});
24     printVec(num);

25     transform(num.begin(),num.end(),num.begin() ,
26         [](int el){ return el*2;});

27     cout<<"Zawartosc num: ";
28     for_each(num.begin(),num.end() ,
29         [](int el){cout<<"**"<<el;});

30     for_each(num.begin(),num.end() , [](int&el) {el=el*2;});
31     printVec(num);
32     return 0;
33 }

```

Listing 12.2. Przykład użycia funkcji lambda

Na rysunku 12.2. przedstawiony został wynik działania kodu z listingu 12.2.

```

Zawartosc num: 9 8 7 6 5 4 3 2 1
Zawartosc num: **18**16**14**12**10**8**6**4**2
Zawartosc num: 36 32 28 24 20 16 12 8 4

```

Rys. 12.2. Wynik działania kodu z listingu 12.2

Listing 12.3. przedstawia przykład przechwytywania zmiennych otoczenia. Kod, który został przedstawiony na listingu należy dołączyć do funkcji *main*. Listing 12.3. zawiera:

- linijka 2 – policzenie sumy elementów znajdujących się w kontenerze. Zmienna *sum* z otoczenia została przekazana przez referencję, a więc modyfikacja jej w funkcji lambda będzie widoczna po za funkcją. W tym przypadku zdecydowanie łatwiej użyć funkcję lambda niż przeciążony *operator()*;
- linijki 6 – 7 – policzenie sumy elementów mniejszych od *val*. Zmienna *sum* z otoczenia została przekazana przez referencję, ponieważ w środku będzie modyfikowana, natomiast zmienna *val*, została przekazana przez wartość więc z punktu widzenia funkcji lambda jest stała, służy tylko do porównania.

```

1  int sum=0;
2  for_each(num.begin(),num.end() , [&sum](int el) {sum+=el;});
3  cout<<"Suma to: "<<sum<<endl;
4  int val=20;
5  sum=0;
6  for_each(num.begin(),num.end() ,
7      [&sum,val](int el) {if(el<val) sum+=el;});
8  cout<<"Suma el<20 to: "<<sum<<endl;

```

Listing 12.3. Przykład przechwytywania zmiennych z otoczenia

Na listingu 12.4. przedstawiony został kolejny przykład zawierający funkcje lambda. Listing 12.4. zawiera:

- linijki 1 – 2 – funkcja zwracająca sumę dwóch liczb a i b , za definicją funkcji lambda nastąpiło od razu jej wywołanie dla liczb 1 i 8;
- linijki 3 – 4 – zapisanie funkcji lambda do zmiennej, funkcja staje się zmienną lokalną. W tym przypadku potrzebne jest określenie typu zwracającego, ponieważ mogą być zwrócone dwa różne typy (*int* lub *float*). Typ funkcji lambda jest automatycznie ustalany przez kompilator;
- linijka 5 – wywołanie funkcji lambda.

```
1 cout<<"Suma a+b: "
2   <<[]( int a, int b ) { return a + b; }( 1, 8 )<<endl;
3 auto lam=[](int a)->float { if( a > 0 ) return 1;
4                               return -1.0;  };
5 cout<<lam(2)<<endl;
```

Listing 12.4. Przykłady funkcji lambda

Funkcje lambda można przekazywać jako argument funkcji. W tym celu należy użyć typu *function*, do którego można skonwertować typ funkcji lambda. Typ *function* znajduje się w pliku nagłówkowym *functional*. Musi zawierać typ zwracany przez funkcję oraz typy argumentów, które są przekazywane do funkcji. Na listingach 12.5. oraz 12.6. przedstawiony został przykład przekazania funkcji lambda do funkcji. Funkcja na listingu 12.5. przyjmuje dwa argumenty. Pierwszy z nich to zmienna typu *function*, w tym przypadku funkcja lambda, a drugi argument to liczba całkowita. Zmienna f (funkcja lambda) wywoływana jest w linijce 4.

```
1 void fun( function < int(int)> f, int n )
2 {
3     for( int i = 0; i < n; ++i )
4         cout << "f(" << i << " ) = " << f(i) <<endl;
5 }
```

Listing 12.5. Funkcja, która jako argument przyjmuje funkcję lambda

Kod z listingu 12.6. należy umieścić w funkcji *main*. Na listingu 12.6. przedstawione zostało wywołanie zdefiniowanej wyżej funkcji. Jako pierwszy argument przekazana została funkcja lambda.

```
fun([]( int x ) { return x+2; }, 5);
```

Listing 12.6. Wywołanie funkcji z argumentem jako funkcja lambda

Na rysunku 12.3. przedstawiony został wynik działania kodu z listingów 12.5. oraz 12.6.

```
f(0) = 2
f(1) = 3
f(2) = 4
f(3) = 5
f(4) = 6
```

Rys. 12.3. Wynik działania kodu z listingów 12.5. oraz 12.6.

Zadania do wykonania:

Zadanie 12.1. Algorytmy

Napisz program, w którym stworzysz kontener *vector* i wypełnisz go liczbami. Następnie umożliwisz wykonanie na nim następujących algorytmów:

- wyświetlenie wszystkich elementów w taki sposób aby każdy z elementów oddzielony był od siebie znakiem |;
- policzenie średniej arytmetycznej elementów w kontenerze;
- policzenie ile elementów parzystych znajduje się w kontenerze;
- usunięcie elementów, które są ujemne;
- posortowanie elementów w kontenerze, tak aby najpierw znajdowały się elementy parzyste a następnie nieparzyste;
- zmodyfikowanie wartości każdego elementu poprzez nadpisanie go liczbą przeciwną;
- policzenie ile elementów jest mniejszych niż otrzymany argument;

Przy implementacji poszczególnych funkcjonalności trzeba skorzystać z algorytmów znajdujących się w bibliotece *algorithm*, jako komparator należy stworzyć funkcję lambda. W funkcji *main* należy przetestować algorytmy.

Zadanie 12.2. Samochody

Stwórz klasę *Car*, która będzie reprezentować samochód i będzie zawierać:

- prywatne pola umożliwiające przechowywanie modelu samochodu, roku produkcji oraz pojemności silnika;
- konstruktor inicjalizujący wszystkie pola w klasie;
- gettery oraz setery do pól;
- metodę wyświetlającą informację o samochodzie.

W funkcji *main*, należy stworzyć kontener, który będzie przechowywał obiekty klasy *Car*. Dodatkowo należy wyświetlić samochody:

- posortowanie rosnąco względem roku produkcji;
- posortowanie malejąco względem pojemności silnika.

Przy wyświetlaniu informacji o samochodzie, przed informacją o danym samochodzie powinien zostać wyświetlony licznik, który to samochód (1,2,3, itd.).

Zadanie 12.3. Statystyka

Napisz funkcję, która jako argument dostanie kontener przechowujący napisy (typ *string*). Funkcja powinna zwrócić następujące wartości:

- długość najkrótszego napisu;
- średnią długość napisu;
- najdłuższy napis.

Do szukania wyżej wymienionych wartości użyj funkcji lambda i dostępnych algorytmów. Wartości powinny zostać zwrócone jako krotka. W funkcji *main* należy wyświetlić wartości zwrócone przez funkcję.

LABORATORIUM 13. TESTOWANIE W C++.

Cel laboratorium:

Omówienie testowania w języku C++

Zakres tematyczny zajęć:

- testowanie,
- testy jednostkowe.

Pytania kontrolne:

1. Na czym polega testowanie?
2. Co to są testy jednostkowe?
3. Co to jest pokrycie kodu?

Testowanie:

Testowanie aplikacji jest procesem związanym z wytwarzaniem oprogramowania. Głównym celem testowania jest weryfikacja oraz walidacja oprogramowania. Testowanie pozwala sprawdzić, czy oprogramowanie jest zgodne ze specyfikacją i oczekiwaniami użytkownika. Dzięki testowaniu możliwe jest wykrycie błędów oprogramowania. Testowanie różni się od debugowania tym, że debugowanie jest to proces znajdowania i usuwania błędów, a testowanie to proces szukania błędów. Z reguły programiści odpowiedzialni są za proces debugowania a testerzy za testowanie. Przeprowadzanie testów jest częścią całego etapu testowania. W ramach tego laboratorium omówiony zostanie proces przeprowadzania testów. Wśród testów można wyróżnić kilka rodzajów i każdy z nich ma inny cel, na przykład:

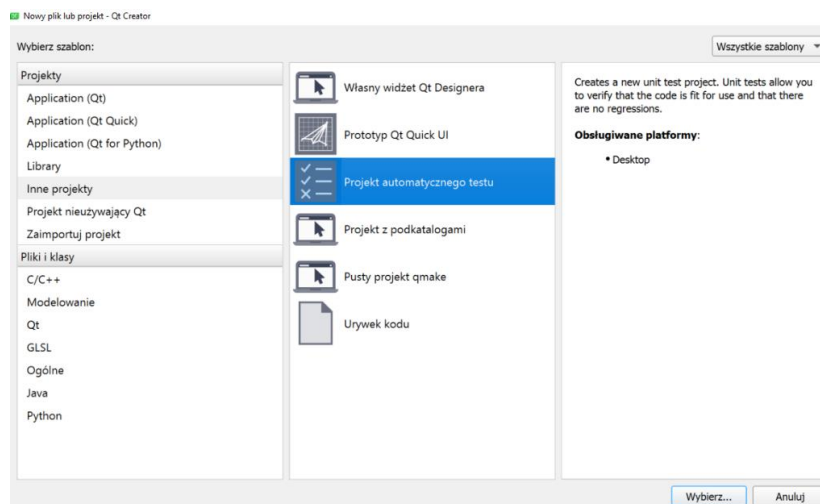
- testy jednostkowe – testowanie poprawności działania pojedynczych elementów;
- testy integracyjne – testowanie poprawności interfejsów oraz interakcji pomiędzy modułami;
- testy akceptacyjne – potwierdzenie działania systemu zgodnie z oczekiwaniami i wymaganiami.

Testy jednostkowe:

Testy jednostkowe (ang. unit test) weryfikują poprawność działania pojedynczych elementów – jednostek. Test jednostkowy sprawdza fragment kodu (testuje go) i porównuje wynik z oczekiwanym wynikiem. Testy jednostkowe powinny być łatwe, zautomatyzowane, powtarzalne oraz działać niezależnie od siebie. Testy powinny wyłapywać i lokalizować błędy. Z testowaniem związane jest pojęcie pokrycia kodu (ang. code coverage), które mówi ile procent kodu zostało sprawdzone przez test. Nazwa testu powinna wyjaśniać w jakim celu został napisany test. Podczas testowania należy uwzględnić przypadki graniczne. Do przeprowadzania testów jednostkowych w języku C++ można użyć narzędzia *Google C++ Testing Framework*. Bibliotekę należy pobrać z repozytorium <https://github.com/google/googletest/> oraz w odpowiedni sposób dodać ją do projektu. Dokumentacja biblioteki *Google Test* dostępna jest na stronie internetowej <https://google.github.io/googletest/>. W łatwy sposób można dodać bibliotekę w środowisku *Qt Creator*. Na rysunkach 13.1. oraz 13.2. zaprezentowane zostało w jaki sposób należy to

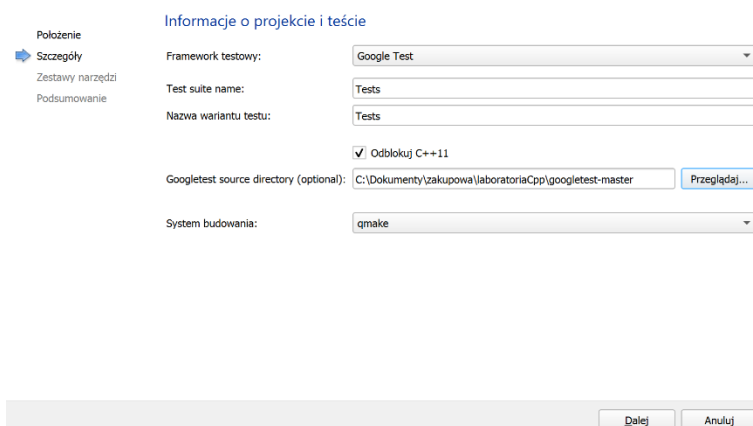


zrobić. Na początku należy wybrać *new Project*, a następnie pokaże się okienko z wyborem projektu. Rysunek 13.1. przedstawia jaki rodzaj projektu trzeba wybrać. Jest to *Projekt automatycznego testu*.



Rys. 13.1. Wybór projektu

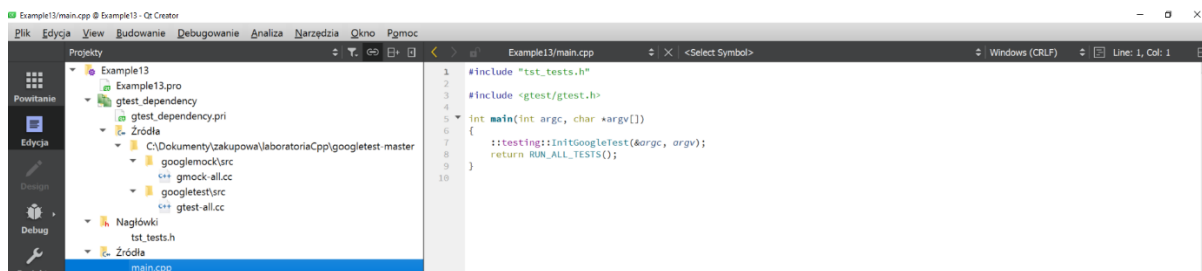
Następnie należy kliknąć dalej i wpisać nazwę projektu. Potem pojawi się okienko *Informacje o projekcie i teście*, które należy uzupełnić tak jak zostało to pokazane na rysunku 13.2. Jako framework testowy należy wybrać *Google Test*. Następnie w pola *Test suite name* oraz *Nazwa wariantu test* należy wpisać nazwy własne. Nazwa wpisana w pole *Nazwa wariantu test* będzie nazwą klasy, która zostanie utworzona (potem można dodawać kolejne klasy lub usunąć stworzoną klasę). Następnie należy wybrać katalog, w którym znajduje się rozpakowany ściągnięty plik biblioteki *Google Test* i jego również dołączyć.



Rys. 13.2. Tworzenie projektu

Po przejściu dalej należy wskazać kompilator. Po utworzeniu projektu powinniśmy zobaczyć to co przedstawia rysunek 13.3. Po lewej stronie znajduje się katalog *gtest_dependency*, który na podstawie wskazanego folderu zawierającego pliki biblioteki *Google Test*, podlinkował je. Nie musieliśmy tego ręcznie ustawiać. W pliku *main.cpp* dołączony jest plik nagłówkowy biblioteki *Google Test* oraz plik *tst_tests.h*, w którym mogą być pisane testy (plik można również skasować i dodać swój). Domyślnie podczas tworzenia

projektu został dodany jeden przypadek testowy, ale jego można usunąć. Testy można pisać zarówno w pliku nagłówkowym jak i źródłowym.



Rys. 13.3. Widok projektu po jego utworzeniu

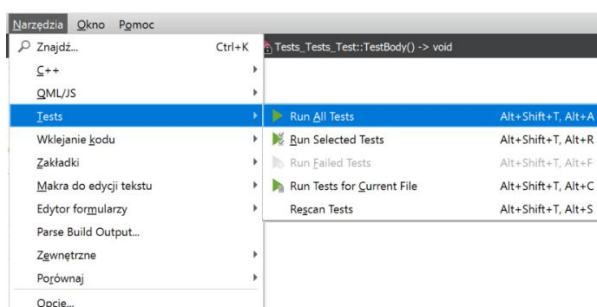
Wykorzystajmy dołączony do projektu przypadek testowy (*tst_tests.h*) aby zaprezentować w jaki sposób uruchamiać testy. Można po prostu nacisnąć na zielony trójkąt z lewej strony, wtedy pojawi się konsola, przedstawiona na rysunku 13.4. Jeden test został wykonany i zakończył się sukcesem.

```
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from Tests
[ RUN      ] Tests.Tests
[       OK ] Tests.Tests (0 ms)
[-----] 1 test from Tests (6 ms total)

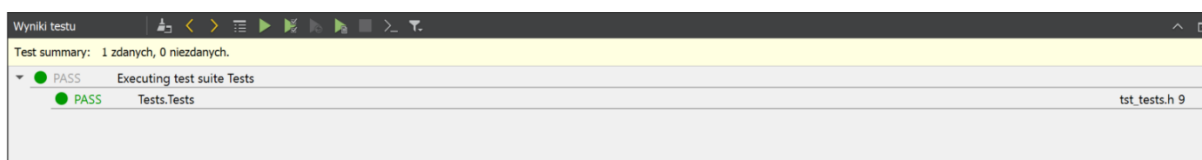
[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (27 ms total)
[ PASSED  ] 1 test.
```

Rys. 13.4. Widok konsoli po uruchomieniu testów

Kolejnym sposobem na uruchomienie testów jest wybór z menu: *Narzędzia*, dalej *Test* i potem *Run All Tests*. Zostało to przedstawione na rysunku 13.5. Wtedy nie pojawi się konsola, ale na dole znajdziemy podsumowanie wykonanych testów z informacją, z jakim statusem się wykonały – rysunek 13.6.

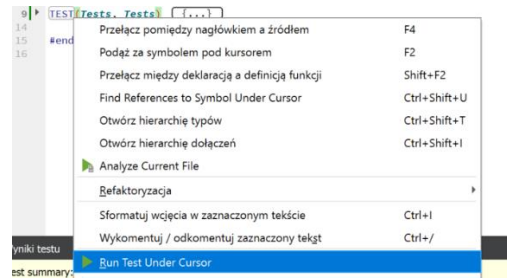


Rys. 13.5. Uruchamianie testów



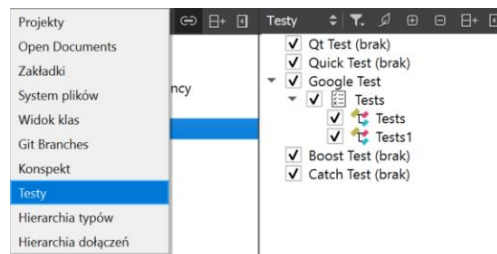
Rys. 13.6. Informacja o statusie testów

Istnieje również możliwość uruchomienia tylko jednego testu. Rysunek 13.6. prezentuje jak to zrobić. Należy kliknąć lewym klawiszem myszy na teście i wybrać *Run Test Under Cursor*.



Rys. 13.7. Uruchamianie wybranego testu

Środowisko *Qt Creator* daje również możliwość wybrania, które testy mają zostać uruchomione. W tym celu należy przełączyć *Projekty* na *Testy* w obszarze po lewej stronie. Wtedy będzie możliwość włączania i wyłączania testów – rysunek 13.8.



Rys. 13.7. Wybieranie testów

Organizacja testów jednostkowych może być różna. Jedno z podejść jest takie, że testy jednej klasy umieszczane są w jednej klasie testowej. Można również zastosować podejście, w którym dana klasa testowa będzie zawierała zbiór podobnych funkcji/metod. Na potrzeby tego laboratorium zostanie przyjęte pierwsze podejście. Z testowaniem powiązane jest pojęcie asercji. Asercja to wyrażenie, które sprawdza czy warunek został spełniony. Wynikiem asercji może być: sukces, błąd krytyczny lub błąd niekrytyczny. W *Google Test* wyróżnia się dwie podstawowe asercje:

- rozpoczynające się od słowa *ASSERT* – powoduje, że wykonywanie bieżącego testu jest przerywane, jeśli warunek nie zostanie spełniony;
- rozpoczynające się od słowa *EXPECT* – powoduje, że wykonywanie bieżącego testu jest kontynuowane jeśli warunek nie zostanie spełniony.

Poniżej zostaną zaprezentowane wybrane asercje rozpoczynające się od słowa *ASSERT* ale takie same występują również z *EXPECT*:

- *ASSERT_EQ(v1,v2)* – sprawdza warunek: $v1 == v2$;
- *ASSERT_NE(v1,v2)* – sprawdza warunek: $v1 != v2$;
- *ASSERT_LT(v1,v2)* – sprawdza warunek: $val1 < val2$;
- *ASSERT_TRUE(v1)* – sprawdza warunek: $v1 == true$;
- *ASSERT_FALSE(v1)* – sprawdza warunek: $v1 == false$;
- *ASSERT_STREQ(v1, v2)* – sprawdza warunek $v1 == v2$ (dla typu *string*);
- *ASSERT_STRNE(v1,v2)* – sprawdza warunek $v1 != v2$ (dla typu *string*);
- *ASSERT_NO_THROW* – sprawdza czy nie jest rzucany wyjątek;
- *ASSERT_THROW(val, des)* – sprawdza czy *val* rzuca wyjątek *desc*;

- *ASSERT_ANY_THROW(val)* – sprawdza czy *val* rzuca jakikolwiek wyjątek.

Biblioteka *Google Test* udostępnia również makro *EXPECT_THAT* (jest również *ASSERT_THAT* ale nie zaleca się jego używać), które można użyć do sprawdzenia wartości w połączeniu z pomocnikami (ang. matcher). Niech pierwszym argumentem będzie zmienna *arg*. Pomocnikami mogą być na przykład:

- *DoubleNear(val, err)* – sprawdzenie czy *arg* jest równe *val* z maksymalnym błędem równym *err*;
- *ContainsRegex(val)* – sprawdzenie czy *arg* zawiera wyrażenie regularne *val*;
- *HabSubstr(val)* – sprawdza czy *arg* zawiera *val*;
- *StrEq(val)* – sprawdza czy *arg* jest równe *val*;
- *Eq(val)* – sprawdza czy *arg* jest równy *val*;
- *Gt(val)* – sprawdza, czy *arg* jest większy niż *val*;
- *Lt(val)* – sprawdza, czy *arg* jest mniejszy niż *val*.

Więcej informacji na temat pomocników można przeczytać na stronie internetowej <https://google.github.io/googletest/reference/assertions.html>

W celu utworzenia testu należy użyć makra *TEST*. W zawiasach należy podać nazwę testowanego przypadku i nazwę testu. Na listingach 13.1. – 13.3. zaprezentowane zostały przykłady testów jednostkowych. Listing 13.1. oraz listing 13.2. zawierają plik nagłówkowy oraz źródłowy, w których są deklaracje oraz definicje funkcji. Zostały stworzone cztery funkcje:

- *isEven* – sprawdzająca czy liczba jest parzysta;
- *sign* – funkcja znaku;
- *difference* – obliczająca różnice między dwoma sąsiednimi elementami;
- *show* – wyświetlająca komunikat na konsolę.

```
1  #ifndef FUNCTIONS_H
2  #define FUNCTIONS_H
3  #include<vector>
4  #include<string>
5  #include<iostream>
6  using namespace std;
7  bool isEven(int n);
8  int sign(int n);
9  vector<int> difference(vector<int> vec);
10 void show(string line);
11 #endif // FUNCTIONS_H
```

Listing 13.1. Plik nagłówkowy *functions.h*

```
1  #include "functions.h"
2  bool isEven(int n){
3      if(n%2==0)
4          return true;
5      else
6          return false;
7  }
8  int sign(int n){
```



```

9      if(n<0)
10         return -1;
11      if(n>0)
12         return 1;
13      else
14         return 0;
15 }

16 vector<int> difference(vector<int> vec) {
17     vector<int>res;
18     for(int i=1; i<vec.size();i++){
19         res.push_back(vec[i]-vec[i-1]);
20     }
21     return res;
22 }

23 void show(string line){
24     if(line.empty())
25         throw runtime_error("Pusty napis");
26     cout<<"Podany napis to: "<<line<<endl;
27 }

```

Listing 13.2. Plik źródłowy *functions.cpp*

W celu przetestowania funkcji stworzony został plik *functionsTest.cpp*, w którym zaimplementowane zostały testy jednostkowe. Listing 13.3. zawiera kod, który znajduje się w pliku *functionsTest.cpp*. Listing 13.3. zawiera:

- linijki 1 – 2 – dołączenie plików nagłówkowych;
- linijki 5 – 8 – implementacja testu jednostkowego. W nawiasach po słowie *TEST* zostały podane dwie nazwy. Pierwsza to nazwa testowanego przypadku a druga to nazwa testu. Testów dla danego przypadku może być kilka stąd nazwa przypadku nie musi być unikatowa, ale nazwa testu musi być unikatowa. W linijce 7 pojawiła się asercja porównująca wartość z wartością oczekiwaną. W tym przypadku oczekujemy wartość *false* ponieważ 5 jest liczbą nieparzystą;
- linijki 9 – 12 – kolejny test jednostkowy dla funkcji *isEven*. Nazwa przypadku testowego jest taka sama ale inna jest nazwa testu. Została użyta asercja, która sprawdza, czy wartość i wartość oczekiwana są różne;
- linijki 13 – 16 – test jednostkowy dla funkcji *sign*. Użyta została asercja, która umożliwia porównanie wartości i wartości oczekiwanej z użyciem pomocnika (matcher);
- linijki 17 – 28 – test jednostkowy dla funkcji *difference*. Stworzone zostały dwa kontenery: *res*, który zawiera wynik funkcji *difference* oraz *resExpected*, który zawiera oczekiwane wartości. W jednym teście zastosowane zostały dwie asercje. Pierwsza z asercji (linijka 23) porównuje rozmiary kontenerów. Jeśli rozmiary nie będą się zgadzały test zakończy się. Jeśli rozmiary będą zgodne następnie w pętli będą porównywane wartości z obydwu kontenerów. W linijkach 25 – 26 występuje asercja, która jeśli pojawi się błąd będzie kontynuowała sprawdzanie dalej. Dodatkowo do każdego rodzaju asercji można dodać operator <<, który spowoduje wyświetlenie się komunikatu;

- linijki 29 – 31 – test jednostkowy dla funkcji *show*. Umieszczona w teście asercja dotyczy rzucania wyjątku. W tym przypadku zostanie rzucony wyjątek *runtime*, ponieważ do funkcji *show* został przekazany pusty napis.

```
1  #include <gtest/gtest.h>
2  #include <gmock/gmock-matchers.h>
3  #include "functions.h"
4  using namespace testing;

5  TEST(IsEvenTest, OddTests)
6  {
7      ASSERT_EQ(false, isEven(5));
8  }

9  TEST(IsEvenTest, EvenTests)
10 {
11     ASSERT_NE(false, isEven(6));
12 }

13 TEST(SignTest, FirstIfTests)
14 {
15     ASSERT_THAT(1, Eq(sign(6)));
16 }

17 TEST(DifferenceTest, OkResultTests)
18 {
19     vector<int> res;
20     vector<int> num{1,2,3,6,5};
21     res=difference(num);
22     vector<int> resExpeted{1,1,3,-1};
23     ASSERT_EQ(res.size(), resExpeted.size());
24     for(int i=0; i<res.size(); i++){
25         EXPECT_EQ(res[i], resExpeted[i])
26             <<" Obrot " <<i<<endl;
27     }
28 }

29 TEST(ShowTest, EmptyLineTests) {
30     EXPECT_THROW(show(""), runtime_error);
31 }
```

Listing 13.3. Plik źródłowy *functionsTest.cpp*

Na listingu 13.4. przedstawiony został plik *main.cpp*. W celu przeprowadzania testów niezbędne są linijki 4 i 5.

```
1  #include <gtest/gtest.h>
2  int main(int argc, char *argv[])
```

```

3  {
4      ::testing::InitGoogleTest(&argc, argv);
5      return RUN_ALL_TESTS();
6  }

```

Listing 13.4. Plik main.cpp

Po uruchomieniu testów dostajemy informację, że wszystkie testy przeszły pomyślnie. Dzięki stosowaniu opisów dla testowanych przypadków w podsumowaniu mamy podział na poszczególne sekcje. Rysunek 13.8. przedstawia uzyskane wyniki testów.

Test summary: 5 zdanych, 0 niezdanych.		
▼	PASS	Executing test suite IsEvenTest
	PASS	IsEvenTest.OddTests
	PASS	IsEvenTest.EvenTests
▼	PASS	Executing test suite SignTest
	PASS	SignTest.FirstIfTests
▼	PASS	Executing test suite DifferenceTest
	PASS	DifferenceTest.OkResultTests
▼	PASS	Executing test suite ShowTest
	PASS	ShowTest.EmptyLineTests

Rys. 13.8. Wyniki testów z listingu 13.3

Teraz dodajmy dwa testy, które zakończą się niepowodzeniem. Listing 13.5. zawiera takie testy. Są to niektóre testy z listingu 13.3, pogrubione zostały elementy, które zostały specjalnie zmienione. Do nazw testów zostało dodane słowo *Err* w celach dydaktycznych aby było łatwiej rozróżnić.

```

1  TEST(IsEvenTest, OddTestsErr)
2  {
3      ASSERT_EQ(true, isEven(5));
4  }

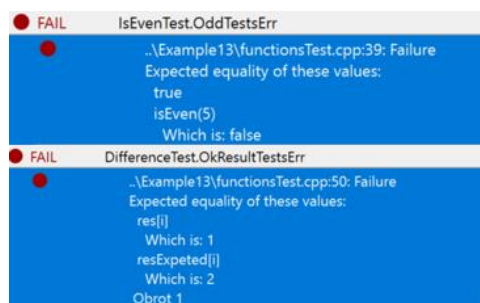
5  TEST(DifferenceTest, OkResultTestsErr)
6  {
7      vector<int> res;
8      vector<int> num{1,2,3,6,5};
9      res=difference(num);
10     vector<int> resExpeted{1,2,3,-1};
11     ASSERT_EQ(res.size(),resExpeted.size());
12     for(int i=0;i<res.size();i++){
13         EXPECT_EQ(res[i],resExpeted[i])
14             <<" Obrot " <<i<<endl;
15     }
16 }

```

Listing 13.5. Testy, które zakończą się niepowodzeniem

Na rysunku 13.9. przedstawiono wynik testów z listingu 13.4. Jeśli test nie zakończy się sukcesem to wyświetlana jest informacja co było wartością oczekiwaną a co się pojawiło. W

przypadku funkcji *difference* tylko dla drugiej iteracji wartości nie były sobie równe, a skoro użyta została asercja *EXPECT* to program kontynuował wykonywanie testów.



Rys. 13.9. Wyniki testów z listingu 13.5

Biblioteka *Google Test* umożliwia również przeprowadzanie parametryzowanych testów oraz umożliwia pisanie testów dla klas generycznych. W taki sam sposób w jaki przedstawione zostało testowanie funkcji można przetestować metody w klasie. Listingi 13.6. – 13.8. przedstawiają przykład przeprowadzenia parametryzowanych testów. Listing 13.6. przedstawia plik nagłówkowy klasy *Number*, a listing 13.7. przedstawia plik źródłowy klasy *Number*. Na tej klasie zostaną przeprowadzone testy jednostkowe. Klasa *Number* zawiera jedno pole *n*, które przechowuje liczbę całkowitą, konstruktor, metodę zwracającą inkrementację liczby *n*, setter oraz getter.

```

1  #ifndef NUMBER_H
2  #define NUMBER_H
3  class Number
4  {
5  private:
6      int n;
7  public:
8      Number(int n1);
9      int inc();
10     int getN();
11     void setN(int n1);
12 };
13 #endif // NUMBER_H

```

Listing 13.6. Plik nagłówkowy klasy *Number*

```

1  #ifndef NUMBER_H
2  #include "number.h"
3  Number::Number(int n1)
4  {
5      n=n1;
6  }
7  int Number::inc() {
8      return n+1;
9  }
10 int Number::getN() {
11     return n;

```



```

12 }

13 void Number::setN(int n1) {
14     n=n1;
15 }

```

Listing 13.7. Plik źródłowy klasy *Number*

Na listingu 13.8. zaprezentowany został plik nagłówkowy *numberTests.cpp*, który zawiera testy dla klasy *Number*. Jeśli chcemy przetestować metody musimy stworzyć obiekt. Biblioteka *Google Test* umożliwia wyodrębnienie fragmentu kodu, który może być używany wielokrotnie. Można tam na przykład umieścić obiekt, który posłuży do wielu testów zamiast tworzyć go od nowa w każdym makro *TEST*. Dodatkowo listing prezentuje w jaki sposób przeprowadzić testy parametryzowane. Listing 13.8. zawiera:

- linijka 4 – dołączenie przestrzeni nazw;
- linijki 5 – 13 – stworzenie struktury (można również stworzyć klasę), w której umieszczony został obiekt klasy, której metody będą testowane. Struktura dziedziczy po *Test*;
- linijki 14 – 16 – stworzenie testu jednostkowego. Zostało użyte makro *TEST_F*, ponieważ tworzone są testy operujące na tym samym zestawie danych (*NumberTest*);
- linijki 20 -23 - stworzenie struktury (można stworzyć również klasę), która przechowuje dwa pola: liczbę i oczekiwany wynik. Struktura ta będzie potrzebna do wykonywania testów sparametryzowanych;
- linijki 24 – 31 – stworzenie struktury (można stworzyć również klasę) dziedziczącej po *NumberTest* oraz *WithParamInterface*. W konstruktorze tej struktury wywołana została metoda *setN*, która ustawia pole w klasie na wartość *nInit*. Jeśli testowana byłaby funkcja wtedy nie trzeba pisać konstruktora. Funkcja *GetParam* umożliwia zwrócenie obiektu agregującego parametry testów (instancji *NumberParamTest*);
- linijki 32 – 36 – test jednostkowy parametryzowany. Nazwa przypadku testowego jest taka sama jak nazwa struktury czyli *NumberParmTest*. W obiekcie *n* w polu *n* została ustawiona wartość z linijki 29;
- linijki 37 – 39 – wywołanie testów parametryzowanych. Jako trzeci argument makro przyjmuje listę przypadków).

```

1  #include <gtest/gtest.h>
2  #include <gmock/gmock-matchers.h>
3  #include "number.h"
4  using namespace testing;
5  struct NumberTest:Test{
6      Number *n;
7      NumberTest() {
8          n=new Number(4);
9      }
10     ~NumberTest() {
11         delete n;
12     }
13 };

14 TEST_F(NumberTest,NumberInc) {

```



```

15     EXPECT_EQ(5,n->inc());
16 }

17 TEST_F(NumberTest,NumberGet){
18     EXPECT_EQ(4,n->getN());
19 }

20 struct NumberState{
21     int nInit;
22     int inc;
23 };

24 struct NumberParamTest: NumberTest,
25                         WithParamInterface<NumberState>
26 {
27     NumberParamTest()
28     {
29         n->setN(GetParam().nInit);
30     }
31 };







32 TEST_P(NumberParamTest, NTests){
33     NumberState as=GetParam();
34     int inc=n->inc();
35     EXPECT_EQ(as.inc,inc);
36 }

37 INSTANTIATE_TEST_CASE_P(Default, NumberParamTest,
38                         Values(NumberState{4,5},
39                         NumberState{3,4}));

```

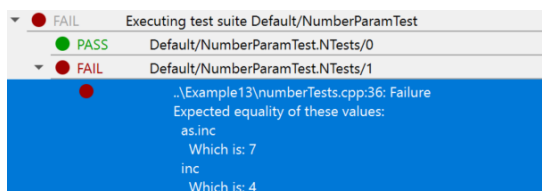
Listing 13.8. Plik źródłowy numberTests.cpp

Na rysunku 13.10. przedstawiony został wynik testów z listingu 13.8.

Test summary: 4 zdanych, 0 niezdanych.		
▼	 PASS	Executing test suite NumberTest
	 PASS	NumberTest.NumberInc
	 PASS	NumberTest.NumberGet
▼	 PASS	Executing test suite Default/NumberParamTest
	 PASS	Default/NumberParamTest.NTests/0
	 PASS	Default/NumberParamTest.NTests/1

Rys. 13.10. Wyniki testów z listingu 13.8

Teraz zmodyfikujemy parametry testu, powodując to, że test zakończy się błędem. Test został wywołany z wartościami {3,4} a wywołajmy go z wartościami {3,7}. Rysunek 13.11. przedstawia wynik testu. Pierwszy test zakończył się sukcesem a drugi błędem.



Rys. 13.11. Wyniki testów po modyfikacji parametrów

Zadania do wykonania:

Zadanie 13.1. Przykładowy kod

Uruchom kody zaprezentowane podczas laboratorium.

Zadanie 13.2. Liczba pierwsza

Napisz funkcję, która przyjmuje całkowitą liczbę. Funkcja powinna zwrócić wartość *true* jeśli liczba jest pierwsza, w przeciwnym wypadku *false*. Następnie przetestuj funkcję.

Zadanie 13.3. Liczba odwrotna

Napisz funkcję, która przyjmuje liczbę zmiennoprzecinkową i zwraca liczbę odwrotną do przekazanej liczby. Funkcja powinna rzucić wyjątek jeśli przekazana liczba to 0. Następnie należy przetestować funkcję.

Zadanie 13.4. Konto bankowe

Zdefiniuj klasę *BankAccount* opisującą rachunek bankowy. Polami klasy powinny być: *last_name*, *account_number* (ciąg znaków), *balance*. Klasa powinna posiadać metody:

- konstruktor umożliwiający inicjalizację atrybutów klasy;
- *add* - metodę dodającą do rachunku przekazaną kwotę. Dodawana wartość ma być liczbą dodatnią;
- *withdraw* - metoda wypłacająca z (odejmująca od) rachunku przekazaną kwotę. Kwota ma być liczbą dodatnią. Po wypłacie stan konta ma być liczbą większą bądź równą 0;
- *isMillionaire* – metoda zwracająca *true* jeśli stan konta jest większy bądź równy milion, w przeciwnym wypadku metoda zwraca *false*;
- odpowiednie setery i gettery do pól;

Następnie przetestuj stworzoną klasę. Spróbuj użyć testów parametrycznych.

LABORATORIUM 14. KOŁOKWIUM 2.

Cel laboratorium:

Weryfikacja nabytych umiejętności dotyczących obsługi wejścia oraz wyjścia, klasy string i wyrażeń regularnych, zarządzania pamięcią, wyrażeń lambda, krotek oraz testowania.

Wytyczne do kolokwium 2:

- zakres laboratoriów 9-13,
- próg zaliczeniowy 51%,
- pozostałe wytyczne i sposób zaliczenia kolokwium ustala prowadzący zajęcia.

LABORATORIUM 15. ZALICZENIE KOŃCOWE.

Cel laboratorium:

Podsumowanie zajęć i wystawienie ocen końcowych. Przeprowadzenie ewentualnych poprawek.

Wytyczne do oceny końcowej:

- pozytywne oceny cząstkowe.
- ewentualne dodatkowe wymagania prowadzącego zajęcia.



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny





Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego