



**POLITECHNIKA LUBELSKA  
WYDZIAŁ ELEKTROTECHNIKI I  
INFORMATYKI**

**KIERUNEK STUDIÓW  
INFORMATYKA**

***MATERIAŁY DO ZAJĘĆ  
LABORATORYJNYCH***

**Programowanie obiektowe w C++**

**Autor:**  
dr inż. Tomasz Nowicki

Lublin 2020

## **INFORMACJA O PRZEDMIOCIE**

### **Cele przedmiotu:**

- Zapoznanie studentów z podstawami programowania obiektowego z wykorzystaniem języka C++.
- Nabycie umiejętności przez studentów programowania zorientowanego obiektowo oraz wykorzystywania obiektowych bibliotek.

### **Efekty kształcenia w zakresie umiejętności:**

- Potrafi posługiwać się dokumentacją opisującą bibliotekę języka C++, wyszukiwać niezbędne informacje w literaturze, także w języku angielskim.
- Potrafi opisać w sposób niesformalizowany wymagania wobec aplikacji o charakterze obiektowym.
- Potrafi zaprojektować aplikację obiektową o średnim i dużym stopniu złożoności, umie opracować prostą aplikację graficzną.

### **Literatura do zajęć:**

- Literatura podstawowa
  1. Josuttis N.M., C++. Biblioteka standardowa. Podręcznik programisty. Wydanie II, Helion, Gliwice 2014.
  2. Grębosz J., Opus magnum C++11. Programowanie w języku C++, Helion, 2018.
  3. Meyer B., Programowanie zorientowane obiektowo, Helion 2005.
- Literatura uzupełniająca
  1. Stroustrup B., Programowanie. Teoria i praktyka z wykorzystaniem C++, Helion, Gliwice 2010.
  2. Grębosz J., Symfonia C++ Standard. Wydawnictwo Edition 2000, Kraków 2008.
  3. Czerwiński D., Digital Filter Implementation in Hadoop Data Mining System, Computer Networks, Communications in Computer and Information Science, Springer 2015.

### **Metody i kryteria oceny:**

- Oceny częściowe:
  - o Przygotowanie merytoryczne do zajęć laboratoryjnych na podstawie: wykładów, literatury, pytań kontrolnych do zajęć.
  - o Frekwencja i kreatywność na zajęciach: próg zaliczeniowy 80%.
  - o Dwa kolokwia: próg zaliczeniowy 60%.
- Ocena końcowa - zaliczenie przedmiotu:
  - o Pozytywne oceny częściowe.
  - o Ewentualne dodatkowe wymagania prowadzącego zajęcia.



### Plan zajęć laboratoryjnych:

<a href="#">Lab 01</a>	Wprowadzenie do zajęć, edytor tekstowy, kompilator.
<a href="#">Lab 02</a>	Zmienne i wyrażenia, instrukcje warunkowe i iteracyjne. Referencje i wskaźniki (Cz.1).
<a href="#">Lab 03</a>	Klasy i obiekty, konstruktory i destruktory (Cz.1).
<a href="#">Lab 04</a>	Klasy i obiekty, konstruktory i destruktory (Cz.2).
<a href="#">Lab 05</a>	Klasy i obiekty, konstruktory i destruktory (Cz.3). Referencje i wskaźniki (Cz.2).
<a href="#">Lab 06</a>	Polimorfizm, funkcje wirtualne, klasy abstrakcyjne.
<a href="#">Lab 07</a>	Dziedziczenie i dostęp do elementów klas.
<a href="#">Lab 08</a>	Kolokwium 1.
<a href="#">Lab 09</a>	Referencje i wskaźniki (Cz.3). Dynamiczna alokacja pamięci.
<a href="#">Lab 10</a>	Biblioteka STL i algorytmy, przeciążanie operatorów.
<a href="#">Lab 11</a>	Praktyka programowania obiektowego. Uruchomienie i testowanie aplikacji <i>Wirtualny ekosystem</i> .
<a href="#">Lab 12</a>	Klasa napisów string, zapis i odczyt plików, formatowanie.
<a href="#">Lab 13</a>	Biblioteka Qt, wykorzystanie klas biblioteki do budowy aplikacji "Wykresy funkcji matematycznych"
<a href="#">Lab 14</a>	Repetitorium.
<a href="#">Lab 15</a>	Kolokwium 2.

### Informacje i założenia wstępne:

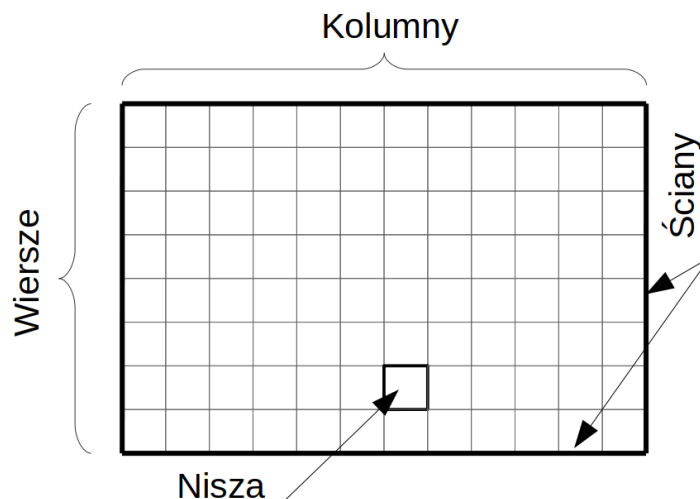
- Przedmiot laboratoryjny *Programowanie obiektowe w C++* jest poprzedzony w toku studiów przedmiotem *Programowanie strukturalne*, które jest prowadzone w języku C. Przyjęte zostało założenie, że każdy student dobrze opanował podstawy programowania w języku C oraz korzystał już z jakiegoś zintegrowanego środowiska programistycznego. W związku z tym, że język C++ rozszerza język C, nie ma już potrzeby wyjaśniania studentom spraw podstawowych.
- Zajęcia laboratoryjne będą prowadzone przy użyciu zintegrowanego środowiska programistycznego *Qt Creator* w wersji bezpłatnej.
- Zajęcia zostały przygotowane, a kody uruchomione i przetestowane w środowisku *Linux Mint 19.1 Tessa*, przy użyciu *Qt Creator 4.11.1* (Bazujący na Qt 5.14.1 (GCC 5.3.1 20160406 (Red Hat 5.3.1-6), 64 bitowy))
- Przedstawione kody programów są częścią ZPRPL i są przewidziane wyłącznie do celów dydaktycznych.
- W trakcie ćwiczeń laboratoryjnych zadaniem studentów jest odpowiednie przeniesienia kodów źródłowych zamieszczonych w tym opracowaniu do własnych indywidualnych projektów, analiza tych kodów, uruchomienie ich oraz co najważniejsze udzielenie odpowiedzi na załączone pytania.
- Studenci archiwizują swoje projekty, ponieważ plik z poprzednich zajęć wykorzystywane są w kolejnych. Zajęcia od 1 do 12 należy zrealizować jako pojedynczy projekt Qt Creator-a.
- Przed kolokwiami studenci potwierdzają swoje przygotowanie do nich wysyłając prowadzącemu kopię swoich projektów, w których jest kompletnie zrealizowany materiał z laboratoriów poprzedzających kolokwium.
- Zadania oznaczone gwiazdką \* nie są obowiązkowe do realizacji.



## Aplikacja Wirtualny ekosystem

Jednym z zamierzeń dydaktycznych jest opanowanie programowania obiektowego w języku C++ na poziomie umożliwiającym wykonanie aplikacji obiektowej o znacznym stopniu złożoności. W tym celu w trakcie zajęć laboratoryjnych stopniowo realizowany jest projekt *Wirtualny ekosystem*.

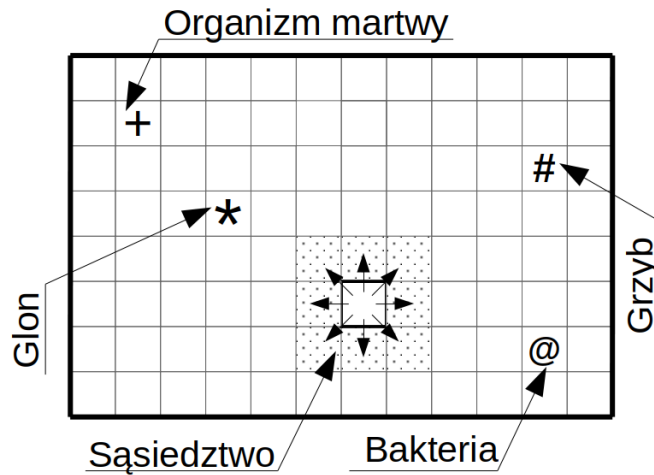
Wirtualny ekosystem to prosty biocybernetyczny symulator rozwoju ekosystemu, który tworzą trzy gatunki wirtualnych organizmów: glony – oznaczone jako \*, grzyby # i bakterie @. Środowisko ekosystemu to prostokątna tablica składająca się z komórek, które nazywane są dalej niszami (Rys. 1). Każda nisza może być albo pusta, albo zajęta przez żywy organizm, albo zajęta przez organizm martwy (Rys. 2). Każda nisza ma swoje bezpośrednie sąsiedztwo (Rys. 2). Aktywność żywych organizmów dotyczy tylko tego sąsiedztwa. Środowisko takie wstępnie jest zasiedlane w sposób dowolny organizmami żywymi. Rozwój ekosystemu następuje w kolejnych krokach symulacji. Krok to przejście od bieżącego stanu symulacji do stanu kolejnego. Stan ekosystemu to rozmieszczenie organizmów w niszach. W trakcie tego przejścia (kroku) organizmy wykonuje swoje funkcje życiowe.



Rys.1. Wirtualne środowisko składające się z 96 pustych nisz ułożonych w 8 wierszy i z 12 kolumn.

Każdy organizm starzeje się i jeżeli przekroczy swój maksymalny wiek (określony w liczbie kroków symulacji) staje się organizmem martwym. Maksymalny wiek organizmu ustalany jest losowo dla każdego organizmu w zadanym przedziale. Każdy gatunek ma swój przedział.

Jeżeli organizm nie jest martwy, to stara się najeść. Grzyb szuka w swoim sąsiedztwie organizmów martwych i jeżeli takie są to losowo wybiera jeden z nich i go wchłania. Organizm wchłonięty znika z ekosystemu. Bakteria natomiast poluje. W pierwszej kolejności szuka w swoim sąsiedztwie glonów i jeżeli takie są wybiera losowo jednego. Jeżeli nie ma glonów to szuka innej bakterii. Organizm upolowany jest wchłonięty i znika z ekosystemu. Glon zaś po prostu pozyskuje pożywienie (prowadzi fotosyntezę). W pojedynczym kroku symulacji dla każdego organizmu możliwy jest 1 posiłek. Każdy organizm ma swój limit posiłków, po przekroczeniu którego już nie jest głodny i już nie szuka lub nie produkuje pożywienia. Limit posiłków jest charakterystyczny dla każdego gatunku.

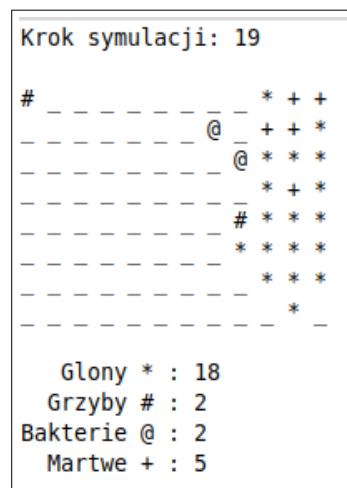


Rys.2. Ilustracja sąsiedztwa wybranej pustej niszy, trzech nisz zajętych przez żywe organizmy (glon, bakteria i grzyb) oraz 1 niszy zajętej przez organizm martwy (już bez znaczenia jaki). Jest to przykładowa konfiguracja.

Jeżeli organizm nie jest głodny to stara się rozmnożyć. Organizm szuka w swoim sąsiedztwie wolnej celi i jeżeli takie są to losowo w jednej z nich umieszcza swojego potomka. Rozmnożenie się zmniejsza licznik posiłków o koszt potomka. Koszt potomka jest charakterystyczny dla każdego gatunku. Potomek jest młody, tzn. jego licznik życia jest ustawiony na wartość początkową i głodny, tzn. licznik posiłków jest wyzerowany.

Jeżeli organizm jest grzybem lub bakterią i nie posila się, ani się nie rozmnaża, to zmienia celę, jeżeli w jego sąsiedztwie jest wolna celda. Zmiana następuje w sposób losowy.

Zrealizowana aplikacja posiada interfejs konsolowy i wyświetla animację rozwoju ekosystemu (Rys. 3).



Rys. 3. Wygląd aplikacji Wirtualny Ekosystem dla przykładowej konfiguracji.

## LABORATORIUM 1. WPROWADZENIE DO ZAJĘĆ, EDYTOR TEKSTOWY, KOMPILATOR.

### Cel laboratorium:

Celem laboratorium jest zaznajomienie się ze środowiskiem programistycznym *Qt Creator* i jego obsługą w zakresie tworzenia nowego projektu programistycznego, budowania i uruchamiania aplikacji.

### Zakres tematyczny zajęć:

- Omówienie sposobu realizacji zajęć laboratoryjnych.
- Przedstawienie najważniejszych elementów środowiska *Qt Creator*.
- Tworzenie nowego projektu programistycznego.
- Edycja kodu źródłowego.
- Budowanie i uruchamianie aplikacji.

### Kompedium wiedzy:

- Środowisko *Qt Creator* jest dostępne pod adresem: <https://www.qt.io>.
- Oprogramowanie jest dostępne na licencji LGPL.
- *Qt Creator* jest dostępny dla systemów operacyjnych: Linux, Windows, OS X.
- Program utworzony w języku C++ w środowisku *Qt Creator* jest **kompilowany** i **linkowany** (czyli **budowany**) dla wskazanego systemu operacyjnego. Następnie może być uruchomiony bezpośrednio w tym systemie.
- Najprostszy program w języku C++ jest przedstawiony na listing 1.1.
- Środowiska programistyczne pozwalają na zbudowanie programu w wersji **Debug** oraz **Release**. Wersji Debug używa się w trakcie prac programistycznych, natomiast wersja Release przeznaczona jest do wdrożenia.

### Pytania kontrolne:

1. Objaśnij co to jest zintegrowane środowisko programistyczne.
2. Przedstaw związek pomiędzy językiem C i C++.
3. Omów strukturę najprostszego programu w języku C++.

### Zadanie 2.1. Tworzenie desktopowej aplikacji konsolowej w środowisku Qt Creator

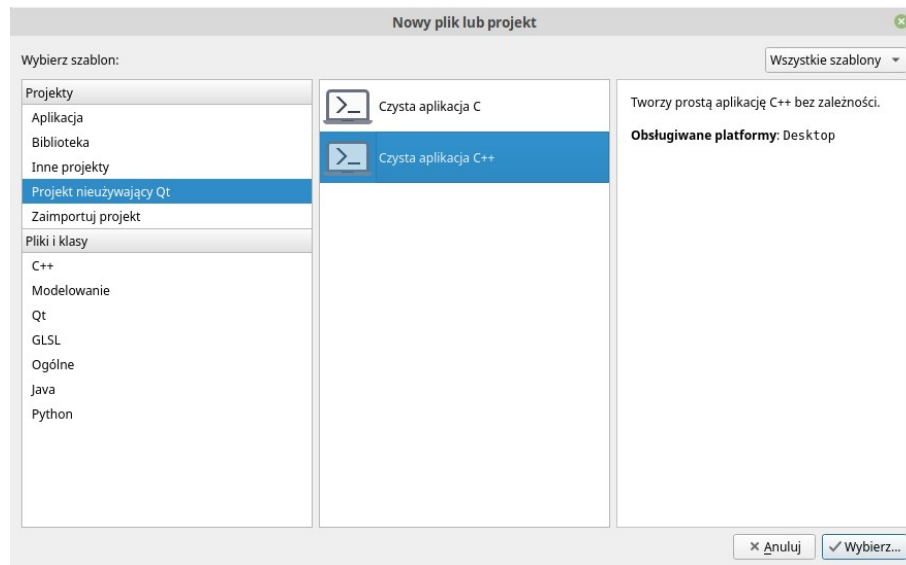
Wykonaj polecenia:



- Uruchom środowisko *Qt Creator*:
- Uruchom okno wyboru nowego projektu: Menu główne → Plik → Nowy plik lub projekt → Pojawia się okno Nowy plik lub projekt (Rys. 1.1).
- Ustaw szablon: Projekty → Projekt nieużywający Qt → Czysta aplikacja C++.
- Zatwierdź szablon i przejdź do kreatora: Przycisk → Wybierz → Pojawia się okno kreatora Czystej aplikacji C++ na etapie Położenie projektu.



- Ustal nazwę i położenie projektu: Nazwa → Wpisz nazwę projektu → **LabCpp**;  
Utwórz w → Wskaż katalog; następnie: Przycisk → Dalej → Przejście do etapu  
Zdefiniuj system budowania .



*Rys 1.1. Pierwsze okno kreatora nowego projektu w środowisku Qt Creator 4.11.1*

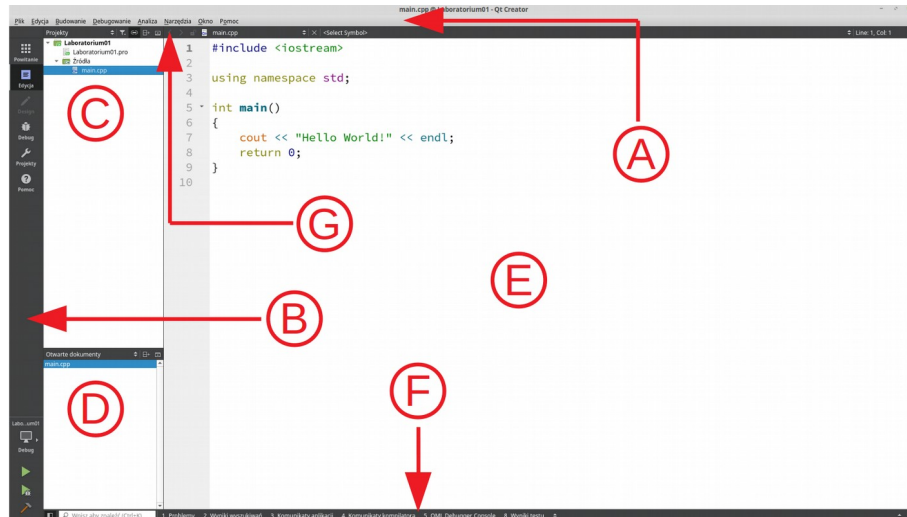
- Ustal system budowania: Pozostaw wartość domyślną; następnie Przycisk → Dalej → Przejście do etapu → Wybór zestawu narzędzi .
- Wybierz zestaw narzędzi: Należy się upewnić, czy jest wskazana opcja *Desktop* (i tylko ta); następne: Przycisk → Dalej → Przejście do etapu Organizacja projektu .
- Zakończ tworzenie nowego projektu: Przycisk → Zakończ → Projekt został utworzony .
- Odnaleźć w systemie plików katalog utworzonego właśnie projektu i sprawdzić czy zawiera pliki: `Laboratorium01.pro` `Laboratorium01.pro.user` `main.cpp` .

## **Zadanie 2.2. Zapoznanie się z elementami środowiska programistycznego Qt Creator**

Odszukaj w uruchomionym środowisku wskazane elementy (A), (B), ..., (F) (Rys. 1.2).

- (A) Menu główne programu. Instrukcje do zadań w tym opracowaniu będą odwoływać się do pozycji tego menu w następujący sposób: Menu główne → ...
- (B) Panel skrótów do najczęściej wybieranych funkcji. Umożliwia m.in. przełączanie trybów pracy edycja, debugowanie, budowanie i uruchamianie aplikacji.
- (C) (D) Bliźniacze okna umożliwiające spojrzenie na bieżący projekt z wybranej perspektywy, a nawet z 2 różnych jednocześnie. Wybór dokonuje się poprzez wskazanie na liście wybieralnej. (Szukaj symbolu rozwijania listy.) Możliwe perspektywy to: Projekty, Otwarte dokumenty, Zakładki, ...
- (E) Edytor tekstu. Tutaj pracujemy z kodem.
- (F) Okna komunikatów. Jeżeli nie ma komunikatów, to okna są zminimalizowane. Kliknięcie na nazwę powoduje rozwinięcie okna, a ponowne kliknięcie zamknięcie okna.
- (G) Pasek skrótów umożliwiający nawigowanie między otwartymi plikami i zawartością tych plików.





Rys 1.2. Standardowy wygląd okna głównego programu Qt Creator 4.11.1

### Zadanie 2.3. Dostosowanie edytora: zmiana rozmiaru czcionki

Wykonaj polecenia:

- Menu główne → Narzędzia → Opcje → Pojawia się okno Opcje .
- Z lewej strony odszukaj pozycję Edytor tekstu → Kliknij .
- Z prawej strony na zakładce Czcionka i kolory → Ustaw rozmiar czcionki np. na 12 .
- Przycisk → OK .

### Zadanie 2.4. Zbudowanie wersji Debug i uruchomienie aplikacji

Wykonaj polecenia:

- Jeżeli w edytorze nie jest widoczny plik `main.cpp`, to go uaktywnij: Okno perspektywy (C) → Wybierz: Projekty → Rozwiń gałęzie → 2× Kliknij `main.cpp` .
- Zmień łańcuch tekstowy `Hello World!` na `Laboratorium 01` .
- Upewnij się czy kod programu wygląda jak na listingu 1.1 .

Listing 1.1 Program wyświetlający napis „Laboratorium 01”

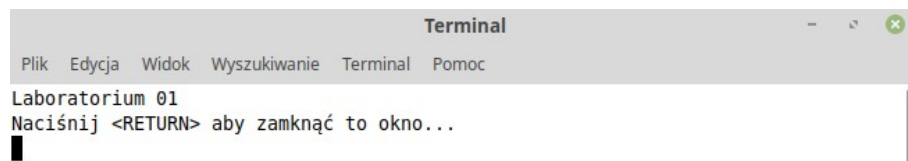
```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     cout << "Laboratorium 01" << endl;
8     return 0;
9 }
```

- Zbuduj program: Menu główne → Budowanie → Zbuduj wszystko
- Sprawdź czy pojawiły się pliki będące wynikiem procesu budowania: Okno perspektywy (D) → Wybierz perspektywę System plików → Przejdź do katalogu wyżej w hierarchii, aż pojawi się katalog, którego nazwa rozpoczyna się od słowa `build...`





- Sprawdź czy w katalogu `build-...` znajdują się pliki: `Makefile` `main.o` `Laboratorium01` (lub `Laboratorium01.exe`)
- Sprawdź ustawienia procesu budowania: 2× Kliknij `Makefile` → W edytorze pojawia się automatycznie wygenerowany plik. Możliwa jest edycja pliku, jednak na tym etapie kursu proszę tego nie robić.
- Sprawdź co zawiera plik `main.o`: 2× Kliknij `main.o` → W edytorze pojawia się obraz pliku. Jest to plik binarny przedstawiony w postaci szesnastkowej. Edycja tego pliku również jest możliwa, wymaga jednak bardzo zaawansowanej wiedzy. Plik `main.o` jest wynikiem kompilacji pliku `main.cpp`.
- Sprawdź co zawiera plik `Laboratorium01`. Jest to również plik binarny. Jest to program uruchamiany przez system operacyjny. Proszę teraz tego pliku nie hakować, tzn. nie edytować.
- Wróć do widoku pliku `main.cpp`: Perspektywa Projekty → 2× Kliknij `main.cpp`.
- Uruchom program: Menu główne → Budowanie → Uruchom → Pojawia się konsola z uruchomionym programem (Rys. 1.3.).



*Rys. 1.3. Wynik działania programu z listing 1.1*

- Zakończ działanie programu: Naciśnąć Enter (lub RETURN).
- Jeżeli program się nie uruchamia (bo np. programista pozmieniał sobie pliki binarne) należy program przebudować przed uruchomieniem: Menu główne → Przebuduj wszystko.
- Uruchom program bez użycia środowiska programistycznego: Odszukaj katalog `build-...` dowolnym managerem plików → Wejdź do tego katalogu → Uruchom konsolę w tym katalogu → np. klikając prawy przycisk myszy i wybierając Otwórz w terminalu → Wpisać komendę `./Laboratorium01` → Program uruchamia się

## **Zadanie 2.5. Zbudowanie wersji Release i uruchomienie aplikacji**

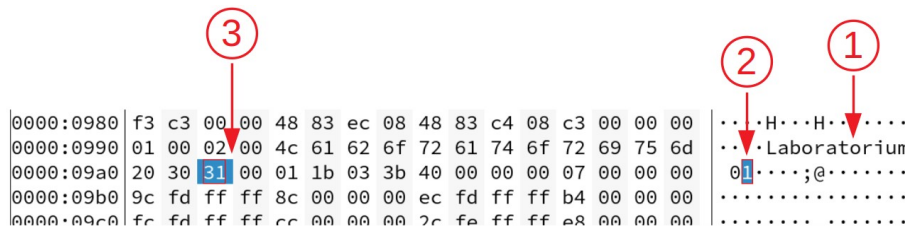
Wykonaj polecenia:

- Wybierz: Panel skrótów (B) → Projekty → Pojawia się Ustawienia budowania.
- Zmień konfigurację: Edycja konfiguracji budowania → Wybierz: Release.
- Zbuduj: Menu główne → Budowanie → Zbuduj projekt `Laboratorium0`.
- Odszukaj katalog w systemie plików, który zawiera pliki będące efektem ostatniego procesu budowania: katalog znajduje się obok poprzedniego katalogu `build-...`, ale w nazwie zawiera słowo **Release** (pierwszy zawiera słowa Debug).
- Wejdź do tego katalogu i uruchom program w konsoli jak poprzednio.
- Przywróć poprzednią konfigurację: Panel skrótów (B) → Projekty → Ustawienia budowania → Edycja konfiguracji budowania → Debug.
- Wrócić do trybu edycji: Panel skrótów (B) → Edycja.

## Zadanie 2.6. \*Modyfikacja pliku binarnego programu (zadania z \* są nieobowiązkowe)

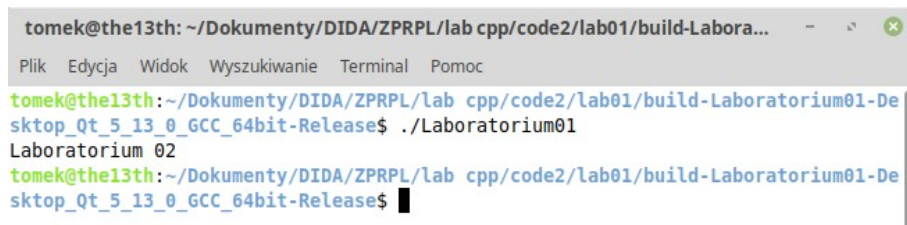
Wykonaj polecenia:

- Otwórz w edytorze plik binarny Laboratorium01 z katalogu **build-...-Release**.
- Wyszukaj fragment ze słowem „Laboratorium” z lewej strony edytora (1) jak na rysunku 1.4.



Rys. 1.4. Podgląd pliku Laboratorium01 w reprezentacji szesnastkowej

- Kliknij cyfrę 1 w napisie „01” (2).
- Zmień ostrożnie (!) wartość w miejscu (3) z 31 na 32. Nie zmieniaj nic innego!
- Zapisz zmieniony plik: Ctrl + S.
- Uruchom program w konsoli bezpośrednio z katalogu **build-...-Release**.



Rys.1.5. Wynik działania programu Laboratorium 01 po modyfikacji pliku binarnego

- Sprawdź czy rzeczywiście wyświetla się napis „Laboratorium 02”.
- UWAGA! Modyfikowanie innego oprogramowania w taki i podobny sposób może być niezgodne z prawem.

## Zadanie 2.7. Zachowanie projektu

Wykonaj polecenia:

- Zapisz pliki projektu: Menu główne → Plik → Zachowaj Wszystko.
- Zamknij środowisko Qt Creator: Menu główne → Plik → Zakończ.
- Zarchiwizuj katalog projektu: **Przenieść pliki w bezpieczne miejsce, tak aby można było projekt otworzyć na każdych następnych zajęciach.**

## LABORATORIUM 2. ZMIENNE I WYRAŻENIA, INSTRUKCJE WARUNKOWE I ITERACYJNE. REFERENCJE I WSKAŹNIKI (CZ.1).

### Cel laboratorium:

Pierwszym celem laboratorium jest wprowadzenie do programowania w języku C++ dla osób, które znają język C. Przykłady dobrane są tak, aby można było łatwo zauważyć podstawowe podobieństwa i różnice pomiędzy językami C i C++. Drugim celem jest przedstawienie klas i obiektów języka C++ jako sposobu organizacji kodu.

### Zakres tematyczny zajęć:

- Wprowadzenie do kodowania w języku C++.
- Zastosowanie klas i obiektów jako sposobu organizacji kodu (hermetyzacja).
- Zmienne, typy zmiennych, rzutowanie.
- Instrukcja warunkowe: `If ... else, switch ... case`.
- Instrukcje iteracyjne: `for, while, do...while`.
- Generowanie liczb losowych.
- Użycie wskaźnika i referencji jako alternatywny sposób dostępu do zmiennej.

### Kompendium wiedzy:

- Środowisko *Qt Creator* domyślnie generuje szkielet programu w języku C++ jak na listingu 2.1. Tekst ten znajduje się w pliku `main.cpp`. Możemy potraktować ten szkielet jako najprostszy program w języku C++.

Listing 2.1. Szkielet pliku `main.cpp` generowany przez *Qt Creator*

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Hello World !" << endl;
7     return 0;
8 }
```

- Wyprowadzanie znaków tekstowych na ekran odbywa się za pomocą obiektu `cout` do którego stosujemy operator wstawiania `<<` za pomocą którego obiektowi `cout` przekazujemy: łańcuchy tekstowe, znaki, zmienne, a nawet inne obiekty (wiersz 6). Wszystko co zostanie przekazane jest konwertowane na znaki `char`. `Cout` jest skrótem od `Character Out`.
- Obiekt `cout` jest tworzony automatycznie i dostępny po zaimportowaniu pliku nagłówkowego `iostream` dyrektywą `#include <iostream>` (wiersz 1).
- Jeżeli nie zostanie użyte: `using namespace std` (wiersz 2), wszędzie w kodzie należy stosować pełną ścieżkę nazwy obiektu `std::cout`. Nazwy w języku C++ mogą być grupowane w tzw. **przestrzenie nazw**, a nazwa `cout` znajduje się w przestrzeni `std`.
- Słowo `endl` (End Line) oznacza to samo co `'\n'`, czyli przejście do następnej linii.



- Wprowadzenie danych z klawiatury odbywa się za pomocą obiektu `cin` (Character Input) z użyciem operatora `>>`. Obiekt `cin` jest również tworzony automatycznie i znajduje się w przestrzeni nazw `std`, a dostęp do niego uzyskuje się poprzez import pliku `iostream`.
- Język C++ jest językiem **typowanym statycznie**, tzn. nadawanie typów zmiennym odbywa się w czasie kompilacji kodu programu. Istnieją języki **typowane dynamicznie**, wówczas typy zmiennych określane są na etapie wykonywania. Wachlarz dostępnych wbudowanych **typów danych** w języku C++ znaleźć można m.in. na stronach:  
<https://en.cppreference.com/w/cpp/language/types>  
<http://www.cplusplus.com/doc/tutorial/variables/>
- Instrukcje iteracyjne: `for`, `while`, `do...while` pochodzą z języka C i działają tak samo.
- **Klasę** języka C++ możemy potraktować jako pudełko na funkcje i zmienne. Funkcje te nie różnią się od „zwykłych” funkcji, ale dla wskazania, że nie są to funkcje „swobodne” nazywa się je **metodami**. Natomiast zmienne w klasie nazywa się **polami**. Ponieważ klasa opakowuje zmienne i funkcje, mówimy, że następuje **hermetyzacja**.
- **Klasa jest typem**. Tzn. należy utworzyć zmienną danego typu klasowego, aby używać metod i pól klasy. Taką zmienną wówczas nazywa się **obiektem** (klasy), aby wskazać, że ta zmienna jest bardziej zaawansowana.

### Pytania kontrolne:

1. Wyjaśnij jaka jest różnica między statycznym a dynamicznym typowaniem.
2. Co w kontekście programowania oznacza hermetyzacja.
3. Jaką wartość będzie miała zmienna `k` we wskazanych miejscach na listingach 2.2 i 2.3.

Listing 2.2. Instrukcje iteracyjne do analizy

```
1  int k = 0;
2  while(k<10) if(k<3) k++; else k+=2;
3  //k=?
4
5  k=13;
6  do{
7      if(k<7) k -= 3;
8      else if(k >= 2) --k;
9  }while (k < 4 && k > 2);
10 //k=?
11
12 k=3;
13 switch (k) {
14     case 2: k *= 5;
15     case 3: k++;
16     case 1: k -= 3;
17     case 5: k = 3;
18     default: k =0;
19 } // k=?
```



```
20 k = 5;
21 for(int i=-1; i<5; i+=3) k--;
22 // k=?
```

4. Odpowiedz które deklaracje są niepoprawne na listingu 2.3.

Listing 2.3. Deklaracje zmiennych do analizy

```
1 unsigned int = -3;
2 short s = -34;
3 bool b = 2;
4 bool q = true;
5 char z = 't';
6 double x = -3.4;
7 long = 3.2;
8 unsigned double y = 3.5;
9 unsigned char f = 24;
```

### Zadanie 2.1. Kontynuacja pracy z istniejącym projektem

Wykonaj polecenia:

- Odszukaj katalog z projektem z poprzednich zajęć i upewnij się czy znajdują się tam wszystkie pliki.
- Uruchom środowisko *Qt Creator*.
- Otwórz projekt w środowisku *Qt Creator*: Menu główne → Plik → Otwórz plik lub projekt → Pojawia się okno Otwórz plik.
- Odszukaj katalog z projektem i wskaż plik: **labCpp.pro**.
- Sprawdź czy pliki projektu są widoczne w perspektywie Projekty.

### Zadanie 2.2. Utworzenie kasy ZLab02 (Zadania Laboratorium02)

Wykonaj polecenia:

- Utwórz nową klasę w projekcie labCpp: Menu Główne → Nowy plik lub projekt → Pojawia się okno *Nowy plik lub projekt – Qt Creator*.
- Wskaż szablon: Wybierz szablon → Pliki i klasy → C ++ → Klasa C++.
- Zatwierdź: → Przycisk Wybierz.
- Nazwij klasę: **ZLab02**.
- Nie wykonuj żadnych dołączeń (QObject, QWidget, ...).
- Sprawdź nazwy plików: nagłówkowego – powinno być: **zlab02.h**, i źródłowego – powinno być: **zlab02.cpp**.
- Sprawdź czy wygenerowana ścieżka prowadzi do katalogu z projektem.
- Zatwierdź: → Przycisk *Dalej* → Przejdźcie do etapu Podsumowanie.
- Sprawdź czy wszystko się zgadza, jeżeli tak to zatwierdź: → Przycisk *Zakończ*.
- Sprawdź czy w perspektywie *Projekty* nowe plik są widoczne.
- Sprawdź czy w katalogu projektu pojawiły się nowe pliki.

### Zadanie 2.3. Pozyskanie informacji na temat typów: `int` oraz `unsigned int`

W zadaniu zostanie stworzona metoda klasy `ZLab02`. Metoda będzie demonstrować typy: `int`, `unsigned int` w zakresie ich rozmiaru w bajtach, granic (`min`, `max`) oraz pojemności, tzn. ile różnych od siebie wartości jest w stanie opisać dany typ. W programie wykorzystano:

- Operator `sizeof`, który może być użyty zarówno do zmiennej jak i typu.
- Metody `min()` oraz `max()` klasy `numeric_limits`, która jest dostępna po włączeniu pliku `limits`. (Obie metody są statyczne dlatego uruchamiamy je operatorem `::`. Natomiast zapis: `numeric_limits<int>` potraktujemy jako synonim: `numeric_limits_for_int`.)
- Funkcja `static_cast<long>()` służy do zamiany typu `int` oraz `unsigned int` na typ `long`. (Funkcja działa w taki sposób, że w nawiasy okrągłe wpisujemy co ma być zamienione, natomiast w nawiasy ostre na jaki typ.)
- Operator `typeid`, który staje się dostępny po włączeniu pliku `typeinfo`, zwraca pewien obiekt i ten obiekt ma metodę `name()`, którą wywołujemy.

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab02.h`: → 2x kliknij plik w perspektywie Projekty.
- Dodaj deklarację metody publicznej o nazwie `zadanie_2_3`: jak na listingu 2.4.

Listing 2.4. Dodanie deklaracji metody `zadanie_2_3` do klasy `ZLab02` w pliku `zlab02.h`

```
1  #ifndef ZLAB02_H
2  #define ZLAB02_H
3
4  class ZLab02
5  {
6  public:
7      ZLab02();
8      void zadanie_2_3();
9  };
10
11 #endif // ZLAB02_H
```

- Wygeneruj definicję tej metody: → Prawy klik na nazwę metody → Refaktoryzacja → Dodaj definicję w `zlab02.cpp` (listing 2.5)

Listing 2.5. Szkielet definicji metody `zadanie_2_3` w pliku `zlab02.cpp`

```
1  #include "zlab02.h"
2
3  ZLab02::ZLab02()
4  {
5
6  }
7
8  void ZLab02::zadanie_2_3()
9  {
10
11 }
```

- W pliku `zlab02.cpp` włącz pliki nagłówkowe: `iostream` `limits` `typeinfo` i użyj przestrzeni nazw `std` (Listing 2.6)

Listing 2.6. Włączenie plików nagłówkowych w `zlab02.cpp`

```
1 #include "lab_typy_proste.h"
2 #include <iostream>
3 #include <limits>
4 #include <typeinfo>
5 using namespace std;
```

- Wpisz kod metody `zadanie_2_3`

Listing 2.7. Pełna definicja metody `zadanie_2_3` w pliku `zlab02.cpp`

```
1 void ZLab02::zadanie_2_3()
2 {
3     int a = -3;
4     unsigned int b;
5     b = 5;
6
7     int int_min = numeric_limits<int>::min();
8     int int_max = numeric_limits<int>::max();
9
10    unsigned int unsigned_int_min
11        = numeric_limits<unsigned int>::min();
12    unsigned int unsigned_int_max
13        = numeric_limits<unsigned int>::max();
14
15    long int_zakres =
16        static_cast<long>(int_max)
17        - static_cast<long>(int_min);
18
19    long unsigned_int_zakres =
20        static_cast<long>(unsigned_int_max)
21        - static_cast<long>(unsigned_int_min);
22
23    cout << "Zmienna c1 jest typu: " << typeid (a).name()
24         << " ma rozmiar: " << sizeof (a) << " bajtów"
25         << endl << "Zakres tej zmiennej to: " << int_min
26         << " -> " << int_max << ", to oznacza "
27         << int_zakres << " różnych wartości" << endl
28         << "Aktualna wartość zmiennej to: " << a
29         << endl << endl;
30
31    cout << "Zmienna c2 jest typu: " << typeid (b).name()
32         << " ma rozmiar: " << sizeof (b) << " bajtów"
33         << endl
34         << "Zakres tej zmiennej to: " << unsigned_int_min
35         << " -> " << unsigned_int_max << ", to oznacza "
```



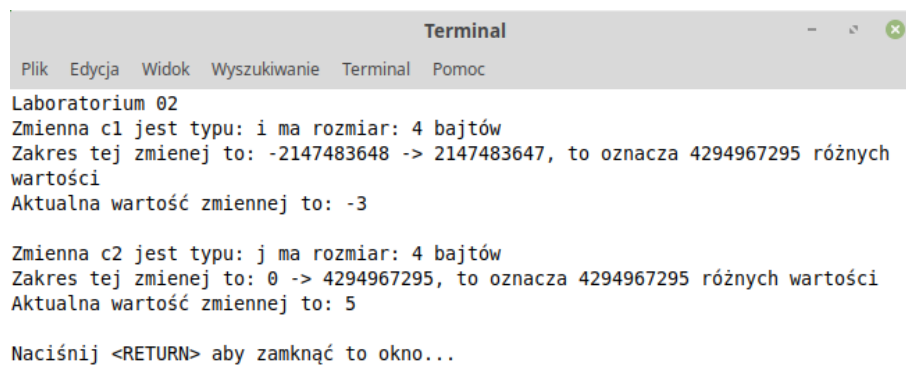
```
36         << unsigned_int_zakres << " różnych wartości"
37         << endl
38         << "Aktualna wartość zmiennej to: " << b << endl
39         << endl;
40     }
```

- Przejdź do pliku `main.cpp`.
- W pliku `main.cpp`: włącz plik nagłówkowy `zlab02.h`. Zmień treść wyświetlanego napisu. Utwórz obiekt klasy `ZLab02`, wywołaj metodę `zadanie_2_3` na utworzonym obiekcie (Listing 2.8).

Listing 2.8. Zawartość pliku `main.cpp`

```
1  #include <iostream>
2  #include "zlab02.h"
3  using namespace std;
4
5  int main()
6  {
7      cout << "Laboratorium 02" << endl;
8
9      ZLab02 lab02;
10     lab02.zadanie_2_3();
11
12     return 0;
13 }
```

- Uruchom Program. (Wynik programu jak na rys. 2.1)



```
Terminal
Plik  Edycja  Widok  Wyszukiwanie  Terminal  Pomoc

Laboratorium 02
Zmienna c1 jest typu: i ma rozmiar: 4 bajtów
Zakres tej zmiennej to: -2147483648 -> 2147483647, to oznacza 4294967295 różnych wartości
Aktualna wartość zmiennej to: -3

Zmienna c2 jest typu: j ma rozmiar: 4 bajtów
Zakres tej zmiennej to: 0 -> 4294967295, to oznacza 4294967295 różnych wartości
Aktualna wartość zmiennej to: 5

Naciśnij <RETURN> aby zamknąć to okno...
```

Rys. 2.1. Wynik działania metody `zadanie_2_3`

## Dyskusja – odpowiedz na pytania:

- Jakie rozszerzenie ma plik nagłówkowy i po co on jest?
- Jakie rozszerzenie ma plik źródłowy i po co on jest?
- Listing 2.6, wiersze 1,2,3,4 → Jaki jest efekt dyrektywy `include`?
- Co oznacza operator `::` (np. w linii 8 na listingu 2.5) ?
- Listing 2.7, wiersz 15 i 19 → Dlaczego zastosowany został typ `long`.
- Jaki jest efekt słowa `endl`?

#### Zadanie 2.4. Pobieranie danych od użytkownika. Konwersja typów.

W zadaniu zostanie stworzona metoda klasy ZLab02 o nazwie `zadanie_2_4` (na początku w nazwie będzie błąd literowy). Metoda pobierze od użytkownika dwie liczby całkowite, a następnie wykona dzielenie tych liczb. Dzielenie zostanie wykonane w zakresie typu `int`, w zakresie typu `double` ale bez rzutowanie typu `int` na `double` i ostatecznie z rzutowaniem typu `int` na `double`.

Wykonaj polecenia:

- W pliku `zlab02.h` zadeklaruj metodę: `void zadanie_2_4()`.  
!UWAGA: Błąd literowy jest zamierzony. Tak proszę zrobić.
- Wygeneruj szkielet metody w pliku `zlab02.cpp`.
- Wpisz kod metody jak na listingu 2.9.

Listing 2.9. Definicja metody w pliku źródłowym

```
10 void ZLab02::zadneie_2_4()
11 {
12     int a = 5, b = 3;
13     int c = a / b;
14     double d = a / b;
15     double e = static_cast<double>(a)/static_cast<double>(b);
16
17     cout << "a=" << a << " b=" << b << endl
18         << " a/b=" << c << endl
19         << " a/b=" << d << endl
20         << " a/b=" << e << endl;
21 }
```

- Zmień plik `main.cpp` tak, aby uruchamiała się nowa metoda (Listing 2.10).

Listing 2.10. Nowa zawartość pliku `main.cpp`

```
1 int main()
2 {
3     cout << "Laboratorium 02" << endl;
4
5     ZLab02 lab02;
6     //lab02.zadanie_2_3();
7     lab02.zadneie_2_4();
8
9     return 0;
10 }
```

- Uruchom program i zaobserwuj działanie (Rys. 2.2)



```
Terminal
Plik  Edycja  Widok  Wyszukiwanie  Terminal  Pomoc
Laboratorium 02
a=5 b=3
a/b=1
a/b=1
a/b=1.66667
Naciśnij <RETURN> aby zamknąć to okno...
```

Rys. 2.2. Wynik działania metody z zadania 2.4.

- Dopiero teraz zmień nazwę metody na **zadanie\_2\_4**: → Plik **zlab02.h** → Prawy klik na nazwę metody z błędem → Refaktoryzacja → Zmień nazwę symbolu pod kursorem (→ Spójrz w dół ekranu jeżeli nie ma edycji w miejscu kliknięcia)
- Sprawdź czy zaszła zmiana w plikach : **zlab02.cpp** oraz **main.cpp**.

### Dyskusja – odpowiedź na pytania:

- Dlaczego wyniki dzielenia są różne?
- Listing 2.9, wiersz 15 → Co tu się dzieje?

### Zadanie 2.5. Dzielenie liczb zmiennoprzecinkowych

W zadaniu zostanie stworzona metoda klasy **ZLab02**. Metoda wyświetli najmniejszą możliwą wartość liczby typu **double** jaka jest dostępna w systemie. Następnie pobierze dwie liczby zmiennoprzecinkowe i je podzieli, jeżeli dzielnik nie jest mniejszy niż ta najmniejsza wartość. W programie zastosowano metodę **precision** obiektu **cout**, która ustala liczbę miejsc po przecinku przy wyświetlaniu zmiennej.

Wykonaj polecenia:

- Dodaj do klasy metodę: **void zadanie\_2\_5()** i zdefiniuj ją wg listingu 2.11.

Listing 2.11. Definicja metody **zadanie\_2\_5**

```
1  #include <cmath>
2  void ZLab02::zadanie_2_5()
3  {
4      cout.precision(20);
5      double eps = numeric_limits<double>::min();
6      cout << "Dzielenie z dokładnością: " << eps << endl;
7      double x,y;
8      cout << "Podaj dzielną: ";
9      cin >> x;
10     cout << "Podaj dzielnik: ";
11     cin >> y;
12     if(fabs(y) > eps)
13         cout << x << "/" << y << "=" << x/y << endl;
14     else
15         cout << "Dzielnik jest za mały" << endl;
16 }
```



- Zmień funkcję `main` w pliku `main.cpp` tak aby uruchamiała się ta metoda.
- Wypróbuj dla liczb bezpiecznych np. 7 i 3 .
- Wypróbuj dla dzielnika bliskiego zeru np.  $1e-500$  .

### Dyskusja i eksperymenty

- Zmień metodę `zadanie_2_5` tak, aby dzielenie wykonywało się bez kontroli dzielnika. Czy działanie programu w tej postaci jest akceptowalne?
- Wy tłumacz jak to się dzieje, że podana przez użytkownika wartość dzielnika poniżej minimalnej wartości typu `double` jest przyjmowana przez program.
- Opisz co oznacza wartość minimalna typu dla `double` i dla `int` .
- Wyjaśnij czy bezpiecznie jest porównywać liczbę typu `double` do zera.

### Zadanie 2.6. Menu programu

W zadaniu zostanie stworzona metoda klasy `ZLab02` o nazwie `menu`, która pozwoli użytkownikowi wybrać procedurę do uruchomienia. Użytkownik wprowadzi liczbę 3, 4, 5, następnie za pomocą instrukcji wyboru `switch`, zostanie uruchomiana właściwa procedura. Wprowadzenie liczby przez użytkownika odbywa się w pętli `do...while`, która kończy powtarzanie, gdy podana liczba równa się 3, 4 lub 5.

Wykonać polecenia:

- Do klasy `ZLab02` dodaj metodę `void ZLab02::menu()` i zdefiniuj ją wg listingu 2.12.

Listing 2.12. Definicja metoda menu klasy `ZLab02`

```
1 void ZLab02::menu()
2 {
3     unsigned short wybor;
4
5     do{
6         cout << "Które zadanie uruchomić?" << endl
7             << "3 -> zadanie 2.3" << endl
8             << "4 -> zadanie 2.4" << endl
9             << "5 -> zadanie 2.5" << endl;
10        cin >> wybor;
11    }while(wybor < 3 || wybor > 5);
12
13    switch (wybor) {
14        case 3 : zadanie_2_3(); break;
15        case 4 : zadanie_2_4(); break;
16        case 5 : zadanie_2_5(); break;
17        default: cout << "Nie ma takiego zadania" << endl;
18    }
19 }
```



- Zmień plik `main.cpp` tak, aby uruchamiała się metoda `menu` (Listing 2.13).

Listing 2.13. Modyfikacja pliku `main.cpp` po zdefiniowaniu metody `menu`

```
1 int main()
2 {
3     cout << "Laboratorium 02" << endl;
4
5     ZLab02 lab02;
6     lab02.menu();
7
8     return 0;
9 }
```

- Uruchom program i wypróbuj jego działanie .
- !UWAGA: Na tym etapie wprowadzać tylko liczby.

### Dyskusja – odpowiedź na pytania:

- Listing 2.12, wiersz 3 → Czy zmienna `wybor` może mieć typ `double` ?
- Listing 2.13, wiersz 3 → Czy zmienna `wybor` może mieć typ `int`?
- Listing 2.13, wiersz 11 → Co oznacza `II` ?
- Listing 2.13, wiersze 14, 15, 16 → Jaka jest rola słowa `break`?

### Zadanie 2.7. Wskaźniki jako alternatywny sposób dostępu do zmiennych

W zadaniu zostanie zademonstrowane użycie wskaźnika jako alternatywny sposób dostępu do zmiennej.

Wykonaj polecenia:

- Do klasy `ZLab02` dodaj metodę `void zadanie_2_6()` wg listingu 2.14.
- Uruchom i zaobserwuj działanie programu.

Listing 2.14. Definicja metody `zadanie_2_6` klasy `ZLab02`

```
1 void ZLab02::zadanie_2_7()
2 {
3     long liczba1 = 3, liczba2 = -5, liczba3 = 0;
4
5     long *wskaznikDoLong;
6     wskaznikDoLong = nullptr;
7
8     cout << "liczba1=" << liczba1
9         << " liczba2=" << liczba2
10        << " liczba3=" << liczba3 << endl;
11
12     wskaznikDoLong = &liczba2;
13
14     *wskaznikDoLong = 10;
```



```
15
16     cout << "liczba1=" << liczba1
17           << " liczba2=" << liczba2
18           << " liczba3=" << liczba3 << endl;
19
20     wskaznikDoLong = &liczba3;
21
22     *wskaznikDoLong = liczba1;
23
24     cout << "liczba1=" << liczba1
25           << " liczba2=" << liczba2
26           << " liczba3=" << liczba3 << endl;
27
28     wskaznikDoLong = &liczba1;
29
30     liczba2 = *wskaznikDoLong;
31
32     cout << "liczba1=" << liczba1
33           << " liczba2=" << liczba2
34           << " liczba3=" << liczba3 << endl;
35 }
```

#### Dyskusja – odpowiedz na pytania:

- Listing 2.14, wiersz 6 → W jakim celu wskaźnikowi nadaje się wartość `nullptr`?
- Listing 2.14, wiersz 12 → Jak działa operator `&` ?
- Listing 2.14, wiersz 14 → Po co jest znak `*` ?

#### Zadanie 2.8. Referencje jako alternatywny sposób dostępu do zmiennych

W zadaniu zostanie zademonstrowana referencja do zmiennej jako alternatywny sposób dostępu do tej zmiennej. Podstawowe różnice między referencją a wskaźnikiem to:

- Referencje ustawia się od razu na wskazywaną zmienną i nie można tego zmienić.
- Przy deklaracji typu używa się symbolu `&` zamiast `*`.
- Przy odwoływaniu się do obiektu przez referencje używa się jedynie nazwy referencji. (W przypadku wskaźnika potrzebny był znak `*`).

Wykonaj polecenia:

- Do klasy `ZLab02` dodaj metodę `void zadanie_2_8()` i zdefiniuj ją wg listingu 2.15.
- Uruchom metodę i zaobserwuj działanie.

Listing 2.15. Definicja metody `zadanie_2_8` klasy `ZLab02`

```

1 void ZLab02::zadanie_2_8()
2 {
3     long liczba1 = 3, liczba2 = -5, liczba3 = 0;
4
5     cout << "liczba1=" << liczba1
6         << " liczba2=" << liczba2
7         << " liczba3=" << liczba3 << endl;
8
9     long &referencjaDoLong = liczba2;
10
11     referencjaDoLong = 10;
12
13     cout << "liczba1=" << liczba1
14         << " liczba2=" << liczba2
15         << " liczba3=" << liczba3 << endl;
16
17     long &referencjaDoLongInna = liczba3;
18
19     referencjaDoLongInna = liczba1;
20
21     cout << "liczba1=" << liczba1
22         << " liczba2=" << liczba2
23         << " liczba3=" << liczba3 << endl;
24
25     long &referencjaDoLongKolejna = liczba1;
26
27     liczba2 = referencjaDoLongKolejna;
28
29     cout << "liczba1=" << liczba1
30         << " liczba2=" << liczba2
31         << " liczba3=" << liczba3 << endl;
32 }

```

### Zadanie 2.9. Generowanie liczb pseudolosowych

W zadaniu zostanie zademonstrowany najprostszy sposób generowania liczb losowych. W tym celu zostanie wykorzystany:

- obiekt klasy `random_device` (generuje liczby pseudolosowe).
- obiekt klasy `uniform_int_distribution` (rzutuje liczby na podany zakres)

Oba obiekty stają się dostępne po włączeniu pliku nagłówkowego `random`. Klasa `uniform_int_distribution` wymaga dodatkowego ustawienia poprzez podanie typu w nawiasach ostre. Podany typ oznacza typ zwracanego wyniku. (Tutaj losowane są liczby typu `short`).

Wykonaj polecenia:

- Do klasy `ZLab02` dodaj metodę `void zadanie_2_9()` i zdefiniuj ją wg listingu 2.16.
- Uruchom metodę i zaobserwuj działanie.





Listing 2.16. Definicja metody zadanie\_2\_9

```

1  #include <random>
2  void ZLab02::zadanie_2_9()
3  {
4      const short N = 10;
5      int tablica[N];
6
7      random_device maszynkaLosujaca;
8      uniform_int_distribution<short> dystrybucja(-3,4);
9
10     for(unsigned short i=0; i<N; i++)
11         tablica[i] = dystrybucja(maszynkaLosujaca);
12
13     for(int j : tablica) cout << j << " " ;
14
15     cout << endl;
16 }

```

**Zadanie 2.10. Wygenerowanie pseudolosowego ciągu niemalejącego**

Celem zadania jest wygenerowanie liczb pseudolosowych, które utworzą ciąg niemalejący. Liczby losowe będą wybierane z zadanego przedziału od min do max. Liczby zostaną zapisane w tablicy. Rozwiązanie polega na wylosowaniu pierwszej liczby. Kolejną liczbę losujemy do momentu, aż nie będzie mniejsza od poprzedniej. Wówczas przechodzimy do losowania następnej w ten sam sposób. Zewnętrzna pętla `for` (Listing 2.17 wiersz 11) wskazuje drugą i kolejne liczby ciągu do ustalenia. Pierwsza liczba ciągu jest ustalona (przez losowanie) przed pętlą. Wewnętrzna pętla `while` (Listing 2.17 wiersz 12) losuje, aż do skutku.

Wykonaj polecenia:

- Do klasy `ZLab02` dodaj metodę `void zadanie_2_10()` i ją uruchom.

Listing 2.17. Definicja metody zadanie\_2\_10 klasy ZLab02

```

1  void ZLab02::zadanie_2_10(short min, short max)
2  {
3      const short N = 10;
4      int tab[N];
5
6      random_device maszynka;
7      uniform_int_distribution<short> dystr(min,max);
8
9      tab[0] = dystr(maszynka);
10
11     for(unsigned short i=1; i<N; i++)
12         while( (tab[i]=dystr(maszynka) ) < tab[i-1] );
13
14     for(int liczba : tab) cout << liczba << " " ;
15 }

```



### Dyskusja – odpowiedz na pytania:

- Listing 2.17, wiersz 4 → Jaki będzie efekt wywołania instrukcji?
- Listing 2.17, wiersz 7 → Co tu się dzieje?
- Listing 2.16, wiersz 12 → Jakie instrukcje znajdują się w bloku pętli **while** ?
- Listing 2.16, wiersz 12 → Jak w ogóle działa pętla **while** ?
- Czy może zdarzyć się, że warunek zakończenia pętli **while** nie wystąpi?
- Czy zaproponowany algorytm można zastosować do wygenerowania ciągu rosnącego?

### Zadanie 2.11. \*Bezpieczne pobieranie liczby od użytkownika

Listing 2.18 pokazuje sposób bezpiecznego pobierania liczby od użytkownika „z klawiatury”. Sposób polega na przyjęciu od użytkownika dowolnego tekstu (wiersz 9), następnie za pomocą funkcji `stod` (string to double), zostaje podjęta próba zamiany tekstu na liczbę. Funkcja ta jest wywoływana w bloku `try...catch`, który przechwytuje zgłoszenie błędu (wyjątku) w przypadku niewłaściwych danych. Przechwycenie wyjątku pozwala na jego zignorowanie i powtórzeniu próby aż do skutku.

Listing 2.18. Definicja metody `pobierzLiczbe` klasy `ZLab02`

```
1  double ZLab02::pobierzLiczbe()
2  {
3      double x = 0;
4      bool sukces = true;
5      string linia;
6
7      do{
8          if(!sukces) cout << "!To ma być liczba." << endl;
9          getline(cin, linia);
10         try {
11             x = stod(linia);
12             sukces = true;
13         } catch (const invalid_argument&) {
14             sukces = false;
15         }
16     }while(!sukces);
17
18     return x;
19 }
```

### Zadanie 2.12. Modyfikacja metody `menu`

Zmodyfikuj metodę `menu` klasy `ZLab02`, tak aby uwzględniała wszystkie metody zdefiniowane w klasie.



## **LABORATORIUM 3. KLASY I OBIEKTY, KONSTRUKTORY I DESTRUKTORY (Cz.1).**

### **Cel laboratorium:**

Celem laboratorium jest nabranie praktyki w definiowaniu klas, tworzeniu obiektów klas i posługiwaniu się tymi obiektami. Dodatkowym celem laboratorium jest stworzenie obiektu *Organizm*, który jest składnikiem aplikacji *Wirtualny ekosystem*.

### **Zakres tematyczny zajęć:**

- Klasy i obiekty jako wydzielona część kodu logicznie grupująca dane i operacje. (abstrakcja).
- Szkielet klasy, operator dostępu.
- Pola i metody klasy.
- Składniki prywatne i publiczne klasy.
- Konstruktor obiektu.
- Lista inicjalizacyjna konstruktora.
- Konstruktor domyślny.
- Konstruktor kopiujący.
- Destruktor obiektu.

### **Kompendium wiedzy:**

- W poprzednim ćwiczeniu klasa została użyta jako pojemnik na funkcje. Dzięki takiej organizacji kodu funkcja uruchomieniowa **main** była krótka, a kod programu czytelnie pogrupowany. Jednak użyteczność klas i obiektów jest znacznie szersza.
- W tym ćwiczeniu obiekty posiadają własne **pola** i **metody**, które na tych polach wykonują operacje. W związku z tym obiekt posiada **stan** i **zachowania**. Obiekt również posiada swoją **tożsamość**, ponieważ każdy utworzony obiekt jest odrębny.
- Klasę możemy traktować jako projekt na podstawie którego tworzy się każdy egzemplarz obiektu. Przypomina ona strukturę z języka C.
- Tworzenie obiektów jest możliwe tylko wtedy kiedy została definiowana klasa.
- Klasa najczęściej jest opisywana za pomocą dwóch plików: **nagłówkowego** (z rozszerzeniem **.h**) oraz **źródłowego** (z rozszerzeniem **.cpp**).
- W pliku nagłówkowym znajdują się **definicja klasy** czyli informacja jak ta klasa wygląda.
- W pliku źródłowym znajdują się **definicje metod klasy** i ewentualnie definicje składników statycznych.
- Szkielet definicji klasy został przedstawiony na listingu 3.1.
- Klasa może posiadać składniki **prywatne** (wiersze: 4,5) i **publiczne** (wiersze: 8,9, ...,15). Składniki prywatne są dostępne wyłącznie dla metod wewnętrznych klasy zarówno prywatnych jak i publicznych.
- Klasa oprócz zwykłych metod (czyli funkcji przypisanych do klasy) może posiadać specjalne metody: **konstruktory** (wiersze: 10,11,12) i **destruktory** (wiersz: 13).
- Konstruktor uruchamiany jest w momencie tworzenia nowego egzemplarza obiektu. Istnieje możliwość zdefiniowania wielu różnych konstruktorów. Obiekt może być stworzony na wiele sposobów. Konstruktor może posiadać argumenty.



Listing 3.1. Przykładowy szkielet definicji klasy

```

1  class NazwaKlasy
2  {
3  private:
4      double nazwaZmiennejPrywatnej;
5      void nazwaMetodyPrywatnej();
6
7  public:
8      int nazwaZmiennejPublicznej;
9
10     NazwaKlasy();
11     NazwaKlasy(double x);
12     NazwaKlasy(const NazwaKlasy& nazwaZmiennej);
13     ~NazwaKlasy();
14
15     void nazwaMetodyPublicznej();
16 };
    
```

- Destruktor uruchamiany jest przed zniszczeniem obiektu, tzn. usunięciem go z pamięci. Destruktor może być tylko jeden. Destruktor nie posiada argumentów. Obiekt może być zniszczony tylko w jeden sposób.
- Konstruktor z wiersza 12 nazywa się **konstruktorem kopiującym**. Poznajemy go po parametrze, tj. referencji do obiektu tej samej klasy. Służy do tworzenia obiektu poprzez skopiowanie innego obiektu swojej klasy.
- Konstruktor bez parametrów z wiersza 10 nazywa się **konstruktorem domyślnym**.
- Konstruktor domyślnym może być również konstruktor z parametrami, jeżeli wszystkie parametry mają wartości domyślne.
- Jeżeli wprost nie zostanie zdefiniowany konstruktor klasy, wtedy zostanie wygenerowany konstruktor automatyczny bez parametrów, czyli domyślny.
- Jeżeli wprost nie zostanie zdefiniowany konstruktor kopiujący, wtedy zostanie wygenerowany domyślny konstruktor kopiujący. Konstruktor taki kopiuje wartości pól klasy do nowego obiektu.
- W pliku źródłowym składniki klasy (pola i metody) wskazuje się poprzedzając ich nazwy nazwą klasy. Wykorzystuje się tutaj **operator zakresu ::** (2x dwukropek) .

### Pytania kontrolne:

1. Wyjaśnij jak jest różnica pomiędzy klasą i obiektem klasy.
2. Odpowiedz co może być składnikiem klasy.
3. Opisz proces tworzenia obiektu.
4. Nazwij metodę, która uruchamia się przy usuwaniu obiektu z pamięci.

### Zadanie 3.1. Stworzenie klasy **Prostokat**

Celem zadania jest stworzenie klasy **Prostokat**. (Klasa **Prostokat** nie jest częścią aplikacji *Wirtualny Ekosystem*. Jest to dodatkowy przykład dydaktyczny.) Każdy obiekt tej klasy przechowuje swoją nazwę, długości boków, informacje czy długości boków są poprawne. Obiekt również oblicza swój obwód i pole. Klasa posiada konstruktor i destruktory. Publiczne metody klasy umożliwiają dostęp do składników prywatnych.



Wykonaj polecenia:

- Otwórz projekt z poprzednich zajęć.
- Dodaj do projektu klasę ZLab03.
- Sprawdź czy pojawiły się nowe pliki w projekcie: `zlab03.h` oraz `zlab03.cpp`.
- Otwórz plik `zlab03.h` i usuń definicję klasy ZLab03, pozostaw dyrektywy (#).
- Otwórz plik `zlab03.cpp` i usuń deklaracje konstruktora `ZLab03::ZLab03()`, pozostaw włączenie pliku `zlab03.h`.
- Otwórz ponownie plik `zlab03.h` i dodaj definicję klasy **Prostokat** (Listing 3.2).

Listing 3.2. Definicja klasy Prostokat

```
1  #ifndef ZLAB03_H
2  #define ZLAB03_H
3
4  #include <string>
5  using namespace std;
6
7  class Prostokat
8  {
9  private:
10     string nazwa;
11     double bok1, bok2;
12     bool poprawny;
13     double obwod, pole;
14
15     bool czyPoprawny(double bok1, double bok2);
16     void obliczObwod();
17     void obliczPole();
18
19 public:
20     Prostokat(string n = "?", double a=1, double b=2);
21     ~Prostokat();
22     bool czyPoprawny();
23     const string& jakaNazwa();
24     double podajPole();
25     double podajObwod();
26     bool zmienBoki(double a, double b);
27
28     std::string doTekstu();
29 };
30
31 #endif // ZLAB03_H
```

- Dodaj szkielety metod klasy **Prostokat** w pliku źródłowym (Prawy klik na nazwę metody np. `czyPoprawny` → Refaktoryzacja → Dodaj definicję w `zlab03.cpp`).
- Uzupełnij metody wg poniższych listingów od 3.3 do 3.13.
- Zwróć uwagę, że definicja konstruktora `Prostokat(string, double, double)` używa listy inicjalizacyjnej (Listing 3.6 wiersz 2). Jest to podstawienie wartości pod zmienną. Tak jakby wywołanie konstruktora tej zmiennej.



Listing 3.3. Definicja metody czyPoprawny klasy Prostokat

```
1 bool Prostokat::czyPoprawny(double bok1, double bok2)
2 {
3     return bok1 > 0 && bok2 > 0;
4 }
```

Listing 3.4. Definicja metody obliczObwod klasy Prostokat

```
1 void Prostokat::obliczObwod()
2 {
3     obwod = 2 * ( bok1 + bok2 );
4 }
```

Listing 3.5. Definicja metody obliczPole klasy Prostokat

```
1 void Prostokat::obliczPole()
2 {
3     pole = bok1 * bok2;
4 }
```

Listing 3.6. Definicja konstruktora klasy Prostokat

```
1 Prostokat::Prostokat(string n, double a, double b)
2     :nazwa(n)
3 {
4     poprawny = czyPoprawny(a,b);
5
6     if(poprawny){
7         bok1 = a;
8         bok2 = b;
9         obliczPole();
10        obliczObwod();
11    }else{
12        bok1 = bok2 = pole = obwod = 0;
13    }
14 }
```

Listing 3.7. Definicja destruktoru klasy Prostokat

```
1 #include <iostream>
2 Prostokat::~Prostokat()
3 {
4     std::cout << "Prostokąt: " << nazwa
5               << " znika." << std::endl;
6 }
```



Listing 3.8. Definicja drugiej metody czyPoprawny klasy Prostokat

```
1 bool Prostokat::czyPoprawny()  
2 {  
3     return poprawny;  
4 }
```

Listing 3.9. Definicja metody jakaNazwa klasy Prostokat

```
1 const std::string &Prostokat::jakaNazwa()  
2 {  
3     return nazwa;  
4 }
```

Listing 3.10. Definicja metody podajPole klasy Prostokat

```
1 double Prostokat::podajPole()  
2 {  
3     return pole;  
4 }
```

Listing 3.11. Definicja metody podajObwod klasy Prostokat

```
1 double Prostokat::podajObwod()  
2 {  
3     return obwod;  
4 }
```

Listing 3.12. Definicja metody zmienBoki klasy Prostokat

```
1 bool Prostokat::zmienBoki(double a, double b)  
2 {  
3     if(czyPoprawny(a,b)){  
4         poprawny = true;  
5         bok1 = a;  
6         bok2 = b;  
7         obliczPole();  
8         obliczObwod();  
9         return true;  
10    } return false;  
11 }
```





Listing 3.13. Definicja metody doTekstu klasy Prostokat

```

1  std::string Prostokat::doTekstu()
2  {
3      std::string napis = "";
4
5      napis = "Prostokąt o nazwie: " + nazwa
6              + " bok1=" + to_string(bok1)
7              + " bok2=" + to_string(bok2);
8
9      if(poprawny) napis += " obwód=" + to_string(obwod)
10                 + " pole=" + to_string(pole);
11     else napis += " !Figura niepoprawna.";
12
13     return napis;
14 }

```

#### Dyskusja – odpowiedz na pytania:

- Listing 3.2 wiersz 10 → Jakie wartości można podstawić pod zmienną **nazwa** ?
- Listing 3.2, wiersz 11 → Czy w funkcji **main** można podstawić pod zmienną **bok1** wartość np. poleceniem **obiektKlasy.bok1 = 3** ?
- Listing 3.2, wiersz 15 → Jaką wartość zwraca metoda **czyPoprawny** ?
- Listing 3.2, wiersz 16 → Czy w funkcji **main** można wywołać metodę **obliczObwod** poleceniem **obiektKlasy.obliczObwod()** ?
- Listing 3.2, wiersz 20 → Jaki jest efekt użycia domyślnych wartości argumentów metody?
- Listing 3.2, wiersz 15 oraz wiersz 22 → Czym różnią się metody? Jak nazywa się taki mechanizm?
- Listing 3.2, wiersz 23 → Co zwraca metoda?
- Listing 3.2, wiersz 28 → Co oznacza **std::** ? Czy jest to konieczne?
- Listing 3.3, wiersz 1 → Co oznacza zapis **Prostokat::** i po co on jest?
- Listing 3.3, wiersz 3 → Skąd bierze się zwracana przez metodę wartość?
- Listing 3.6, wiersz 2 → co oznacza zapis **:nazwa(n)** i jaki jest jego efekt? Czy można zastąpić go przypisaniem **nazwa = n** w bloku { } ?
- Listing 3.6, wiersz 12 → Jaką wartość będzie miało pole **bok2** ?
- Listing 3.7 → Czy użycie **std::** jest tutaj konieczne?
- Listing 3.12 → Jak działa metoda? Dlaczego tutaj można wywoływać metody prywatne klasy?
- Listing 3.13, wiersz 6, 7, 9 → Co robi funkcja biblioteczna **to\_string** ?

#### Zadanie 3.2. Stworzenie, uruchomienie i testowanie obiektów klasy **Prostokat**

Celem zadania jest stworzenie 3 obiektów klasy **Prostokat** i praktyka posługiwania się nimi.

Wykonaj polecenia:

- Zmień plik **main.cpp** wg listingu 3.14. Następnie uruchom program i zaobserwuj działanie. Własne eksperymenty mile widziane.



Fundusze Europejskie  
Wiedza Edukacja Rozwój



Rzeczpospolita  
Polska

Unia Europejska  
Europejski Fundusz Społeczny



Listing 3.14. Zawartość pliku main.cpp w zadaniu 3.2.

```
1  #include <iostream>
2  #include "zlab03.h"
3  using namespace std;
4
5  int main()
6  {
7      cout << "Laboratorium 03" << endl;
8
9      Prostokat p1;
10     Prostokat p2("koc",1.4,2.2);
11     Prostokat p3("pułapka",-2,-3);
12
13     cout.precision(2);
14     cout << p1.doTekstu() << endl;
15     cout << p2.doTekstu() << endl;
16     cout << p3.doTekstu() << endl;
17
18     p3.zmienBoki(2,3);
19     cout << endl << p3.doTekstu() << endl;
20
21     cout << endl << "Użycie metod obiektu:" << endl;
22     cout << "Nazwa: " << p2.jakaNazwa() << endl
23         << "Poprawny: " << p2.czyPoprawny() << endl
24         << "Obwód=" << p2.podajObwod() << endl
25         << "Pole=" << p2.podajPole() << endl;
26
27     return 0;
28 }
```

#### Dyskusja – odpowiedź na pytania:

- Listing 3.14, wiersz 14,15,16 → W jaki sposób uzyskuje się dostęp do pola lub metody obiektu mając jego nazwę?
- Listing 3.14, wiersz 9 → Jakie wartości przyjmuje konstruktor?
- Listing 3.14, wiersz 13 → Co robi metoda **precision** wywołana na obiekcie automatycznym **cout** ?
- Listing 3.14, wiersze 14, 15, 16 → Dlaczego efekt metody **precision** nie dotyczy tych poleceń?
- Listing 3.14, wiersze 21-25 → Ile instrukcji języka C++ jest w tych wierszach?
- Wyjaśnij czy istnieje możliwość zmiany nazwy obiektu klasy prostokąt, np.: **p1.nazwa = "Inna nazwa"** ?
- Listing 3.14 → W definicji metody nie ma wywołania destruktora obiektu. Czy destruktor się uruchamia?



### Zadanie 3.3. Stworzenie klasy **Organizm**

Klasa **Organizm** jest częścią aplikacji *Wirtualny ekosystem*. Klasa **organism** obejmuje wspólne cechy i funkcjonalności 3 różnych gatunków wirtualnych organizmów w symulacji.

Wykonaj polecenia:

- Dodaj do projektu klasę o nazwie **Organizm**.
- Uzupełnij definicję klasy w pliku **organism.h** → Listing 3.15.
- Uzupełnij definicje metod w pliku **organism.cpp** → Listingi od 2.16 do 3.19
- Zwrócić uwagę na definicję metod w pliku nagłówkowym (**.h**). Jest to alternatywny sposób definiowania metod. Można go stosować do krótkich metod.

Listing 3.15. Definicja klasy *Organizm*

```
1  class Organizm{
2
3  public:
4      const unsigned short limitPosilkow;
5      const unsigned short kosztPotomka;
6
7  private:
8      unsigned short licznikZycia;
9      unsigned short licznikPosilkow;
10
11 public:
12     Organizm(unsigned short dlugoscZycia,
13              unsigned short limitPosilkow,
14              unsigned short kosztPotomka);
15
16     bool zywy() const
17     {return licznikZycia > 0;}
18
19     bool glodny() const
20     {return zywy() && licznikPosilkow < limitPosilkow;}
21
22     bool paczkujacy() const
23     {return zywy() && licznikPosilkow > kosztPotomka;}
24
25     unsigned short stanLicznikaZycia() const
26     {return licznikZycia;}
27
28     unsigned short stanLicznikaPosilkow() const
29     {return licznikPosilkow;}
30
31     bool posilek();
32     bool potomek();
33
34     void krokSymulacji();
35 };
```



Klasa **Organizm** jest zaprojektowana w taki sposób, że obiekty tej klasy posiadają wewnętrzny **licznikZycia**, któremu nadawana jest wartość dodatnia w momencie tworzenia obiektu. Wartość tego licznika zmniejszana jest w każdym kroku symulacji i gdy osiągnie wartość 0 organizm jest wirtualnie martwy. Zmniejszanie licznika realizuje **metoda krokSymulacji**. Drugim licznikiem jest **licznikPosilkow**, który w konstruktorze jest ustawiany na wartość 0, co oznacza, że organizm jest wirtualnie głodny. Każde wywołanie metody **posilek** zwiększa ten licznik o 1, aż wartość tego licznika osiągnie wartość równą **limitPosilkow**. Metoda **potomek** zmniejsza **licznikPosilkow** o wartość **kosztPotomka**. Pola **limitPosilkow** i **kosztPotomka** są ustawiane w konstruktorze i nie można ich zmienić. Działanie metod zależy od stanu obiektu: czy jest „żywy” i czy jest „głodny”. Metoda **krokSymulacji** zmniejszy **licznikZycia** tylko wtedy kiedy ten jest większy od zera. Metoda **posilek** zwiększa **licznikPosilkow** tylko wtedy, kiedy ten nie przekroczy wartości **limitPosilkow**. Metoda **potomek** wykona zmniejszenie zmiennej **licznikPosilkow**, jeżeli organizm jest „pączkujący”.

Podsumowując: Klasa **Organizm** zamyka w sobie pewną funkcjonalność, która jest kompletna choć nie opisuje w żaden sposób żadnego żywego organizmu np. glon, grzyb lub bakteria. Mówimy, że zasza tutaj **abstrakcja** fragmentu większej całości.

Listing 3.16. Definicja konstruktora klasy

```
1  Organizm::Organizm(unsigned short dlugoscZycia,
2                        unsigned short limitPosilkow,
3                        unsigned short kosztPotomka):
4      limitPosilkow(limitPosilkow),
5      kosztPotomka(kosztPotomka),
6      licznikZycia(dlugoscZycia),
7      licznikPosilkow(0)
8  {
9
10 }
```

Listing 3.17. Definicja metody posilek

```
1  bool Organizm::posilek()
2  {
3      if(glodny()){
4          licznikPosilkow++;
5          return true;
6      } else return false;
7  }
```

Listing 3.18. Definicja metody potomek

```
1  bool Organizm::potomek()
2  {
3      if(paczkujacy()){
4          licznikPosilkow -= kosztPotomka;
5          return true;
6      } else return false;
7  }
```



Listing 3.19. Definicja metody `krokSymulacji`

```
1 void Organizm::krokSymulacji()
2 {
3     if(zywy()) licznikZycia--;
4 }
```

### Dyskusja – odpowiedź na pytania:

- Co oznacza, że obiekt klasy `organizm` jest „pączkujący”. Znajdź odpowiednie fragmenty w kodzie.
- Listing 3.15, wiersz 4 i 5 → Jaki efekt ma słowo `const` ?. Czy różne obiekty klasy `Organizm` mogą mieć inne wartości pola `kosztPotomka` ? Czy wartość pola `limitPosilkow` może być odczytana w funkcji zewnętrznej np. w `main` ?
- Listing 3.15, wiersz 8, 9 → Dlaczego pola `licznikZycia` oraz `licznikPosilkow` nie mogą być `const` ?
- Listing 3.16, wiersz 16 → Co oznacza słowo `const` dołączone do nazwy metody ?
- Listing 3.16 → Które metody będzie można wywołać na obiekcie klasy `Organizm` ?
- Listing 3.17 → Które metody będzie można wywołać na stałej klasy `Organizm`, tzn. obiekcie zdefiniowanym np. `const Organizma organizm(10,4,2);` ?
- Czy można stworzyć obiekt klasy `Organizm` instrukcją:  
`Organizm organizmPusty;` ?
- Listing 3.16, wiersze 4,5,6,7 → Co tutaj się dzieje? Czy można alternatywnie zastąpić te wiersze podstawieniami w bloku `{ }` ?
- Listing 3.17 → Co robi metoda `posilek` ? Tzn. jaki jest jej algorytm?
- Listing 3.18 → Co robi metoda `potomek` ?
- Listing 3.19 → Co robi metoda `krokSymulacji` ?

### Zadanie 3.4. Test klasy `Organizm`

Klasa `organizm` jest częścią aplikacji *Wirtualny ekosystem*. Zanim zostanie użyta wymaga przetestowania. Wielką zaletą programowania obiektowego jest możliwość oddzielnego testowania elementów aplikacji.

Wykonaj polecenia:

- Poprzednią zawartość pliku `main.cpp` przenieś do innego pliku. (Np. do pliku `zlab03.h` na koniec i wykomentuj cały przeniesiony tekst. W razie potrzeby będzie można do tego kodu wrócić.)
- W pliku `main.cpp` odtwórz szkielet funkcji `main` wg listingu 3.20.

Listing 3.20. Wyjściowa postać pliku `main.cpp`

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     return 0;
7 }
```

- Włącz plik nagłówkowy **organizm.h** do pliku **main.cpp**.
- Dodaj globalną funkcję **drukujOrganizm** do pliku **main.cpp** wg listingu 3.21.s

Listing 3.21. Definicja funkcji globalnej **drukujOrganizm** w pliku **main.cpp**

```
1 void drukujOrganizm(const Organizm & o)
2 {
3     cout << "licznikZycia: "
4         << o.stanLicznikaZycia()
5         << " licznikPosilkow: "
6         << o.stanLicznikaPosilkow() << endl
7         << "limitPosilkow: "
8         << o.limitPosilkow
9         << " kosztPotomka: "
10        << o.kosztPotomka << endl
11        << "głodny: " << o.głodny()
12        << " pączkujący: "
13        << o.paczkujacy() << endl << endl;
14 }
```

- Uzupełnij funkcję **main** wg listingu 3.22, uruchom, obserwuj efekt, eksperymentuj.

Listing 3.22. Definicja metody **main**

```
1 int main()
2 {
3     //1. Test kreacji obiektów
4     Organizm organizm1(8,3,2);
5     Organizm organizm2 = organizm1;
6     Organizm organizm3(organizm1);
7
8     cout << "Wynik testu kreacji obiektów:"
9         << endl << endl;
10    cout << "Organizm1:" << endl;
11    drukujOrganizm(organizm1);
12    cout << "Organizm2" << endl;
13    drukujOrganizm(organizm1);
14    cout << "Organizm3" << endl;
15    drukujOrganizm(organizm2);
16
17    //2. Test niezależności obiektów
18    organizm1.posilek();
19    organizm1.posilek();
20    organizm2.posilek();
21    organizm3.krokSymulacji();
22
23    cout << "Wynik testu niezależności obiektów:"
24        << endl << endl;
25    cout << "Organizm1:" << endl;
26    drukujOrganizm(organizm1);
```



```
27     cout << "Organizm2" << endl;
28     drukujOrganizm(organizm2);
29     cout << "Organizm3" << endl;
30     drukujOrganizm(organizm3);
31
32     //3. Test symulacji
33     Organizm organizm4(8,3,2);
34
35     cout << "Wynik testu symulacji:"
36           << endl << endl;
37     cout << "Stan początkowy:" << endl;
38     drukujOrganizm(organizm4);
39
40     for(int i=1; i<11; i++){
41
42         organizm4.krokSymulacji();
43
44         if(organizm4.paczkujacy()){
45             organizm4.potomek();
46             cout << "---> Potomek" << endl;
47         } else organizm4.posilek();
48
49         cout << "Po wykonaniu kroku symulacji: "
50               << i << endl;
51         drukujOrganizm(organizm4);
52     }
53
54     return 0;
55 }
```

- Po zakończeniu testów zawartość pliku `mian.cpp` przenieś na koniec pliku `organizm.h` i wykomentuj przeniesiony tekst. (Może się jeszcze przyda.)
- W pliku `main.cpp` odtwórz szkielet funkcji `main`.

### Dyskusja – odpowiedz na pytania

- Listing 3.21 → Ile instrukcji języka C++ zostało umieszczonych w wierszach od 3 do 13?
- Listing 3.21, wiersz 1 → Co jest parametrem zdefiniowanej funkcji?
- Listing 3.21 → Które metody klasy `Organizm` nie mogą być użyte wewnątrz funkcji na obiekcie `o`?
- Listing 3.22, wiersz 5 → Jaki konstruktor został użyty do stworzenia obiektu `organizm2`? Przy tworzeniu obiektu jest zawsze uruchamiany jakiś konstruktor.
- Listing 3.22 → Na czym polega test niezależności obiektów?
- Listing 3.22 → Na czym polega test symulacji?



## LABORATORIUM 4. KLASY I OBIEKTY, KONSTRUKTORY I DESTRUKTORY (Cz.2).

### Cel laboratorium:

Celem laboratorium jest nabycie praktyki w zakresie projektowania klas. Efektem prac są elementy aplikacji *Wirtualny Ekosystem*: klasy *UstawieniaSymulacji*, *GeneratorLosowy* oraz 3 typy wyliczeniowe *RodzajMieszkanca*, *AkcjaMieszkanca*, *Polozenie*.

### Zakres tematyczny zajęć:

- Typ wyliczeniowy.
- Składnik statyczny klasy.
- Wzorzec projektowy Singleton.
- Generowanie liczb pseudolosowych.

### Kompendium wiedzy:

- **Typ wyliczeniowy** w języku C++ jest skończonym zbiorem liczb całkowitych np. {0, 2, 8, 100}, którym nadajemy nazwy np. {PUSTY=0, DOMYSLNY=2, ZAPASOWY=8, BLEDU=100}. Całemu zbiorowi również nadajemy nazwę słowem **enum**, tutaj możemy np: `enum Sloty {PUSTY=0, DOMYSLNY=2, ZAPASOWY=8, BLEDU=100}`, gdzie słowo `Sloty` staje się nazwą typu. Od tej pory możemy definiować zmienną tego typu i odnosząc się do jej wartości używać nazw zamiast liczb. Kod staje się wówczas łatwiejszy do czytania. Jeżeli elementom zbioru nie nadamy wprost wartości, wówczas przyjmą wartości domyślne: 0, 1, 2, itd.
- **Składniki statyczne** klasy tworzy się dodając do deklaracji pól lub metod słowo **static**. **Pole statyczne** klasy jest wspólne dla wszystkich obiektów klasy. Jest tylko **jedno** bez względu na liczbę obiektów klasy. Istnieje nawet wtedy, gdy nie ma żadnych obiektów klasy. Wartość tego pola jest taka sama dla wszystkich obiektów. **Metoda statyczna** klasy nie jest związana z żadnym obiektem klasy i można ją wywołać nawet wtedy, kiedy nie ma jeszcze żadnych obiektów tej klasy. Metody statyczne wywołuje się odnosząc się do klasy stosując operator zakresu `NazwaKlasy::nazwaMetodyStatycznej()`.
- **Wzorzec projektowy** w językach obiektowych to specjalny sposób zaprojektowania klasy, tak aby miała jakieś szczególne właściwości. Wzorzec projektowy **Singleton** to sposób zaprojektowania klasy w taki sposób, aby w całym programie mógł istnieć wyłącznie **jeden egzemplarz** obiektu klasy (lub nie istnieć żaden). Szkielet wzorca Singleton przedstawia listing 4.1. W klasie takiej **konstruktor jest metodą prywatną**, co nie pozwala utworzyć obiektu klasy (wiersz 7 i 8). Prywatny konstruktor może być wywołany przez każdą metodę tej samej klasy. Zatem obiekt klasy tworzony jest przez metodę wewnętrzną klasy. Metoda ta musi być statyczna, aby można ją było wywołać wówczas, gdy nie ma jeszcze żadnego obiektu. Metoda ta tworzy obiekt i zwraca do niego referencje. Sam obiekt jest przypisany do pola statycznego (wiersz 13) co gwarantuje, że jest tylko jeden taki obiekt.



Listing 4.1. Przykładowy sposób realizacji wzorca projektowego Singleton

```
1 class NazwaKlasy
2 {
3     public:
4         //Tutaj część publiczna
5
6     private:
7         NazwaKlasy();
8         NazwaKlasy(const NazwaKlasy&);
9
10    public:
11        static NazwaKlasy & pobierzObiekt()
12        {
13            static NazwaKlasy ustawienia;
14            return ustawienia;
15        }
16 };
```

#### Pytania kontrolne:

1. Wy tłumacz czym jest typ wyliczeniowy w języku C++ i podaj przykład zastosowania.
2. Wyjaśnij czym są składniki statyczne klasy.
3. Odpowiedz czym jest wzorzec projektowy klasy. Podaj przykład.

#### Zadanie 4.1. Utworzenie typów wyliczeniowych

Celem zadania jest zdefiniowanie typów wyliczeniowych, które będą używane w aplikacji *Wirtualny Ekosystem*.

Wykonaj polecenia:

- Otwórz projekt z poprzednich zajęć.
- Utwórz plik nagłówkowy o nazwie **ustawienia.h**: Menu Główne → Plik → Nowy plik lub projekt → C++ → Plik nagłówkowy. Nie twórz pliku źródłowego.
- W pliku zdefiniuj 3 typy wyliczeniowe wg listingu 4.2.
- Symbole położenia oznaczają położenia względne od niszy (Rys. 2.):  
P – prawo, G – góra, L – lewo, D – dół, PG – prawo górny, LG – lewy górny, LD – lewy dolny, PD – prawy dolny. Pozostałe nazwy same się tłumaczą, co jest zaletą typu wyliczeniowego.

Listing 4.2. Definicje typów wyliczeniowych

```
1 enum RodzajMieszkanca {GLON, GRZYB, BAKTERIA, PUSTKA, SCIANA,
2                        TRUP, NIEZNANE};
3
4 enum AkcjaMieszkanca {POTOMEK, POLOWANIE, ROZKLAD, NIC};
5
6 enum Polozenie {P=0, PG=1, G=2, LG=3, L=4, LD=5, D=6, PD=7,
7                NIGDZIE=8 };
```



**Zadanie 4.2. Utworzenie klasy UstawieniaSymulacji**

Celem zadania jest utworzenie klasy zgodnej ze wzorcem Singleton, w którym będą przechowywane ustawienia całej symulacji. Dzięki temu zmiana ustawień dokonana przez jeden obiekt będzie od razu dostępna dla innych obiektów.

Wykonaj polecenia:

- Do pliku `ustawienia.h` dodaj klasę wg listingu 4.3 .
- Przetestuj klasę `UstawieniaSymulacji` w pliku `main.cpp` wg listingu 4.4 .
- Po przetestowaniu kod z pliku `main.cpp` zarchiwizuj.

*Listing 4.3. Definicja klasy UstawieniaSymulacji*

```

1  class UstawieniaSymulacji
2  {
3  public:
4
5      const char
6          znakGlon = '*',
7          znakGrzyb = '#',
8          znakBakteria = '@',
9          znakTrup = '+',
10         znakNieokreslony = '?',
11         znakPustaNisza = '-',
12         znakSeparator = ' ';
13
14         unsigned short
15             glonZycieMin = 5,
16             glonZycieMax = 10,
17             glonKosztPotomka = 2,
18             glonLimitPosilkow = 6,
19
20             grzybZycieMin = 40,
21             grzybZycieMax = 60,
22             grzybKosztPotomka = 3,
23             grzybLimitPosilkow = 30,
24
25             bakteriaZycieMin = 25,
26             bakteriaZycieMax = 40,
27             bakteriaKosztPotomka = 3,
28             bakteriaLimitPosilkow = 10;
29
30         bool poprawnyZnakNiszy(char znak) const
31         {
32             return znak == znakGlon ||
33                    znak == znakGrzyb ||
34                    znak == znakBakteria ||
35                    znak == znakTrup ||
36                    znak == znakPustaNisza;
37         }

```



```
38     bool poprawnySeparator(char znak) const
39     {
40         return znak == znakSeparator;
41     }
42
43 private:
44     UstawieniaSymulacji(){}
45     UstawieniaSymulacji(UstawieniaSymulacji&);
46
47 public:
48     static UstawieniaSymulacji & pobierzUstawienia()
49     {
50         static UstawieniaSymulacji ustawienia;
51         return ustawienia;
52     }
53 };
```

Listing 4.4. Kod do testowania klasy UstawieniaSymulacji

```
1     void wyswietl(UstawieniaSymulacji & UST){
2         cout << "Znak glon:" << UST.znakGlon
3             << " zycieMin=" << UST.glonZycieMin
4             << " zycieMax=" << UST.glonZycieMax << endl;
5     }
6
7     int main()
8     {
9         //1. Dostęp do obiektu klasy UstawieniaSymulacji
10        UstawieniaSymulacji & UST1
11            = UstawieniaSymulacji::pobierzUstawienia();
12        UstawieniaSymulacji & UST2
13            = UstawieniaSymulacji::pobierzUstawienia();
14        UstawieniaSymulacji & UST3
15            = UstawieniaSymulacji::pobierzUstawienia();
16
17        cout << "Pobranie ustawien 3x" << endl;
18        cout << "UST1: "; wyswietl(UST1);
19        cout << "UST2: "; wyswietl(UST2);
20        cout << "UST3: "; wyswietl(UST3);
21
22        cout << endl << "Zmiana wartości tylko 1x" << endl;
23        UST2.glonZycieMax = 100;
24        cout << "UST1: "; wyswietl(UST1);
25        cout << "UST2: "; wyswietl(UST2);
26        cout << "UST3: "; wyswietl(UST3);
27        return 0;
28    }
```



### Dyskusja – odpowiedz na pytania:

- Listing 4.2, wiersz 1 → Jakie wartości przyjmują nazwy typu wyliczeniowego?
- Listing 4.3, wiersze od 5 do 12 → Ile instrukcji języka C++ jest zawartych w tych wierszach?
- Listing 4.3, wiersze od 14 do 28 → Ile instrukcji języka C++ jest zawartych w tych wierszach?
- Listing 4.3, wiersze od 32 do 36 → Co oznacza symbol `ll` ?
- Listing 4.3, wiersz od 30 do 37 → Jak działa metoda `poprawnyZnakNiszy` ?
- Listing 4.3, wiersze od 38 do 41 → Jak pozyskiwana jest wartość zwracana przez metodę?
- Listing 4.3, wiersze 44, 44 → Jak nazywają się konstruktory? Dlaczego przy jednym z nich jest `{}` a przy drugim nie? ( i działa? )
- Listing 4.4, wiersze od 1 do 5 → Czy wewnątrz funkcji `wyświetl` można trwale zmienić ustawienia symulacji?
- Listing 4.4 → Na czym polega sprawdzenie, czy rzeczywiście istnieje tylko 1 obiekt klasy `UstawieniaSymulacji` ?

### Zadanie 4.3. Utworzenie klasy `GeneratorLosowy`

Celem zadania jest utworzenie klasy, która będzie zawierała metody do generowania liczb pseudolosowych w zadanych zakresach i określonego typu. Korzystanie z klasy będzie odbywało się bez tworzenia obiektów klasy. Wywoływane będą wyłącznie metody statyczne. Generowanie liczb pseudolosowych omówione było w Zadaniu 2.9.

Wykonaj polecenia:

- Do projektu dodaj klasę `GeneratorLosowy` (tym razem plik `.h` i plik `.cpp`).
- W pliku nagłówkowym zdefiniuj klasę wg listingu 4.5.
- W pliku źródłowym zdefiniuj składnik statyczny i metody klasy wg listingów od 4.6 do 4.9.
- Zmień plik `main.cpp` wg listingu 4.10 i uruchom.
- Po wykonaniu testów plik `main.cpp` zarchiwizuj.

Listing 4.5. Definicja klasy `GeneratorLosowy`

```
1  #include <random>
2
3  class GeneratorLosowy
4  {
5  private:
6      static std::random_device device;
7      GeneratorLosowy(){}
8
9  public:
10     static unsigned short losujPomiedzy
11         (unsigned short min, unsigned short max);
12
13     static long losujPomiedzy(long min, long max);
```



```
14     static int losujOdZeraDo(int max);
15 };
16
17 typedef GeneratorLosowy GEN; //Poza definicją klasy
```

Listing 4.6. Definicja składnika statycznego klasy GeneratorLosowy

```
1  std::random_device GeneratorLosowy::device;
```

Listing 4.7 Definicja metody losujPomiedzy klasy GeneratorLosowy

```
1  unsigned short GeneratorLosowy::losujPomiedzy
2      (unsigned short min, unsigned short max)
3  {
4      if(min>max){
5          unsigned short t = min;
6          min = max;
7          max = t;
8      }
9
10     std::uniform_int_distribution<unsigned short>
11         dist(min, max);
12
13     return dist(GeneratorLosowy::device);
14 }
```

Listing 4.8. Definicja drugiej metody losujPomiedzy klasy GeneratorLosowy

```
1  long GeneratorLosowy::losujPomiedzy(long min, long max)
2  {
3      if(min>max){
4          long t = min;
5          min = max;
6          max = t;
7      }
8
9      std::uniform_int_distribution<long> dist(min, max);
10     return dist(GeneratorLosowy::device);
11 }
```

Listing 4.9. Definicja metody losujOdZeraDo klasy GeneratorLosowy

```
1  int GeneratorLosowy::losujOdZeraDo(int max)
2  {
3      std::uniform_int_distribution<int> dist(0, max);
4      return dist(GeneratorLosowy::device);
5  }
```



Listing 4.10. Kod do testowania klasy GeneratorLosowy

```
1  #include "generatorlosowy.h"
2
3  int main()
4  {
5      cout << "Liczby losowe typu int:" << endl;
6      cout << " od 0 do 5: ";
7      for(int i=0; i<10; i++)
8          cout << GEN::losujOdZeraDo(5) << " ";
9      cout << endl << " od 0 do 3: ";
10     for(int i=0; i<10; i++)
11         cout << GEN::losujOdZeraDo(3) << " ";
12     cout << endl << " od 0 do 20: ";
13     for(int i=0; i<10; i++)
14         cout << GEN::losujOdZeraDo(20) << " ";
15
16     cout << endl << endl;
17
18     cout << "Liczby losowe typu long: " << endl;
19     cout << " od -2 do 2:";
20     for(int i=0; i<10; i++)
21         cout << GEN::losujPomiedzy(-2L, 2L) <<" ";
22
23     cout << endl << endl
24         << "Liczby losowe typu unsigned short: " << endl;
25     cout << " od 2 do 6: ";
26     unsigned short min=2, max=6;
27
28     for(int i=0; i<10; i++)
29         cout << GEN::losujPomiedzy(max, min) <<" ";
30
31     cout << endl << endl;
32
33     return 0;
34 }
```

#### Dyskusja – odpowiedz na pytania:

- Listing 4.5 → W jaki sposób zostało zagwarantowane, że nie zostanie utworzony obiekt klasy **GeneratorLosowy** ?
- Listing 4.5, wiersz 6 → Co należałoby zrobić, aby nie było konieczności użycia **std::** ?
- Listing 4.5, wiersz 17 → Co daje zdefiniowanie synonimu nazwy?
- Listing 4.6 → Dlaczego taka definicja w pliku .cpp jest konieczna?
- Listing 4.7, wiersze od 4 do 8 → Po co jest ten fragment kodu?
- Listing 4.9 → Dlaczego w metodzie **losujOdZeraDo(int max)** nie można po prostu wywołać metody **losujPomiedzy(0,max)** ?



## LABORATORIUM 5. KLASY I OBIEKTY, KONSTRUKTORY I DESTRUKTORY (Cz.3). REFERENCJE I WSKAŹNIKI (Cz.2).

### Cel laboratorium:

Celem laboratorium jest stworzenie klasy **Sasiedztwo**, która jest elementem aplikacji *Wirtualny Ekosystem*.

### Zakres tematyczny zajęć:

- Referencje i wskaźniki jako sposoby przekazywania argumentów do metod.
- Wskaźnik jako typ zwracany przez metodę.
- Referencja jako typ zwracany przez metodę.
- Polimorfizm statyczny.

### Kompedium wiedzy:

- W języku C++ istnieją 3 możliwości przekazania argumentu do metody lub funkcji: przez **kopię**, przez **wskaźnik \***, przez **referencję &**.
- Argument przekazany przez kopię może być modyfikowany wewnątrz metody, ale zmienna oryginalna nie ulega modyfikacji, bo jej nie ma wewnątrz metody tylko jej kopia, która nie jest powiązana z oryginałem
- Argument przekazany przez wskaźnik, to tak naprawdę również przekazanie kopii, ale tym razem wskaźnika do jakiegoś obiektu (w szerokim rozumieniu słowa obiekt), a nie kopii obiektu. Kopię wskaźnika również możemy modyfikować wewnątrz i na zewnątrz metody oryginał wskaźnika się nie zmienia, ale teraz sytuacja jest inna: oryginał wskaźnika (na zewnątrz metody) i jego kopia (wewnątrz metody) pokazują na ten sam obiekt na zewnątrz metody. To ten obiekt modyfikujemy, a nie wskaźniki do niego. Nie ma znaczenia, za pomocą którego wskaźnika docieram do obiektu. W skrócie mówimy, że „argument przekazany przez wskaźnik może być wewnątrz metody modyfikowany”. Inna sytuacja zachodzi kiedy przekazujemy wskaźnik do obiektu stałego, czyli takiego którego modyfikować nie wolno. Wówczas próba modyfikacji obiektu, na który pokazuje ten wskaźnik, nie powiedzie się. Przy czym wskaźnikiem do obiektu stałego można wskazywać dowolny obiekt niekoniecznie stałą. Wskaźnik do obiektu stałego nie gwarantuje, że obiekt jest stały, tylko, że za pomocą tego wskaźnika nie zmodyfikujemy go. Istnieją też wskaźniki stałe (nie „do stałej”), których wartości nie można zmieniać (bo są po prostu stałymi). Taki wskaźnik jest na zawsze ustawiony na konkretny obiekt. Ten konkretny obiekt nie musi być stały.
- Referencje można rozumieć jako wskaźniki stałe. Mogą być one do stałej, ale nie muszą. Przekazane jako argument funkcji zachowują się tak samo jak wskaźniki.
- Analogicznie metoda lub funkcja może jako swoją wartość zwracać wskaźnik lub referencje. Tutaj znowu metoda zwraca kopię czegoś co ma w środku. Ale ta kopia pokazuje na jakiś obiekt i to ten obiekt jest dla nas produktem metody.
- **Polimorfizm statyczny** zachodzi wówczas kiedy klasa posiada dwie metody o tej samej nazwie. Metody mogą mieć taką samą nazwę tylko wtedy, kiedy te metody różnią się zestawem argumentów. Dzięki temu możliwe jest ustalenie na etapie budowania która metoda jest tak naprawdę wywoływana.



### Pytania kontrolne:

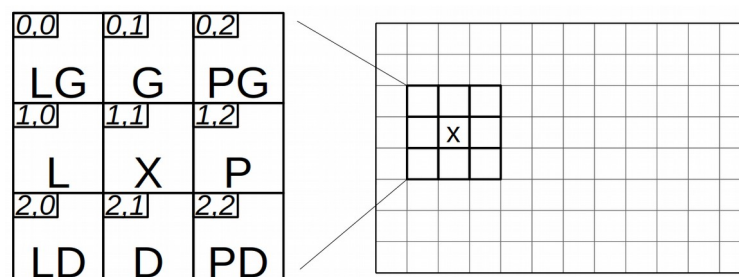
1. Wyłutnacj jest różnica pomiędzy wskaźnikiem do stałej, a stałym wskaźnikiem. Czy jest możliwe, alby wskaźnik był jednocześnie stały i do stałej?
2. Odpowiedz w jakim celu przekazujemy do wnętrza metody wskaźnik (lub referencje) do obiektu stałego, skoro wewnątrz metody nie można tego obiektu zmodyfikować. Czy nie wystarczyłaby kopia tego obiektu?
3. Wyjaśnij co to jest polimorfizm statyczny. Użyj klasy **GeneratorLosowy** z poprzednich zajęć jako ilustracji.

### Zadanie 5.1. Utworzenie klasy **Sasiedztwo**

Przed przystąpieniem do czytania należy przypomnieć sobie typy wyliczeniowe **RodzajMieszkanca** oraz **Polozenie** (Zadanie 4.1.).

Klasa **Sasiedztwo** będzie używana do przekazania informacji co znajduje się w najbliższym sąsiedztwie wskazanej niszy (Rys. 5.1 oraz Rys. 2). Wskazanie niszy odbywa się poprzez podanie jej położenia, czyli numeru wiersza i numeru kolumny. Mając wskazaną niszę możemy określić jej sąsiedztwo, czyli stworzyć obiekt klasy **Sasiedztwo** i wypełnić go odpowiednimi wartościami. (Te czynności będzie wykonywać już inna klasa.) Klasa **Sasiedztwo** posiada prywatne pole **sasiad**, które jest tablicą kwadratową o wymiarach 3x3 (patrz lewa strona rys. 5.1.). Środkowy element tej tablicy, nie jest wykorzystywany, gdyż nie oznacza sąsiada tylko wskazaną nisze. Tablica **sasiad** jako tablica posiada indeksowane komórki: [0][0], [0][1], itd. , jednak w programie wygodniej będzie posługiwać się nazwami literowymi: LG – lewy górny, G – górny, itd. Konieczne jest zatem przeliczanie położenia „literowego” na indeksy numeryczne. Przeliczanie odbywa się na różnych poziomach. Wewnątrz klasy metoda **elementWewnetrzny** zwraca wskaźnik od odpowiedniej komórki w tablicy **sasiad** na podstawie położenia literowego. Dwie metody o nazwie **zmienIdeksyWgPolozenia** zmieniają indeksy komórki w tablicy globalnej (patrz prawa strona rys. 5.1.) na indeksy jej sąsiada określonego położeniem literowym.

Metoda **losujPolozenie** zwraca losową wartość położenia w zapisie literowym jedną z: P, PG, G, LG, L, LD, D, PD. Natomiast metoda **losujSasiada** również zwraca położenie literowe, ale losuje tylko spośród sąsiadów wskazanego typu np. GLON. Metoda **okreslSasiada** umieszcza informacje o rodzaju sąsiada we wskazanym położeniu, tzn. wypełnia klasę sąsiedztwo danymi (patrz listing 5.12, wiersze od 5 do 12). Natomiast metoda **ktoJestSasiadem** zwraca informacje jaki rodzaj sąsiada jest w danym miejscu. Metoda **ile** zwraca liczbę sąsiadów wskazanego rodzaju.



Rys 5.1. Sąsiedztwo wskazanej niszy X w postaci tablicy 3x3. Pokazano indeksy liczbowe tej tablicy oraz indeksy opisowe (P-prawy, PG-prawy góra, ...)



Wykonaj polecenia:

- Otwórz projekt z poprzednich zajęć.
- Utwórz nową klasę **Sasiedztwo**.
- Zdefiniuj klasę w pliku nagłówkowym **sasiedztwo.h** (Listing 5.1)
- Zdefiniuj metody klasy w pliku źródłowym **sasiedztwo.cpp** (Listingi od 5.2 do 5.10)
- Przygotuj plik **main.cpp** wg listingów 5.11 i 5.12 ;
- Uruchom, zaobserwuj działanie, zarchiwizuj plik **main.cpp**.

Listing 5.1. Definicja klasy Sasiedztwo

```
1  class Sasiedztwo{
2
3  private:
4      RodzajMieszkanca sasiad[3][3];
5
6      RodzajMieszkanca * elementWewnetrzny(
7          Polozenie polozenie);
8
9      static Polozenie losujPolozenie();
10
11 public:
12     Sasiedztwo(RodzajMieszkanca rodzaj = NIEZNANE);
13
14     void okreslSasiada(Polozenie polozenie,
15         RodzajMieszkanca rodzaj);
16
17
18     RodzajMieszkanca ktoJestSasiadem(Polozenie polozenie);
19
20     unsigned short ile(RodzajMieszkanca rodzaj) const;
21
22     Polozenie losujSasiada(RodzajMieszkanca rodzaj);
23
24     static void zmienIdeksyWgPolozenia(
25         Polozenie polozenie, long & wiersz, long & kolumna);
26
27     static void zmienIdeksyWgPolozenia(
28         Polozenie polozenie, unsigned int & wiersz,
29         unsigned int & kolumna);
30
31 };
```

Listing 5.2. Metoda elementWewnetrzny klasy Sasiedztwo

```
1  RodzajMieszkanca * Sasiedztwo::
2  elementWewnetrzny(Polozenie polozenie)
3  {
4      switch (polozenie) {
5          case P: return &sasiad[1][2];
```



```
6     case PG: return &sasiad[0][2];
7     case G: return &sasiad[0][1];
8     case LG: return &sasiad[0][0];
9     case L: return &sasiad[1][0];
10    case LD: return &sasiad[2][0];
11    case D: return &sasiad[2][1];
12    case PD: return &sasiad[2][2];
13    case NIGDZIE: return nullptr;
14    }
15    return nullptr;
16 }
```

Listing 5.3. Metoda `losujPolozenie` klasy `Sasiedztwo`

```
1  Polozenie Sasiedztwo::losujPolozenie()
2  {
3      unsigned short min = 0, max=8;
4      return static_cast<Polozenie>
5          (GeneratorLosowy::losujPomiedzy(min,max));
6  }
```

Listing 5.4. Konstruktor klasy `Sasiedztwo`

```
1  Sasiedztwo::Sasiedztwo(RodzajMieszkanca rodzaj)
2  {
3      for(auto i : {0,1,2})
4          for(auto j: {0,1,2})
5              sasiad[i][j] = rodzaj;
6
7      sasiad[1][1] = NIEZNANE;
8  }
```

Listing 5.5. Metoda `okreslSasiada` klasy `Sasiedztwo`

```
1  void Sasiedztwo::okreslSasiada
2  (Polozenie polozenie, RodzajMieszkanca rodzaj)
3  {
4      if(polozenie != NIGDZIE)
5          *elementWewnetrzny(polozenie) = rodzaj;
6  }
```

Listing 5.6. Metoda `ktoJestSasiadem` klasy `Sasiedztwo`

```
1  RodzajMieszkanca Sasiedztwo::
2  ktoJestSasiadem(Polozenie polozenie)
3  {
4      if(polozenie != NIGDZIE)
5          return * elementWewnetrzny(polozenie);
6      else return NIEZNANE;
7  }
```



Listing 5.7. Metoda ile klasy Sasiedztwo

```
1 unsigned short Sasiedztwo::
2 ile(RodzajMieszkanca rodzaj) const
3 {
4     unsigned short licznik = 0;
5
6     for(int i : {0,1,2})
7         for(int j: {0,1,2}){
8             if(i==1 && j==1) continue;
9             if(sasiad[i][j]==rodzaj) licznik++;
10        }
11
12    return licznik;
13 }
```

Listing 5.8. Metoda losujSasiada klasy Sasiedztwo

```
1 Polozenie Sasiedztwo::
2 losujSasiada(RodzajMieszkanca rodzaj)
3 {
4     if(ile(rodzaj)==0) return NIGDZIE;
5     else{
6         Polozenie polozenie = Sasiedztwo::losujPolozenie();
7
8         while(ktoJestSasiadem(polozenie) != rodzaj)
9             polozenie = Sasiedztwo::losujPolozenie();
10
11         return polozenie;
12     }
13 }
```

Listing 5.9. Metoda zmienIdeksyWgPolozenia klasy Sasiedztwo

```
1 void Sasiedztwo::
2 zmienIdeksyWgPolozenia(Polozenie polozenie,
3                          long &wiersz, long &kolumna)
4 {
5     if(polozenie==PG || polozenie==P || polozenie==PD)
6         kolumna++;
7     else if(polozenie==LG || polozenie==L | polozenie==LD)
8         kolumna--;
9
10    if(polozenie==LG || polozenie==G || polozenie==PG)
11        wiersz--;
12    else if(polozenie==LD || polozenie==D ||polozenie==PD)
13        wiersz++;
14 }
```



Listing 5.10. Druga metoda zmienIdeksyWgPolozenia klasy Sasiedztwo

```
1 void Sasiedztwo::
2 zmienIdeksyWgPolozenia(Polozenie polozenie,
3 unsigned int &wiersz, unsigned int &kolumna)
4 {
5     long w = static_cast<long>(wiersz);
6     long k = static_cast<long>(kolumna);
7
8     zmienIdeksyWgPolozenia(polozenie,w,k);
9
10    wiersz = static_cast<unsigned int>(w);
11    kolumna = static_cast<unsigned int>(k);
12 }
```

Listing 5.11. Funkcja globalna w pliku main.cpp

```
1 string nazwaRodzaju(RodzajMieszkanca rodzaj){
2     switch (rodzaj) {
3         case GLON: return "GLON";
4         case GRZYB: return "GRZYB";
5         case BAKTERIA: return "BAKTERIA";
6         case PUSTKA: return "PUSTKA";
7         case SCIANA: return "ŚCIANA";
8         case TRUP: return "TRUP";
9         case NIEZNANE: return "NIEZNANE";
10    }
11 }
```

Listing 5.12. Kod metody main

```
1 int main()
2 {
3     Sasiedztwo sasiedztwo;
4
5     sasiedztwo.okreslSasiada(P, GLON);
6     sasiedztwo.okreslSasiada(PG, GRZYB);
7     sasiedztwo.okreslSasiada(G, GRZYB);
8     sasiedztwo.okreslSasiada(LG, GLON);
9     sasiedztwo.okreslSasiada(L, BAKTERIA);
10    sasiedztwo.okreslSasiada(LD, BAKTERIA);
11    sasiedztwo.okreslSasiada(D, GLON);
12    sasiedztwo.okreslSasiada(PD, PUSTKA);
13
14    cout << "Przegląd sąsiedztwa:" << endl;
15
16    for(int i=0; i<8; i++){
17
18        Polozenie p = static_cast<Polozenie>(i);
19    }
```



```
20         RodzajMieszkanca
21             r = sasiedztwo.ktoJestSasiadem(p);
22
23         cout << "polozenie=" << p << " rodzaj="
24             << nazwaRodzaju(r) << endl;
25     }
26
27     cout << endl << "Policzenie sasiadów"
28         << "określonego rodzaju:" << endl
29         << " glony=" << sasiedztwo.ile(GLON) << endl
30         << " grzyby=" << sasiedztwo.ile(GRZYB) << endl
31         << " trupy=" << sasiedztwo.ile(TRUP) << endl;
32
33     cout << endl << "Wylosowanie sasiada:" << endl
34         << " glon -> "
35         << sasiedztwo.losujSasiada(GLON) << endl
36         << " pustka -> "
37         << sasiedztwo.losujSasiada(PUSTKA) << endl
38         << " trup -> "
39         << sasiedztwo.losujSasiada(TRUP) << endl;
40
41
42     long wiersz, kolumna;
43     cout << endl
44         << "Zmiana indeksów [5][7]"
45         << "wg polozenia:" << endl;
46
47     for(int i=0; i<8; i++){
48
49         Polozenie p = static_cast<Polozenie>(i);
50         wiersz = 5; kolumna = 7;
51
52         Sasiedztwo::
53             zmienIdeksyWgPolozenia(p,wiersz,kolumna);
54
55         cout << " położenie: " << p << " ->[" << wiersz
56             << "][" << kolumna << "]" << endl;
57     }
58
59     cout << endl;
60
61     return 0;
62 }
```





**Dyskusja – odpowiedz na pytania:**

- Listing 5.1 → Gdzie znajdują się definicje typów **RodzajMieszkanca** i **Polozenie** ?
- Listing 5.1, wiersze od 28 do 33 → W jaki sposób są przekazywane argumenty metody?
- Listing 5.2, wiersz 15 → Co oznacza **nullptr** ?
- Listing 5.3, wiersz 4 i 5 → Co tu się dzieje?
- Listing 5.4 → Co oznacza słowo **auto**?
- Listing 5.5, linia 5 → Dlaczego jest użyty znak **\*** ?
- Listing 5.6 → Co zwraca metoda i w jaki sposób ?
- Listing 5.7 → Opisz działanie obu pętli **for**.
- Listing 5.8, wiersze 8 i 9 → Czy zachodzi ryzyko, że pętla **while** się nie zakończy?
- Listing 5.9 → Jaki jest algorytm metody?
- Listing 5.10 → Dlaczego zachodzi konieczność rzutowania?
- Listing 5.9 oraz Listing 5.10 → Czy jest to polimorfizm?
- Listing 5.12, wiersz 49 → Na czym polega rzutowanie?
- Listing 5.12, wiersze 52 i 53 → Która z 2 metod została wywołana?



## **LABORATORIUM 6. POLIMORFIZM, FUNKCJE WIRTUALNE, KLASY ABSTRAKCYJNE.**

### **Cel laboratorium:**

Pierwszym celem laboratorium jest przedstawienie polimorfizmu dynamicznego jako naturalnej potrzeby w przypadku programowania obiektowego oraz dziedziczenia wielobazowego. W tym celu zostaną stworzone klasy **Kwadrat** oraz **Obliczenia**. Klasy te nie jest częścią aplikacji *Wirtualny Ekosystem*. Drugim celem laboratorium jest stworzenie klas: **ZamiarMieszkanca** i **Mieszkaniec**, które są częścią tej aplikacji.

### **Zakres tematyczny zajęć:**

- Dziedziczenie.
- Dostęp do składników klas bazowych.
- Dziedziczenie wielokrotne.
- Metody i klasy wirtualne.
- Polimorfizm dynamiczny.

### **Kompendium wiedzy:**

- Dziedziczenie w języku C++ dotyczy klas. Klasa, która dziedziczy (**potomek / klasa potomna**) otrzymuje wszystkie składniki klasy po której dziedziczy (**przodek / klasa bazowa**). Zwykle klasa potomna jest rozbudowywana, tj. korzysta z tego co już jest i dodaje swoje elementy. Nie ma konieczności przepisywania kodu z klasy bazowej do klasy pochodnej, ponieważ on już tam jest.
- Nie wszystko jest dziedziczone. Nie są dziedziczone konstruktory. Nie są również dziedziczone destruktory.
- Klasa potomna ma dostęp do wszystkich składników publicznych klasy bazowej.
- Klasa potomna nie ma dostępu do składników prywatnych klasy bazowej.
- Istnieje jeszcze rodzaj składników **protected** (obok public i private). Składniki protected są dostępne dla klas potomnych, ale nie są dostępne dla składników spoza klasy.
- Rodzaj składników klasy bazowej (public, protected, private) jest tylko jednym z dwóch czynników określających widoczność składników klasy bazowej w klasie pochodnej. Drugim czynnikiem jest sposób dziedziczenia. Klasa potomna może dziedziczyć na trzy sposoby: public, protected i private (tzw. **operatory widoczności**). Zbieżność nazw może być myląca. Dziedziczenie „public” powoduje, że odziedziczone składniki public są nadal public, a protected są nadal protected. Dziedziczenie „protected” powoduje, że odziedziczone składniki public i protected stają się protected. Dziedziczenie „private” powoduje, że odziedziczone składniki public i protected stają się private. Natomiast składniki, które są private w klasie bazowej są dziedziczone, ale nie ma do nich bezpośredniego dostępu z klasy pochodnej. Są jak gdyby wewnętrznymi mechanizmami, których w klasie pochodnej już nie można zmienić.
- W języku C++ jest możliwe **dziedziczenie wielobazowe**. Dziedziczenie wielobazowe zachodzi wówczas gdy klasa pochodna posiada więcej niż jedną klasę bazową.

- **Metoda wirtualna** jest specjalną metodą, taką która występuje zarówno w klasie bazowej jak i pochodnej. W obu klasach ma taką samą nazwę i taką samą listę argumentów i jest poprzedzona słowem **virtual**.
- Metoda wirtualna nie musi być zdefiniowana w klasie bazowej. Nie jest to konieczne, jeżeli nie będą nigdy tworzone obiekty klasy bazowej. Wówczas mówimy, że ta metoda w klasie bazowej jest **abstrakcyjna**. Sama klasa bazowa wtedy jest również abstrakcyjna. Nie można utworzyć obiektów klasy abstrakcyjnej.
- **Polimorfizm dynamiczny** dopełnia mechanizm dziedziczenia i dotyczy metod wirtualnych. Zakładając, że klasa bazowa i pochodna mają zdefiniowaną taką samą metodę wirtualną, można zadeklarować wskaźnik do klasy bazowej, wskaźnikiem tym pokazać na klasę pochodną i wywołać tę metodę wirtualną. Wówczas pomimo, że wskaźnik jest do klasy bazowej, to wywołana zostaje metoda klasy pochodnej. Oczywiście polimorfizm ma wówczas sens, gdy klasa bazowa i pochodna przedmiotową metodę realizują w inny sposób.

### Pytania kontrolne:

1. Opisz czym jest dziedziczenie.
2. Odpowiedz jakie składniki się nie dziedziczą, a jakie się dziedziczą.
3. Przedstaw jak sterować widocznością składników przy dziedziczeniu.
4. Wyjaśnij jak utworzyć metodę wirtualną.
5. Opisz na czym polega polimorfizm dynamiczny.

### Zadanie 6.1. Uzyskanie polimorfizmu dynamicznego

W zadaniu zostanie utworzona klasa **Kwadrat** mająca klasę bazową **Prostokat** z zadania 3.1.

Wykonaj polecenia:

- Otwórz projekt z poprzednich zajęć.
- Utwórz nową klasę **ZLab06**, upewnij się czy pojawiły się 2 pliki ( **.h** i **.cpp**).
- Usuń z pliku nagłówkowego deklarację klasy **ZLab06**, a z pliku źródłowego definicję konstruktora tej klasy.
- W pliku nagłówkowym zdefiniuj klasę **Kwadrat**, która jest potomkiem klasy **Prostokat** wg listingu 6.1.

*Listing 6.1. Klasa Kwadrat*

```
1 class Kwadrat: public Prostokat
2 {
3
4 };
```

- W funkcji **main.cpp** utwórz obiekt klasy **Kwadrat** i go przetestuj jak na listingu 6.2. Usuń znak komentarza z wiersza 6 i zaobserwuj efekt.



Listing 6.2. Pierwszy test – kod funkcji main

```
1  int main()
2  {
3      Kwadrat k1;
4      cout << k1.doTekstu() << endl;
5
6      //Kwadrat k2("Kwadrat",3,3);
7
8      return 0;
9  }
```

- Otwórz plik `zlab03.h` i zmień słowo `private` na `protected` w definicji klasy `Prostokat`.
- Dodaj konstruktor i destruktor klasy `Kwadrat` wg listingu 6.3.

Listing 6.3. Uzupełnienie definicji klasy `Kwadrat`

```
1  class Kwadrat: public Prostokat
2  {
3  public:
4
5      Kwadrat(string nazwa = "?", double bok =0)
6          :Prostokat(nazwa,bok,bok){}
7
8      ~Kwadrat()
9      {
10         cout << "Kwadrat: " << nazwa
11             <<" kończy działanie" << endl;
12     }
13 };
```

- Zmień kod funkcji `main.cpp` wg listingu 6.4. Uruchom i zaobserwuj efekt.

Listing 6.4. Drugi test – kod funkcji main

```
1  int main()
2  {
3      Kwadrat k1;
4      Kwadrat k2("Drugi kwadrat",4);
5      cout << k1.doTekstu() << endl << endl;
6      cout << k2.doTekstu() << endl << endl;
7      return 0;
8  }
```

- Łatwo zauważyć, że działanie metody `doTekstu` nie jest satysfakcjonujące. `Kwadrat` przedstawia się jako prostokąt.
- Wstaw do klasy `Kwadrat` deklaracje metod `string doTekstu()`.
- W pliku `zlab06.cpp` umieść definicję tej metody wg listingu 6.5.

Listing 6.5. Definicja metody `doTekstu` klasy `Kwadrat`

```

1  string Kwadrat::doTekstu()
2  {
3      string napis = "";
4      napis = "Kwadrat o nazwie: " + nazwa
5              + " bok=" + to_string(bok1);
6      if(poprawny) napis += " obwod=" + to_string(obwod)
7                  + " pole=" + to_string(pole);
8      else napis += " !Figura niepoprawna.";
9      return napis;
10 }
```

- Uruchom program zaobserwuj działanie. Czy teraz kwadrat przedstawia się jako kwadrat?
- Zmień kod funkcji `main.cpp` wg listingu 6.6. Uruchom i zaobserwuj efekt.

Listing 6.6. Trzeci test – kod funkcji `main`

```

1  int main()
2  {
3      Kwadrat kwadrat("Kwadrek",4);
4      Prostokat prostokat("Prostak",2,4);
5
6      Prostokat * wskaznikDoProstokat;
7
8      wskaznikDoProstokat = &prostokat;
9      cout << wskaznikDoProstokat->doTekstu()
10         << endl << endl;
11
12     wskaznikDoProstokat = &kwadrat;
13     cout << wskaznikDoProstokat->doTekstu()
14         << endl << endl;
15
16     return 0;
17 }
```

- Jest możliwe wskazanie na obiekt klasy potomnej wskaźnikiem do pokazywania na obiekcie klasy bazowej (Listing 6.6, wiersz 12), ale wtedy uruchamiają się metody klasy bazowej, bo nie ma polimorfizmu dynamicznego.
- Otwórz plik `zlab03.h` i dodaj słowo `virtual` do deklaracji metody `doTekstu`: `virtual std::string doTekstu()`.
- Uruchom program i zaobserwuj działanie. Teraz uruchamia się metod `doTekstu` klasy `Kwadrat` pomimo tego, że używany jest wskaźnik do klasy `Prostokat`. **To jest polimorfizm dynamiczny.**
- Wróć do klasy `Prostokat` i dodaj słowo `virtual` do deklaracji destruktora.
- !UWAGA. Jeżeli naszą intencją jest, aby każda klasa potomna po klasie `Kwadrat` mogła skorzystać z polimorfizmu, w klasie `Kwadrat` należy również dodać słowo `virtual` do deklaracji odpowiednich metod.

- Zmień funkcję `main` wg listingu 6.7. Uruchom. Zaobserwuj działanie.

Listing 6.7. Czwarty test – kod funkcji `main`

```
1  int main()
2  {
3
4      Kwadrat kwadrat("Kwadrek",4);
5      Prostokat prostokat("Prostak",2,4);
6
7      Prostokat & referencjaDoProstokat1 = prostokat;
8
9      cout << referencjaDoProstokat1.doTekstu()
10         << endl << endl;
11
12     Prostokat & referencjaDoProstokat2 = kwadrat;
13
14     cout << referencjaDoProstokat2.doTekstu()
15         << endl << endl;
16
17     return 0;
18 }
```

- Odpowiedz czy polimorfizm dynamiczny działa „po referencji”.

## Zadanie 6.2. Klasy wirtualne i dziedziczenie wielokrotne

W zadaniu zostanie stworzona klasa abstrakcyjna `Obliczenia`, którą dodatkowo odziedziczy klasa `Kwadrat`. Klasa `Obliczenia` będzie miała dwie metody abstrakcyjne: `promienKolaWgPola` oraz `promienOkreguWgObwodu`. Każda klasa, która odziedziczy po klasie `Obliczenia` będzie musiała te metody zdefiniować (zaimplementować). Metoda `promienKolaWgPola` zwraca promień koła o takim samym polu jak dana figura (tutaj `Kwadrat`), natomiast `promienKolaWgOkregu` zwraca promień koła o takim samym obwodzie jak dana figura (tutaj `Kwadrat`).

Wykonaj polecenia:

- Dodaj definicję klasy `Obliczenia` w pliku `zlab06.h` powyżej klasy `Kwadrat` wg listingu 6.8.

Listing 6.8. Definicja klasy `Obliczenia`

```
1  class Obliczenia {
2  protected:
3      const double pi = 3.14;
4  public:
5      virtual double promienKolaWgPola() = 0;
6      virtual double promienOkreguWgObwodu() = 0;
7      virtual ~Obliczenia();
8  };
```



- W deklaracji klasy `Kwadrat` dodaj kolejną klasę bazową:  
`class Kwadrat: public Prostokat, public Obliczenia`
- W pliku `zlab06.cpp` dodaj definicje trzech metod wg listingu 6.9.

*Listing 6.9. Uzupełnienie pliku `zlab06.cpp`*

```
1  Obliczenia::~Obliczenia()
2  {
3
4  }
5
6  #include <cmath>
7  double Kwadrat::promienKolaWgPola()
8  {
9      return sqrt(pole / pi);
10 }
11
12 double Kwadrat::promenOkreguWgObwodu()
13 {
14     return obwod / (2*pi);
15 }
```

- Zmień funkcję `main` tak, aby można było wypróbować nową funkcjonalność obiektów klasy `Kwadrat` (Listing 6.10)

*Listing 6.10. Piąty test – kod funkcji `main`*

```
1  int main()
2  {
3      Kwadrat kwadrat("Kwadrek",4);
4
5      Kwadrat & refDoKwadrat = kwadrat;
6
7      double r0 = refDoKwadrat.promienKolaWgPola();
8      double r1 = refDoKwadrat.promenOkreguWgObwodu();
9
10     cout << "r0=" << r0 << endl
11          << "r1=" << r1 << endl;
12
13     return 0;
14 }
```

### **Zadanie 6.3. Klasa `Mieszkaniec`**

W zadaniu zostaną stworzone dwie klasy `ZamiarMieszkanca` oraz `Mieszkaniec`. Obiekty klasy `ZamiarMieszkanca` będą służyły do przekazania informacji o decyzji wirtualnych organizmów (glon, grzyb, bakteria) będącej reakcją na otaczające ich sąsiedztwo. Natomiast klasa `Mieszkaniec` będzie klasą abstrakcyjną, która zawiera niezbędne metody do komunikacji pomiędzy organizmem i wirtualnym miejscem bytowania tego organizmu, tj. niszą.



Wykonaj polecenia:

- Dodaj do projektu nową klasę o nazwie **Mieszkaniec**.
- Otwórz plik **mieszkaniec.h** i dodaj definicje klasy **ZamiarMieszkanca** powyżej klasy **Mieszkaniec** wg listingu 6.11.

*Listing 6.11. Definicja klasy ZamiarMieszkanca*

```
1  class ZamiarMieszkanca{
2  public:
3      const AkcjaMieszkanca akcja;
4      const Polozenie gdzie;
5
6      ZamiarMieszkanca(AkcjaMieszkanca _akcja = NIC,
7                      Polozenie _gdzie = NIGDZIE)
8          :akcja(_akcja),gdzie(_gdzie){}
9  };
```

- Dodaj do pliku **mieszkaniec.h** deklaracje wyprzedzającą klasy **Sasiedztwo** oraz definicję klasy **Mieszkaniec** wg listingu 6.12.

*Listing 6.12. Deklaracja wyprzedzająca klasy Sasiedztwo oraz definicja klasy Mieszkaniec*

```
1  class Sasiedztwo;
2
3  class Mieszkaniec
4  {
5  protected:
6      char symbol;
7
8  public:
9      Mieszkaniec(char _symbol =
10                  UstawieniaSymulacji::
11                  pobierzUstawienia().znakNieokreslony);
12
13      Mieszkaniec(const Mieszkaniec& mieszkaniiec);
14
15      virtual char jakiSymbol() const final;
16
17      virtual ~Mieszkaniec();
18
19      virtual RodzajMieszkanca kimJestes() = 0;
20
21      virtual ZamiarMieszkanca
22          wybierzAkcje(Sasiedztwo sasiedztwo) =0;
23
24      virtual Mieszkaniec * dajPotomka() = 0;
25
26      virtual void
27          przyjmijZdobycz(Mieszkaniec * mieszkaniiec) = 0;
28  };
```

- Otwórz plik `mieszkaniec.cpp` i dodaj definicje niewirtualnych metod klasy `Mieszkaniec`

Listing 6.13. Definicje niewirtualnych metod klasy `Mieszkaniec`

```
1  Mieszkaniec::Mieszkaniec(char _symbol)
2  {
3      symbol = _symbol;
4  }
5
6  Mieszkaniec::Mieszkaniec(const Mieszkaniec &mieszkaniec)
7  {
8      symbol = mieszkaniac.symbol;
9  }
10
11 char Mieszkaniec::jakiSymbol() const
12 {
13     return symbol;
14 }
15
16 Mieszkaniec::~~Mieszkaniec()
17 {
18
19 }
```

#### Dyskusja – odpowiedz na pytania:

- Listing 6.2, wiersz 4 → Skąd się wzięła metoda `doTekstu` w klasie `Kwadrat`?
- Listing 6.2, wiersz 6 → Dlaczego to nie zadziała?
- Listing 6.3, wiersze 5 i 6 → Jak działa konstruktor klasy `Kwadrat`?
- Listing 6.6, wiersz 12 → Po co jest znak `&` ? I jak on działa w tym przypadku?
- Listing 6.7, wiersz 12 → Czy można byłoby użyć tutaj pierwszej referencji?
- Listing 6.8, wiersze 5 i 6 → Co oznacza „=0” ?
- Listing 6.10 → Czy zamiast referencji można użyć wskaźnika?
- Czy można utworzyć obiekt klasy `Mieszkaniec`?
- Czy można utworzyć obiekt klasy `ZamiarMieszkanca`?
- Listing 6.11, wiersze 6 i 7 → Co oznaczają fragmenty „= NIC” oraz „= NIGDZIE”?
- Listing 6.12, wiersz 1 → czy fragment `class Sasiedztwo` można zastąpić `#include „sasiedztwo.h”` ?
- Listing 6.12, wiersz 13 → Jak nazywa się ta metoda?
- Listing 6.15, wiersz 15 → Co powoduje słowo `final` ?
- Listing 6.12 → Które metody są abstrakcyjne ?



## LABORATORIUM 7. DZIEDZICZENIE I DOSTĘP DO ELEMENTÓW KLAS.

### Cel laboratorium:

Celem laboratorium jest stworzenie klas będących częścią aplikacji *Wirtualny Ekosystem* oraz przetestowanie tych klas. Są to klasy: **Glon**, **Grzyb**, **Bakteria** oraz **Nisza**. Pierwsze trzy klasy są zaprojektowane z wykorzystaniem dziedziczenia i polimorfizmu.

### Zakres tematyczny zajęć:

- Praktyka projektowania opartego na dziedziczeniu i polimorfizmie.
- Zastosowanie dziedziczenia wielobazowego.
- Zastosowanie polimorfizmu.
- Przeciążenie operatora podstawiania.

### Kompendium wiedzy:

- Przeciążanie (przeładowywanie) operatorów jest to definiowanie operatorów do własnych typów.
- Przeciążanie może być zrealizowane jako metoda składowa klasy albo funkcja globalna.
- Wyczerpujące informacje na temat przeciążania operatorów dostępne są pod adresem: <https://en.cppreference.com/w/cpp/language/operators>

### Pytania kontrolne:

- 1 . Przedstaw sposób przeciążenia operatora przypisania. (Patrz listing 7.12. oraz 7.14)

### Zadanie 7.1. Stworzenie klas **Glon**, **Grzyb** oraz **Bakteria**

Klasy **Glon**, **Grzyb** oraz **Bakteria** są potomkami klasy konkretnej **Organizm** oraz klasy abstrakcyjnej **Mieszkaniec**. Trzy nowe klasy nie definiują własnych pól i metod. Natomiast implementują metody wirtualne.

Wykonaj polecenia:

- Otwórz projekt z poprzednich zajęć.
- Dodaj do projektu klasę **Osobniki** .
- Otwórz w edytorze plik **osobniki.h** i usuń z tego pliku definicję klasy **Osobniki** .
- Otwórz w edytorze plik **osobniki.cpp** i usuń definicję konstruktora klasy **Osobniki**.
- Wróć do pliku **osobniki.h** i włącz pliki **organizm.h** oraz **mieszkaniec.h** .
- Dodaj definicję klasy **Glon** wg listingu 7.1.

Listing 7.1. Definicja klasy Glon

```
1 class Glon: protected Organizm, public Mieszkaniec {
2     public:
3         Glon();
4         virtual RodzajMieszkanca kimJestes() final;
5
6         virtual ZamiarMieszkanca
7             wybierzAkcje(Sasiedztwo sasiedztwo);
8
9         virtual Mieszkaniec * dajPotomka();
10
11        virtual void przyjmijZdobycz
12            (Mieszkaniec * mieszkaniac);
13 };
```

- Dodaj klasę definicję klasy **Grzyb** wg listingu 7.1.

Listing 7.2. Definicja klasy Grzyb

```
1 class Grzyb: protected Organizm, public Mieszkaniec {
2     public:
3         Grzyb();
4         virtual RodzajMieszkanca kimJestes() final;
5
6         virtual ZamiarMieszkanca
7             wybierzAkcje(Sasiedztwo sasiedztwo);
8
9         virtual Mieszkaniec * dajPotomka();
10
11        virtual void przyjmijZdobycz
12            (Mieszkaniec * mieszkaniac);
13 };
```

- Dodaj klasę definicję klasy **Bakteria** wg listingu 7.1.

Listing 7.3. Definicja klasy Bakteria

```
1 class Bakteria: protected Organizm, public Mieszkaniec {
2     public:
3         Bakteria();
4         virtual RodzajMieszkanca kimJestes() final;
5
6         virtual ZamiarMieszkanca
7             wybierzAkcje(Sasiedztwo sasiedztwo);
8
9         virtual Mieszkaniec * dajPotomka();s
10        virtual void przyjmijZdobycz
11            (Mieszkaniec * mieszkaniac);
12 };
```



- Otwórz w edytorze plik **osobniki.h** i włącz pliki wg listingu 7.4. oraz zdefiniuj referencje do obiektu ustawień symulacji.

Listing 7.4. Niezbędne włączenia w pliku **osobniki.cpp** i definicja referencji

```
1 #include "ustawienia.h"
2 #include "generatorlosowy.h"
3 #include "sasiedztwo.h"
4
5 static const UstawieniaSymulacji & UST
6     = UstawieniaSymulacji::pobierzUstawienia();
```

- Dodaj definicje konstruktorów klas **Glon**, **Grzyb** oraz **Bakteria** wg listingu 7.5.

Listing 7.5. Definicje konstruktorów klas **Glon**, **Grzyb** oraz **Bakteria**

```
1 Glon::Glon():
2     Organizm(GeneratorLosowy::
3         losujPomiedzy(UST.glonZycieMin, UST.glonZycieMax),
4         UST.glonLimitPosilkow, UST.glonKosztPotomka),
5     Mieszkaniec (UST.znakGlon){}
6
7 Grzyb::Grzyb():
8     Organizm (GeneratorLosowy::
9         losujPomiedzy(UST.grzybZycieMin, UST.grzybZycieMax),
10        UST.grzybLimitPosilkow, UST.grzybKosztPotomka),
11    Mieszkaniec (UST.znakGrzyb){}
12
13 Bakteria::Bakteria():
14    Organizm (GeneratorLosowy::
15        losujPomiedzy(UST.bakteriaZycieMin, UST.bakteriaZycieMax),
16        UST.bakteriaLimitPosilkow, UST.bakteriaKosztPotomka),
17    Mieszkaniec (UST.znakBakteria){}
```

- Zdefiniuj metody **kimJestes** dla klas **Glon**, **Grzyb** oraz **Bakteria** wg listingu 7.6.

Listing 7.6. Definicje metod **kimJestes** klas **Glon**, **Grzyb** oraz **Bakteria**

```
1 RodzajMieszkanca Glon::kimJestes() {
2     return zywy() ? GLON : TRUP;
3 }
4
5 RodzajMieszkanca Grzyb::kimJestes() {
6     return zywy() > 0 ? GRZYB : TRUP;
7 }
8
9 RodzajMieszkanca Bakteria::kimJestes(){
10     return zywy() ? BAKTERIA : TRUP;
11 }
```

- Zdefiniuj metody **dajPotomka** dla klas **Glon**, **Grzyb** oraz **Bakteria** wg listingu 7.7.



Listing 7.7. Definicje metod *dajPotomka* klas *Glon*, *Grzyb* oraz *Bakteria*

```
1  Mieszkaniec * Glon::dajPotomka() {
2      Mieszkaniec * m = nullptr;
3      if(potomek()) m = new Glon();
4      return m;
5  }
6
7  Mieszkaniec * Grzyb::dajPotomka() {
8      Mieszkaniec * m = nullptr;
9      if(potomek()) m = new Grzyb();
10     return m;
11 }
12
13 Mieszkaniec * Bakteria::dajPotomka() {
14     Mieszkaniec * m = nullptr;
15     if(potomek()) m = new Bakteria();
16     return m;
17 }
```

- Zdefiniuj metody *przyjmijZdobycz* dla klas *Glon*, *Grzyb* oraz *Bakteria* wg listingu 7.8.

Listing 7.8. Definicje metod *przyjmijZdobycz* klas *Glon*, *Grzyb* oraz *Bakteria*

```
1  void Glon::przyjmijZdobycz(Mieszkaniec * mieszkaniac) {
2      if(mieszkaniac != nullptr) delete mieszkaniac;
3  }
4
5  void Grzyb::przyjmijZdobycz(Mieszkaniec * mieszkaniac) {
6      if(mieszkaniac != nullptr){
7          if(mieszkaniac->kimJestes() == TRUP) posilek();
8          delete mieszkaniac;
9      }
10 }
11
12 void Bakteria::przyjmijZdobycz(Mieszkaniec * mieszkaniac){
13     if(mieszkaniac != nullptr){
14         RodzajMieszkanca r = mieszkaniac->kimJestes();
15         if(r == GLON || r == GRZYB) posilek();
16         delete mieszkaniac;
17     }
18 }
```



- Zdefiniuj metodę **wybierzAkcje** dla klasy **Glon** wg listingu 7.9.

Listing 7.9. Definicja metody **wybierzAkcje** klasy **Glon**

```
1  ZmiarMieszkanca Glon::wybierzAkcje(Sasiedztwo sasiedztwo)
2  {
3      krokSymulacji();
4
5      if(zywy() && paczkujacy() && sasiedztwo.ile(PUSTKA)>0)
6          return ZmiarMieszkanca(
7              POTOMEK, sasiedztwo.losujSasiada(PUSTKA));
8
9      if(zywy() && glodny()) posilek();
10
11     if(!zywy() && symbol != UST.znakTrup)
12         symbol = UST.znakTrup;
13
14     return ZmiarMieszkanca();
15 }
```

- Zdefiniuj metodę **wybierzAkcje** dla klasy **Grzyb** wg listingu 7.10.

Listing 7.10. Definicja metody **wybierzAkcje** klasy **Grzyb**

```
1  ZmiarMieszkanca Grzyb::wybierzAkcje(Sasiedztwo sasiedztwo)
2  {
3      krokSymulacji();
4
5      if(zywy() && paczkujacy() && sasiedztwo.ile(PUSTKA)>0)
6          return ZmiarMieszkanca(
7              POTOMEK, sasiedztwo.losujSasiada(PUSTKA));
8
9      if(zywy() && glodny() && sasiedztwo.ile(TRUP)>0)
10         return ZmiarMieszkanca(
11             ROZKLAD, sasiedztwo.losujSasiada(TRUP));
12
13     if(!zywy() && symbol != UST.znakTrup)
14         symbol = UST.znakTrup;
15
16     return ZmiarMieszkanca();
17 }
```



- Zdefiniuj metodę **wybierzAkcje** dla klasy **Bakteria** wg listingu 7.11.

Listing 7.11. Definicja metody **wybierzAkcje** klasy **Bakteria**

```
1  ZmiarMieszkanca Bakteria::wybierzAkcje(Sasiedztwo sasiedztwo)
2  {
3      krokSymulacji();
4
5      if(zywy() && paczkujacy() && sasiedztwo.ile(PUSTKA)>0)
6          return ZmiarMieszkanca(
7              POTOMEK, sasiedztwo.losujSasiada(PUSTKA));
8
9      if(zywy() && glodny() && sasiedztwo.ile(GLON)> 0)
10         return ZmiarMieszkanca(
11             POLOWANIE, sasiedztwo.losujSasiada(GLON));
12
13     if(zywy() && glodny() && sasiedztwo.ile(BAKTERIA)> 0)
14         return ZmiarMieszkanca(
15             POLOWANIE, sasiedztwo.losujSasiada(BAKTERIA));
16
17     if(!zywy() && symbol != UST.znakTrup)
18         symbol = UST.znakTrup;
19
20     return ZmiarMieszkanca();
21 }
```

#### Dyskusja – odpowiedz na pytania:

- Listing 7.1, wiersz 4 → Jaki jest efekt słowa **final** ?
- Listing 7.4, wiersz 5 i 6 → Co się tutaj dzieje ?
- Listing 7.5, wiersze od 1 do 5 → Wy tłumacz jak to działa?
- Listing 7.6, wiersz 2 → Czy można zastosować tutaj **if** ?
- Listing 7.7, wiersz 3 → Skąd się wzięła metoda **potomek** ?
- Listing 7.8 → Jaka jest różnica w algorytmach metod **przyjmijZdobycz** ?
- Listing 7.9 → Opisz algorytm metody.
- Listing 7.10 → Opisz algorytm metody.
- Listing 7.11 → Opisz algorytm metody.

#### Zadanie 7.2 Stworzenie klasy **Nisza**

Klasa **Nisza** będąc pojedynczą komórką środowiska (Rys. 1.) jest kontenerem na pojedynczy obiekt klasy **Mieszkaniec** lub obiekt klasy potomnej klasy **Mieszkaniec** (**Glon**, **Grzyb**, **Bakteria**). Klasa **Nisza** (jako pole prywatne) posiada wskaźnik do klasy **Mieszkaniec** o nazwie **lokator** i w ten sposób może „umieścić” w sobie mieszkańca. Wtedy staje się zajęta. Może też być pusta – wskaźnik wówczas ma wartość **nullptr**. Klasa **Nisza** posiada metody za pomocą, których można sprawdzić stan lokatora, jego zamiar oraz przeprowadzać akcje symulacyjne. Operacje wykonywane na niszach nie mogą prowadzić do powielenia mieszkańców. Zatem konstruktor kopiujący i operator przypisania muszą usuwać mieszkańca ze źródła.

Wykonaj polecenia:

- Dodaj do projektu klasę **Nisza**.
- W pliku nagłówkowym włącz pliki **sasiedztwo.h** oraz **mieszkaniec.h**.
- Dodaj definicję klasy **Nisza** wg listingu 7.12.

*Listing 7.12. Definicja klasy Nisza*

```
1  class Nisza
2  {
3  private:
4      Mieszkaniec * lokator;
5
6  public:
7      Nisza();
8      Nisza(Nisza & innaNisza);
9      ~Nisza();
10
11     Nisza& operator=(Nisza & innaNisza);
12
13     void przyjmijLokatora(Mieszkaniec * lokatorBezdomny);
14
15     Mieszkaniec * oddajLokatora();
16
17     bool zajeta() const {return lokator != nullptr;}
18
19     RodzajMieszkanca ktoTuMieszka() {
20         return zajeta() ? lokator->kimJestes(): PUSTKA;
21     }
22
23     bool lokatorZywy() const;
24
25     char jakiSymbol() const;
26
27 private:
28
29     ZamiarMieszkanca aktywujLokatora(
30         Sasiedztwo sasiedztwo)
31     {
32         return lokator->wybierzAkcje(sasiedztwo);
33     }
34
35     Mieszkaniec * wypuscPotomka() {
36         return lokator->dajPotomka();
37     }
38
39     void przyjmijZdobycz(Mieszkaniec * ofiara) {
40         lokator->przyjmijZdobycz(ofiara);
41     }
42 };
```



- W plik `nisza.cpp` dodaj definicje konstruktorów klasy **Nisza**.

Listing 7.13. Definicje konstruktorów klasy **Nisza**

```
1  Nisza::Nisza() :lokator(nullptr){}
2
3  Nisza::Nisza(Nisza & innaNisza){
4
5      if(innaNisza.zajeta()){
6          lokator = innaNisza.lokator;
7          innaNisza.lokator = nullptr;
8      }else{ lokator = nullptr; }
9  }
10
11 Nisza::~Nisza() {
12     if(lokator != nullptr) delete lokator;
13 }
```

- Dodaj definicję operatora przypisania klasy **Nisza** wg listingu 7.14.

Listing 7.14. Definicja operatora przypisania klasy **Nisza**

```
1  Nisza &Nisza::operator=(Nisza &innaNisza) {
2      Mieszkaniec * tmp = lokator;
3      lokator = innaNisza.lokator;
4      innaNisza.lokator = tmp;
5      return *this;
6  }
```

- Dodaj definicje metod `przyjmijLokatora` oraz `oddajLokatora` klasy **Nisza** wg listingu 7.15.

Listing 7.15. Definicje metod `przyjmijLokatora` i `oddajLokatora` klasy **Nisza**

```
1  void Nisza::przyjmijLokatora(Mieszkaniec *lokatorBezdomny{
2      if(!zajeta()){
3          lokator = lokatorBezdomny;
4          lokatorBezdomny = nullptr;
5      }
6  }
7
8  Mieszkaniec *Nisza::oddajLokatora() {
9      Mieszkaniec * tmp = nullptr;
10
11     if(zajeta()){
12         tmp = lokator;
13         lokator = nullptr;
14     }
15
16     return tmp;
17 }
```



- Dodaj definicje metod `lokatorZywy` oraz `jakiSymbol` klasy `Nisza` wg listingu 7.16.

Listing 7.16. Definicje metod `lokatorZywy` oraz `jakiSymbol` klasy `Nisza`

```
1  bool Nisza::lokatorZywy() const
2  {
3      if(zajeta()){
4          RodzajMieszkanca r = lokator->kimJestes();
5          return r==GLON || r==GRZYB || r==BAKTERIA;
6      } else return false;
7  }
8
9  char Nisza::jakiSymbol() const
10 {
11     if(!zajeta()) return UstawieniaSymulacji::
12         pobierzUstawienia().znakPustaNisza;
13     else return lokator->jakiSymbol();
14 }
```

#### Dyskusja – odpowiedz na pytania:

- Listing 7.12, wiersz 4 → Czy zamiast wskaźnika można użyć referencji ?
- Listing 7.12, wiersz 17 → Jak to działa ?
- Listing 7.12, wiersz 23 i 25 → Jakie konsekwencje pociąga za sobą słowo `const` ?
- Listing 7.12, wiersz 40 → Dlaczego użyto symbolu `->` a nie kropki ?
- Listing 7.13 → Czy operator kopiujący mógłby mieć nagłówek:  
`Nisza::Nisza(const Nisza & innaNisza)` ? Czy to jest typowy operator kopiujący?
- Listing 7.14 → Czy operator przypisania mógłby mieć nagłówek:  
`Nisza &Nisza::operator=(const Nisza & innaNisza)` ? Czy jest to typowy operator przypisania?

#### Zadanie 7.3. Test klas `Mieszkaniec` oraz `Nisza`

Wykonaj polecenia:

- Poprzednią zawartość pliku `main.cpp` zarchiwizuj, jeżeli już nie została zarchiwizowana.
- Przygotuj plik `main.cpp` wg listingu 7.17.
- Uruchom, zaobserwuj efekt działania, następnie plik `main.cpp` zarchiwizuj.

Listing 7.17. Plik `main.cpp` do wykonania testu klas `Mieszkaniec` oraz `Nisza`

```
1  #include <iostream>
2  using namespace std;
3
4  #include "nisza.h"
5  #include "osobniki.h"
6
7  static Nisza n1, n2, n3;
8
```



```
9 static char sep = UstawieniaSymulacji::
10     pobierzUstawienia().znakSeparator;
11
12 void wyswietlNisze()
13 {
14     cout << n1.jakiSymbol() << sep
15         << n2.jakiSymbol() << sep
16         << n3.jakiSymbol() << endl;
17 }
18
19 int main()
20 {
21     cout << "Puste nisze: ";
22     wyswietlNisze();
23
24     cout << "Lokowanie mieszkańców: ";
25     n1.przyjmijLokatora(new Glon());
26     n3.przyjmijLokatora(new Grzyb());
27     wyswietlNisze();
28
29     cout << "Przesuwanie lokatorów: ";
30     n2 = n1;
31     wyswietlNisze();
32
33     cout << "Przesuwanie lokatorów:";
34     n3 = n2;
35     wyswietlNisze();
36
37     cout << endl;
38     return 0;
39 }
```

#### Dyskusja – odpowiedz na pytania:

- Listing 7.17, wiersz 7 → Czy słowo **static** jest konieczne ?
- Listing 7.17, wiersz 30 → Jak to działa ?

#### Zadanie 7.4. Test określania sąsiedztwa

Wykonaj polecenia:

- Przygotuj plik `main.cpp` wg listingu 7.18.
- Uruchom, zaobserwuj efekt działania, następnie plik `main.cpp` zarchiwizuj.

*Listing 7.18. Plik `main.cpp` do wykonania testu określania sąsiedztwa*

```
1 #include <iostream>
2 using namespace std;
3
4 #include "nisza.h"
5 #include "osobniki.h"
```



```
6  #include "sasiedztwo.h"
7
8  int main()
9  {
10     Nisza nisza;
11     nisza.przyjmijLokatora(new Bakteria());
12
13     cout << "Kto w niszy: "
14           << nisza.ktoTuMieszka() << endl;
15     cout << "Czy żywy: "
16           << nisza.lokatorZywy() << endl;
17
18     Sasiedztwo sasiedztwo;
19     sasiedztwo.okreslSasiada(P, GLON);
20     sasiedztwo.okreslSasiada(PG, GRZYB);
21     sasiedztwo.okreslSasiada(G, GRZYB);
22     sasiedztwo.okreslSasiada(LG, GLON);
23     sasiedztwo.okreslSasiada(L, BAKTERIA);
24     sasiedztwo.okreslSasiada(LD, BAKTERIA);
25     sasiedztwo.okreslSasiada(D, GLON);
26     sasiedztwo.okreslSasiada(PD, PUSTKA);
27
28     ZamiarMieszkanca zamiar =
29         nisza.aktywujLokatora(sasiedztwo);
30
31     cout << "Akjca: " << zamiar.akcja << " gdzie: "
32           << zamiar.gdzie << endl;
33
34     cout << endl;
35     return 0;
36 }
```

#### **Dyskusja – odpowiedz na pytanie:**

- Czy metoda `aktywujLokatora` działa poprawnie?



## **LABORATORIUM 8. KOŁOKWIUM 1**

### **Cel laboratorium:**

Weryfikacja nabytych umiejętności posługiwania się podstawowymi mechanizmami programowania obiektowego.

### **Zakres tematyczny kolokwium:**

- Laboratorium od 2 do 7.

### **Pytania kontrolne:**

1. Warunkiem przystąpienia do kolokwium jest wykonanie wszystkich zadań z zakresu laboratoriów od 1 do 7 i wysłanie kopii prowadzącemu zajęcia.

### **Zadanie 8.1. \*Przykładowy problem zaliczeniowy 1**

Tematem zadania jest program do manipulowania figurami geometrycznymi. Do dyspozycji mamy koło, kwadrat i trójkąt równoboczny. Możemy: 1) dodać dowolną figurę do dowolnej – wtedy figura powiększa się, aby jej pole było większe o pole figury dodanej, 2) odjąć figury – analogicznie, 3) zamienić dowolną figurę na inną dowolną – wtedy pole ma być zachowane.

W programie zaprojektować klasy:

- Bazową **Figura**.
- Abstrakcyjną **Operacje** z metodami wirtualnymi: **dodaj**, **odejmij**, **zamien**. Metody te jako argument przyjmują obiekt klasy **Figura**.
- Klasy konkretne **Kolo**, **Kwadrat**, **TrojkatRownoboczny**.

### **Zadanie 8.2 \*Przykładowy problem zaliczeniowy 2**

Tematem zadania jest program wykonujący działania: dodaj, odejmij, pomnóż i podziel na liczbach: rzeczywistych i zespolonych.

W programie zaprojektować klasy:

- Bazową **Liczba**.
- Abstrakcyjną **Operacje** z metodami wirtualnymi: **dodaj**, **odejmij**, **pomnóż** i **podziel**. Metody te jako argument przyjmują obiekt klasy **Liczba**.
- Klasy konkretne **LiczbaRzeczywista** oraz **LiczbaZespolona**.



## **LABORATORIUM 9. DYNAMICZNA ALOKACJA PAMIĘCI. REFERENCJE I WSKAŹNIKI (CZ.3).**

### **Cel laboratorium:**

Pierwszym celem laboratorium jest zestawienie wiadomości na temat rodzajów polimorfizmów dostępnych w języku C++, tzn. wiedzy która była już praktycznie wykorzystywana w trakcie poprzednich zajęć. Drugim celem zajęć jest wprowadzenie do dynamicznej alokacji pamięci.

### **Zakres tematyczny zajęć:**

- Rodzaje polimorfizmów w języku C++ – podsumowanie.
- Omówienie kolokwium z poprzednich zajęć ze zwróceniem uwagi na konieczność zastosowania polimorfizmu.
- Dynamiczna alokacja pamięci.
- Wskaźniki jako narzędzie do zarządzania pamięcią.

### **Kompendium wiedzy:**

- Polimorfizm w języku C++ może być **statyczny** albo **dynamiczny**. Dodatkowo w języku C++ istnieje mechanizm **szablonów**, który niezależnie rozszerza wielopostaciowość funkcji.
- **Polimorfizm statyczny** oznacza, że dwie metody lub funkcje istniejące we wspólnym obszarze nazw (plik, klasa, przestrzeń nazw) mają taką samą nazwę, ale różnią się listą argumentów, co pozwala je odróżnić i umożliwia automatyczną zmianę nazwy tych metod jeszcze przed kompilacją, np. poprzez dodanie do nazwy metody nazw typów parametrów w tej kolejności w jakiej występują. Polimorfizm statyczny dotyczy funkcji i może występować niezależnie od klas. Mówimy, że nazwa takiej funkcji / metody jest **przeładowana** lub **przeciążona**.
- **Polimorfizm dynamiczny** może zaistnieć jedynie w przypadku hierarchii klas. W tym przypadku dwie metody mają taką samą nazwę i taką samą listę argumentów, ale zdefiniowane są w różnych obszarach nazw (różne klasy). Bez wirtualizacji tych metod, tzn. bez słowa **virtual** w definicji metod dochodzi do **przesłaniania**, tzn. w danej klasie mamy dostęp tylko do metody, która jest zdefiniowana w tej klasie. Natomiast, jeżeli słowo **virtual** zostało użyte możemy korzystać z polimorfizmu dynamicznego. Do tego jest potrzebny **wskaźnik do klasy bazowej**. Taki wskaźnik ustawiamy **na obiekt klasy potomnej** i wywołujemy metodę. Uruchamia się wtedy metoda klasy potomnej.
- Jeżeli obiekt klasy jest obsługiwany wskaźnikiem wówczas, w celu wywołania metody lub uzyskania dostępu do pola klasy stosujemy **operator ->** (strzałka), a nie **.** (kropka).
- **Przeciążanie operatorów** polega na dodaniu metod do klasy o takich samych nazwach jak symbole operatorów np.: **+**, **++**, **=**, **==**, **[]**, itd. Druga możliwość to zdefiniowanie funkcji zewnętrznej względem klasy, która nazywa się jak operator i jako argument przyjmuje m.in. obiekt naszej klasy. Istnieją zasady w jaki sposób można przeładowywać operatory. Nie wszystkie operatory podlegają dziedziczeniu.



- Mechanizmem przypominającym polimorfizm statyczny są tzw. **domyślne wartości argumentów** funkcji i metod. Jednak w tym przypadku istnieje tylko jedna funkcja lub metoda i nie można tutaj mówić o przeładowaniu nazwy.
- **Dynamiczna alokacja pamięci** w języku C++ realizowana jest przy pomocy operatora **new** lub **new []** natomiast zwalnianie pamięci odbywa się przy użyciu operatora **delete** lub **delete []**. Wersje operatora z nawiasami kwadratowymi dotyczą tablic. Operator **new** zwraca wskaźnik do zarezerwowanego obszaru pamięci.
- Wyczerpujący opis operatorów **new** i **delete** można znaleźć na stronach:  
[http://www.cplusplus.com/reference/new/operator%20new\[\]](http://www.cplusplus.com/reference/new/operator%20new[])  
[http://www.cplusplus.com/reference/new/operator%20delete\[\]](http://www.cplusplus.com/reference/new/operator%20delete[])  
[https://en.cppreference.com/w/cpp/memory/new/operator\\_new](https://en.cppreference.com/w/cpp/memory/new/operator_new)  
[https://en.cppreference.com/w/cpp/memory/new/operator\\_delete](https://en.cppreference.com/w/cpp/memory/new/operator_delete)
- **Referencja** w języku C++ jest aliasem, który może być realizowany jako wskaźnik stały lub inaczej w zależności od kontekstu i od kompilatora. Referencje są używane do skracania długich nazw, przekazywania argumentów funkcji i zwracania wartości funkcji.

#### **Pytania kontrolne – odpowiedz:**

1. Na czym polega polimorfizm statyczny?
2. Na czym polega polimorfizm dynamiczny?
3. Czy w języku C++ jest możliwe połączenie polimorfizmu dynamicznego i statycznego?
4. Czy mechanizm wartości domyślnych argumentów metod zalicza się do polimorfizmu?
5. Czy polimorfizm statyczny się dziedziczy?
6. Czy argumenty domyślne się dziedziczy?
7. Co się dzieje, gdy istnieje już polimorfizm, ale usunie się słowo **virtual**?
8. Po co są te „polimorfizmy”?
9. W jaki sposób w języku C++ pobiera się i zwalnia pamięć?

#### **Zadanie 9.1. Zbadanie rozmiarów wybranych typów wskaźnikowych**

Wykonaj polecenia:

- Otwórz projekt z poprzednich zajęć.
- Dodaj do projektu klasę **ZLab09**.
- Otwórz w edytorze plik **zlab09.h** i dodaj metodę publiczną **void rozmiary()**.
- Dodaj definicję metody w pliku **zlab09.cpp** wg listingu 9.1. wykonaj odpowiednie włączenie (**#include**).
- Uruchom metodę w funkcji **main**, zaobserwuj działanie.



Listing 9.1. Definicja metody rozmiary klasy ZLab09

```

1 void ZLab09::rozmiary()
2 {
3     cout<< "--->Rozmiary" << endl
4         << "    int : " << sizeof (int) << endl
5         << "    int * : " << sizeof (int *) << endl
6         << "    int ** : " << sizeof (int **) << endl
7         << "    int *** : " << sizeof (int ***) << endl
8         << "    Glon : " << sizeof(Glon) << endl
9         << "    Glon * : " << sizeof(Glon *) << endl
10        << "    Glon ** : " << sizeof(Glon **) << endl
11        << "    UstawieniaSymulacji : "
12        << sizeof (UstawieniaSymulacji) << endl
13        << "    UstawieniaSymulacji * : "
14        << sizeof (UstawieniaSymulacji*) << endl
15        << "long:" << sizeof (long)
16        << endl << endl;
17 }

```

**Dyskusja – odpowiedz na pytanie:**

- Czy rozmiar zmiennej wskaźnikowej zależy od typu na jaki ten wskaźnik pokazuje?

**Zadanie 9.2. Alokacja pamięci dla zmiennej skalarnej**

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab09.h` i dodaj pola publiczne:  
`int * wskInt1D = nullptr` oraz `int ** wskInt2D = nullptr`.
- Otwórz w edytorze plik `zlab09.h` i dodaj metodę publiczną `void skalar()`.
- Dodaj definicje metody w pliku `zlab09.cpp` wg listingu 9.2.
- Uruchom metodę w funkcji `main`, zaobserwuj działanie.

Listing 9.2. Definicja metody skalar klasy ZLab09

```

1 void ZLab09::skalar()
2 {
3     cout << "--->Skalar" << endl;
4     wskInt1D = new int;
5     *wskInt1D = 5;
6     cout << "*wskInt1D = " << *wskInt1D << endl
7         << "*wskInt1D + 3 = " << *wskInt1D + 3 << endl
8         << "adres zapisany w wskInt1D ->" << wskInt1D
9         << endl << endl;
10
11     delete wskInt1D;
12 }

```



### Dyskusja – odpowiedź na pytania:

- Listing 9.2, wiersz 11 → Jaki byłby efekt, gdyby tego wiersza nie było ?
- Listing 9.2, wiersze 5 i 7 → Po co jest znak \* ?
- Co oznacza zapis: `int ** wskInt2D = nullptr` ?

### Zadanie 9.3. Alokacja pamięci dla tablicy jednowymiarowej

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab09.h` i dodaj metodę publiczną `void tablica1D()`.
- Dodaj definicję metody w pliku `zlab09.cpp` wg listingu 9.3.
- Uruchom metodę w funkcji `main`, zaobserwuj działanie.

Listing 9.3. Definicja metody `tablica1D` klasy `ZLab09`

```
1 void ZLab09::tablica1D()
2 {
3     cout << "--->Tablica 1D" << endl;
4     wskInt1D = new int [3];
5
6     wskInt1D[0] = 2;
7     *(wskInt1D+1) = 5;
8     wskInt1D[2] = 7;
9
10    cout << *wskInt1D << " " << wskInt1D[1]
11         << " " << *(wskInt1D+2) << endl;
12
13    int * iter = wskInt1D;
14    cout << *iter << " ";
15    iter++;
16    cout << *iter << " ";
17    iter++;
18    cout << *iter << endl;
19
20    delete [] wskInt1D;
21 }
```

### Dyskusja – odpowiedź na pytania:

- Listing 9.3, wiersz 4 → Ile pamięci zostanie załokowane ?
- Listing 9.3, wiersze 7 i 11 → Co powoduje operator + ?
- Listing 9.3, wiersze 15 i 17 → Co powoduje operator ++ ?

### Zadanie 9.4. Alokacja pamięci dla tablicy dwuwymiarowej

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab09.h` i dodaj metodę publiczną `void tablica2D(int w, int k)`.
- Dodaj definicję metody w pliku `zlab09.cpp` wg listingu 9.4.
- Uruchom metodę w funkcji `main`, zaobserwuj działanie.

Listing 9.4. Definicja metody `tablica2D` klasy `ZLab09`

```
1 void ZLab09::tablica2D(int w, int k)
2 {
3     cout << "--->Tablica 2D" << endl;
4
5     wskInt2D = new int*[w];
6
7     for(int i=0; i<w; i++) wskInt2D[i] = new int(k);
8
9     for(int i=0; i<w; i++)
10         for(int j=0; j<k; j++)
11             wskInt2D[i][j] = i + j;
12
13     for(int i=0; i<w; i++)
14     {
15         for(int j=0; j<k; j++)
16             cout << wskInt2D[i][j] << " ";
17         cout << endl;
18     }
19
20     for(int i=0; i<k; i++) delete [] wskInt2D[i];
21     delete [] wskInt2D;
22
23 }
```

#### Dyskusja – odpowiedz na pytania:

- Dlaczego lokowanie pamięci odbywa się dwuetapowo?
- Dlaczego zwalnianie pamięci odbywa się dwuetapowo?
- Wykonaj odpowiednią ilustrację.

#### Zadanie 9.5. Referencje a wskaźniki

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab09.h` i dodaj metodę publiczną `void referencja()`.
- Dodaj definicje metody w pliku `zlab09.cpp` wg listingu 9.5.
- Uruchom metodę w funkcji `main`, zaobserwuj działanie.

Listing 9.5. Definicja metody referencja klasy ZLab09

```

1 void ZLab09::referencja()
2 {
3     cout << "---> Referencje" << endl;
4
5     //int & refInt = new int;
6     //int & refInt = 4;
7
8     int a = 1, b = 5;
9     cout << "a=" << a << "  b=" << b << endl;
10
11     int & refInt = a; //łączenie z a
12     cout << "a=" << a << "  b=" << b << endl;
13
14     refInt = b; //?łączenie z b?
15     cout << "a=" << a << "  b=" << b << endl;
16
17     cout << "To samo ale na wskaźnikach" << endl;
18     int A = 1, B = 5;
19     cout << "A=" << A << "  B=" << B << endl;
20
21     int * wskInt = &A;
22     cout << "A=" << A << "  B=" << B << endl;
23
24     wskInt = &B;
25     cout << "A=" << A << "  B=" << B << endl;
26
27     Glon glon;
28     Glon &refGlon = glon;
29
30     cout << "Rozmiary" << endl
31         << "sizeof (int &)=" << sizeof (int &) << endl
32         << "sizeof(refInt)=" << sizeof(refInt) << endl
33         << "sizeof(Glon &)=" << sizeof(Glon &) << endl
34         << "sizeof(refGlon)=" << sizeof(refGlon) << endl;
35 }

```

**Dyskusja – odpowiedz na pytania:**

- Listing 9.5, wiersze 5,6 → Dlaczego to nie zadziała?
- Listing 9.5, wiersz 14 → Co się tutaj dzieje?
- Listing 9.5, wiersze 14 i 24 → Jaka jest różnica?
- Czy rozmiar referencji zależy od typu na jaki ten wskaźnik pokazuje?  
(Czy takie pytanie ma w ogóle sens?)



## **LABORATORIUM 10. BIBLIOTEKA STL I ALGORYTMY. PRZECIĄŻANIE OPERATORÓW.**

### **Cel laboratorium:**

Celem laboratorium jest demonstracja wybranych elementów biblioteki STL i biblioteki algorytmów języka C++ oraz ćwiczenia praktyczne związane z przeciążaniem operatorów.

### **Zakres tematyczny zajęć:**

- Demonstracja klasy kontenerowej vector.
- Demonstracja klasy kontenerowej set.
- Demonstracja klasy kontenerowej map.
- Demonstracja przeciążenia operatora >> .
- Demonstracja przeciążenia operatora << .
- Demonstracja przeciążenia operatora ++ .
- Demonstracja przeciążenia operatora ! .

### **Kompendium wiedzy:**

- **Biblioteka STL** (Standard Template Library) jest bardzo popularną i często stosowaną biblioteką języka C++. Obejmuje **kontenery**, **iteratory** oraz **algorytmy** do stosowania na obiektach w kontenerach.
- Jest to **biblioteka standardowa** co oznacza, że po włączeniu odpowiednich plików nagłówkowych jest gotowa do użycia.
- **Kontenery** to pojemniki na obiekty różnego rodzaju. Cechą wspólną kontenerów jest automatyczna zmiana rozmiaru w zależności od liczby dodanych elementów.
- **Wektor** (vector) jest dynamiczną tablicą jednowymiarową. Elementy zajmują indeksowane miejsca i mogą się powtarzać. Kontener automatycznie dopasowuje swój rozmiar do liczby elementów.
- **Zbiór** (set) jest również kontenerem dynamicznym, ale nie pozwala, aby elementy w nim zawarte się powtarzały. Mówimy, że zbiór przechowuje klucze, czyli obiekty unikalne.
- **Mapa** jest tablicą dynamiczną do przechowywania par **klucz–wartość**. Klucze są „indeksami położenia” i nie mogą się powtarzać. Wartości mogą się powtarzać, ale dla innych kluczy.
- **Iterator** służy do sekwencyjnego przeglądania kontenera i przypomina wskaźnik związany z tym kontenerem. Wskaźnik taki można ustawić na dowolny element kontenera, przesunąć do przodu lub do tyłu. Za pomocą iteratora uzyskuje się dostęp do elementu kontenera.
- Biblioteka STL jest **biblioteką generyczną**, tzn. komponenty biblioteki są szablonami. Stąd konieczność podawania typów w nawiasach ostrych przy każdej konkretyzacji szablonu.

### **Pytania kontrolne – odpowiedz:**

- 1 . Czym różni się wektor od „zwykłej” tablicy dynamicznej?
- 2 . Czym różni się wektor od zbioru?





3. Czym różni się mapa od wektora?
4. Co ma wspólnego zbiór z mapą?
5. Jak nazywa się specjalny wskaźnik przewidziany do pracy z kontenerami?
6. W jaki sposób przeciąża się operator `>>` ?
7. W jaki sposób przeciąża się operator `<<` ?
8. W jaki sposób przeciąża się operator `++` ?
9. W jaki sposób przeciąża się operator `!` ?

### Zadanie 10.1. Demonstracja klasy `vector`

Wykonaj polecenia:

- Otwórz projekt z poprzednich zajęć.
- Dodaj do projektu klasę `ZLab10`.
- Otwórz w edytorze plik `zlab10.h` i dodaj metodę publiczną `void wektor()`.
- Dodaj definicje metody w pliku `zlab10.cpp` wg listingu 10.1. wykonaj odpowiednie włączenie (`#include`).
- Uruchom metodę w funkcji `main`, zaobserwuj działanie.

Listing 10.1. Definicja metody wektor klasy `ZLab10`

```
1 void ZLab10::wektor()
2 {
3     vector<double> wektor;
4
5     wektor.push_back(2.3);
6     wektor.push_back(4.5);
7     wektor.push_back(-2.3);
8     wektor.push_back(0.1);
9
10    cout << "Rozmiar wektora: " << wektor.size() << endl;
11    for(double d : wektor) cout << d << " ";
12    cout << endl << endl;
13
14    //Próba zapisu poza granicą
15    for(unsigned long i=5; i<10; i++) wektor[i]= 3.2;
16    cout << "Rozmiar wektora: " << wektor.size() << endl;
17    for(double d : wektor) cout << d << " ";
18    cout << endl << endl;
19
20    //Zwiększenie rozmiaru
21    wektor.resize(12);
22    for(unsigned long i=5; i<10; i++) wektor[i]= 3.2;
23    cout << "Rozmiar wektora: " << wektor.size() << endl;
24    for(double d : wektor) cout << d << " ";
25    cout << endl << endl;
26
27    //Zapis przy użyciu zwykłej pętli
28    for(unsigned long i=0; i<wektor.size(); i++)
29        wektor[i]= i;
```

```
30     cout << "Rozmiar wektora: " << wektor.size() << endl;
31     for(double d : wektor) cout << d << " ";
32     cout << endl << endl;
33
34     //Zapis przy użyciu pętli zakresowej
35     for(double & d: wektor) d = 3.14;
36
37     for(double d : wektor) cout << d << " ";
38     cout << endl << endl;
39
40     //Czyszczenie
41     wektor.clear();
42     cout << "Rozmiar wektora: " << wektor.size() << endl;
43     for(double d : wektor) cout << d << " ";
44     cout << endl << endl;
45 }
```

#### Dyskusja – odpowiedz na pytania:

- Listing 10.1, wiersze 5,6,7 i 8 → Do czego służy metoda `push_back` ?
- Listing 10.1, wiersz 10 → Do czego służy metoda `size` ?
- Listing 10.1, wiersz 17 → Jak działa pętla `for` ?
- Listing 10.1, wiersz 21 → Do czego służy metoda `resize` ?
- Listing 10.1, wiersz 35 → Dlaczego została użyta referencja ?
- Listing 10.1, wiersz 41 → Do czego służy metoda `clear` ?

#### Zadanie 10.2. Demonstracja klasy `sort`

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab10.h` i dodaj metodę publiczną:  
`void wektor_sortowanie()` .
- Dodaj definicję metody w pliku `zlab10.cpp` wg listingu 10.2.
- Uruchom metodę w funkcji `main`, zaobserwuj działanie.

*Listing 10.2. Definicja metody `wektor_sortowanie` klasy `ZLab10`*

```
1 void ZLab10::wektor_sortowanie()
2 {
3     vector<int> wektor;
4
5     wektor.resize(13);
6
7     for(int& j: wektor) j= GEN::losujOdZeraDo(20);
8
9     cout << "Wektor wygenerowany" << endl;
10    for(auto j : wektor) cout << j << " ";
11    cout << endl << endl;
12
13    cout << "Wektor posortowany" << endl;
```



```
14     sort(wektor.begin(), wektor.end());
15     for(auto j : wektor) cout << j << " ";
16     cout << endl << endl;
17
18     cout << "Wektor pomieszany" << endl;
19     random_shuffle(wektor.begin(), wektor.end(),
20                   GEN::losujOdZeraDo);
21     for(auto j : wektor) cout << j << " ";
22     cout << endl << endl;
23 }
```

### Dyskusja – odpowiedz na pytania:

- Listing 10.2, wiersz 7 → Skąd się wziął GEN ?
- Listing 10.2, wiersz 14 → Jak określa się zakres sortowania ?

### Zadanie 10.3. Demonstracja klasy **set**

Wykonaj polecenia:

- Otwórz w edytorze plik **zlab10.h** i dodaj metodę publiczną: **void zbior()** .
- Dodaj definicję metody w pliku **zlab10.cpp** wg listingu 10.3.
- Uruchom metodę w funkcji **main**, zaobserwuj działanie.

Listing 10.3. Definicja metody **zbior** klasy **ZLab10**

```
1  void ZLab10::zbior() {
2      set<string> imiona;
3      imiona.insert("Adam");
4      imiona.insert("Ewa");
5      imiona.insert("Maciek");
6
7      for(string s: imiona) cout << s << " ";
8      cout << endl << endl;
9
10     imiona.insert("Adam");
11     imiona.insert("Iwona");
12     imiona.insert("Ewa");
13     imiona.insert("Beata");
14     imiona.insert("Maciek");
15
16     for(string s: imiona) cout << s << " ";
17     cout << endl << endl;
18
19     set<string>::iterator iter = imiona.begin();
20     while(iter != imiona.end()){
21         cout << *iter << endl;
22         iter++;
23     }
24 }
```



### Dyskusja – odpowiedź na pytania:

- Listing 10.3, wiersz 3 → Do czego służy metoda `insert` ?
- Listing 10.3 → W jaki sposób zostało pokazane, że zbiór nie powieli elementów?
- Listing 10.3, wiersz 19 → Jaka jest początkowa pozycja iteratora?
- Listing 10.3, wiersz 20 → Jak sprawdza się czy iterator osiągnął ostatni element w kontenerze?
- Listing 10.3, wiersz 22 → Jak przesuwa się iterator ?

### Zadanie 10.4. Demonstracja klasy `map`

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab10.h` i dodaj metodę publiczną: `void mapa()` .
- Dodaj definicje metody w pliku `zlab10.cpp` wg listingu 10.4.
- Uruchom metodę w funkcji `main`, zaobserwuj działanie.

Listing 10.4. Definicja metody `mapa` klasy `ZLab10`

```
1 void ZLab10::mapa()
2 {
3     map<char,double> pomiary;
4
5     pomiary.insert(pair<char,double>('A',20));
6     pomiary.insert(pair<char,double>('B',30));
7     pomiary.insert(pair<char,double>('C',50));
8     pomiary.insert(pair<char,double>('D',25));
9     pomiary.insert(pair<char,double>('A',20));
10    pomiary.insert(pair<char,double>('A',40));
11
12    cout << "Wyświetlenie mapy" << endl;
13    for( char c : {'A','B','C'})
14        cout << c << "->" << pomiary[c] << endl;
15
16    cout << "Policz A -> " << pomiary.count('A') << endl;
17
18    pomiary.erase('A');
19    cout << "Wyświetlenie mapy" << endl;
20    for( char c : {'A','B','C'})
21        cout << c << "->" << pomiary[c] << endl;
22    cout << "Policz A -> " << pomiary.count('A') << endl;
23 }
```

### Dyskusja – odpowiedź na pytania:

- Listing 10.4, wiersz 3 → Jak definiuje się obiekt klasy `map` ?
- Listing 10.4, wiersze od 5 do 10 → W jaki sposób umieszcza się elementy w mapie?
- Listing 10.4, wiersz 14 → W jaki sposób uzyskuje się dostęp do elementu mapy?
- Listing 10.4, wiersz 16 → Do czego służy metoda `count` ?
- Listing 10.4, wiersz 18 → Do czego służy metoda `erase` ?

### Zadanie 10.5. Przeciążenie operatora >>

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab03.h` i poniżej definicji klasy `Prostokat` dodaj deklaracje:  
`std::ostream& operator<<(std::ostream & strumien,  
Prostokat & prostokat);`  
Potrzebne będzie też włączenie: `#include <iostream>`
- Dodaj definicje operatora w pliku `zlab03.cpp` wg listingu 10.5.
- Przetestuj działanie operatora w funkcji `main` wg listingu 10.6, zaobserwuj działanie.

Listing 10.5. Definicja operatora >> dla klasy Prostokat

```
1 std::ostream& operator<<(std::ostream & strumien,  
2                           Prostokat & prostokat)  
3 {  
4     strumien << prostokat.doTekstu();  
5     return strumien;  
6 }
```

Listing 10.6. Kod do uruchomienia w funkcji main

```
1 Prostokat p("Prostokat", 2, 3);  
2 Kwadrat k("Kwadrat", 4);  
3 cout << p << endl;  
4 cout << k << endl;
```

### Dyskusja – odpowiedz na pytania:

- Listing 10.5, wiersze 1 i 2 → Czy argumenty mogą być przekazane jako referencje do stałych?
- Listing 10.6 → Czy obiekt klasy Kwadrat wyświetla się poprawnie? Dlaczego?

### Zadanie 10.6. Przeciążenie operatora <<

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab03.h` i dodaj deklaracje:  
`std::istream& operator>>(std::istream & strumien,  
Prostokat & prostokat);`  
!UWAGA: Teraz jest `istream` a nie `ostream`.
- Dodaj definicje operatora w pliku `zlab03.cpp` wg listingu 10.7.
- Przetestuj działanie operatora w funkcji `main` wg listingu 10.8.  
!UWAGA: Należy wprowadzić dwie liczby rozdzielone spacją i nacisnąć Enter.

Listing 10.7. Definicja operatora << dla klasy Prostokat

```
1 std::istream& operator>>(std::istream & strumien,  
2                           Prostokat & prostokat){  
3     double a, b;  
4     strumien >> a >> b;  
5     prostokat.zmienBoki(a, b);  
6     return strumien; }
```



Listing 10.8. Kod do uruchomienia w funkcji main

```
1 Prostokat p("Prostokat", 2, 3);
2 cout << p << endl;
3 cin >> p;
4 cout << p << endl;
```

#### Dyskusja – odpowiedz na pytania:

- Listing 10.7, wiersze 1 i 2 → Czy argumenty mogą być przekazane jako referencje do stałych?

#### Zadanie 10.7. Przeciążenie operatora ++

Wykonaj polecenia:

- Otwórz w edytorze plik `zlab03.h` i dodaj do klasy `Prostokat` deklarację publicznej metody: `Prostokat & operator++(int);`
- Dodaj definicje tej metody w pliku `zlab03.cpp` wg listingu 10.9.
- Przetestuj działanie operatora w funkcji `main` wg listingu 10.6.

Listing 10.9. Definicja operatora postinkrementacji ++ dla klasy Prostokat

```
1 Prostokat & Prostokat::operator++(int)
2 {
3     this->bok1 *=2;
4     this->bok2 *=2;
5     this->obliczPole();
6     this->obliczObwod();
7     return *this;
8 }
```

Listing 10.10. Kod do uruchomienia w funkcji main

```
1 Prostokat p("Prostokat", 2, 3);
2 cout << p << endl;
3 p++;
4 cout << p << endl;
```

#### Dyskusja – odpowiedz na pytania:

- Jaka jest różnica pomiędzy operatorami **post**inkrementacji i **pre**inkrementacji ?
- Listing 10.9, wiersz 1 → Po co jest ten `int` ?
- Listing 10.9, wiersze od 3 do 7 → Co to jest ten `this` ?
- Listing 10.9, wiersze od 3 do 7 → Czy `this` jest niezbędny ?



### Zadanie 10.7. Przeciążenie operatora !

Wykonaj polecenia:

- Otwórz w edytorze plik `z1ab03.h` i dodaj do klasy `Prostokat` publiczną metodę:

```
bool operator!() {return czyPoprawny();}
```

!UWAGA: Tutaj od razu definicja operatora w definicji klasy.

- Przetestuj działanie operatora w funkcji `main` wg listingu 10.11, zaobserwuj działanie.

*Listing 10.11. Kod do uruchomienia w funkcji main*

```
1 Prostokat p0("Prostokat",2,3);
2 cout << "!p0=" <<!p0 << endl;
3 Prostokat p1("Prostokat",0,0);
4 cout << "!p1=" <<!p1 << endl;;
```

### Dyskusja – odpowiedz na pytania:

- Co to oznacza, że definicja metody jest wewnątrz definicji klasy?
- Listing 10.11, wiersze 2 i 3 → W jaki sposób wyświetlana jest zmienna typu `bool`?



## LABORATORIUM 11. PRAKTYKA PROGRAMOWANIA OBIEKTOWEGO. URUCHOMIENIE I TESTOWANIE APLIKACJI WIRTUALNY EKOSYSTEM

### Cel laboratorium:

Stworzenie ostatniej klasy o nazwie **Srodowisko**. Uruchomienie i testowanie aplikacji *Wirtualny Ekosystem*.

### Zakres tematyczny zajęć:

- Praktyka programowania obiektowego.

### Kompendium wiedzy:

- W języku C++ istnieje możliwość wywołania komendy systemu operacyjnego z poziomu kodu. Służy do tego funkcja `system` np.: `system("clear")` wyczyści konsolę.
- Metoda `ignore()` odbiera pojedynczy znak ze strumienia `cin` po czym go ignoruje. Używana jest w programie do wstrzymania pracy. Jeżeli strumień jest pusty to program czeka na wciśnięcie klawisza Enter.

### Zadanie 11.1. Uzupełnienie klasy **GeneratorLosowy**

Celem zadania jest uzupełnienie klasy **GeneratorLosowy** o metodę `indeksyLosowe`, która dla zadanej liczby wierszy i kolumn macierzy generuje wektor ze wszystkimi położeniami w tej tablicy w postaci par {wiersz,kolumna}. Kolejność tych par jest losowa, co pozwala na dostęp do każdego elementu tablicy w przypadkowej kolejności.

Wykonaj polecenia:

- Otwórz plik `generatorlosowy.h`.
- Dodaj definicję klasy `Indeks2D` i definicję typu `WektorIndeks2D` wg listingu 11.1.

*Listing 11.1. Definicja klasy Indeks2D i typu WektorIndeks2D*

```
1 class Indeks2D{
2 public:
3     unsigned int wiersz, kolumna;
4     Indeks2D(unsigned int _w=0, unsigned int _k=0)
5         :wiersz(_w),kolumna(_k){}
6 };
7
8 typedef std::vector<Indeks2D> WektorIndeksow2D;
```

- Dodaj deklarację publicznej metody o nazwie `indeksyLosowe` w klasie **GeneratorLosowy**:  
`static WektorIndeksow2D indeksyLosowe(unsigned int wiersze, unsigned int kolumny);`

- Dodaj definicję metody `indeksyLosowe` w pliku `generatorlosowy.cpp` wg listingu 11.2.
- W celu sprawdzenia działania metody uruchom w funkcji `main` kod z listingu 11.3. Pamiętaj o włączeniu pliku `generatorlosowy.h`.

Listing 11.2. Definicja metody `indeksyLosowe` klasy `GeneratorLosowy`

```
1  #include <algorithm>
2
3  WektorIndeksow2D GeneratorLosowy::
4  indeksyLosowe(unsigned int wiersze, unsigned int kolumny)
5  {
6      WektorIndeksow2D indeksy;
7
8      for(unsigned int w=0; w < wiersze; w++)
9          for(unsigned int k = 0; k < kolumny; k++)
10             indeksy.push_back(Indeks2D(w,k));
11
12     std::random_shuffle(indeksy.begin(),indeksy.end(),
13                         GeneratorLosowy::losujOdZeraDo);
14
15     return indeksy;
16 }
```

Listing 11.3. Kod do testowania metody `indeksyLosowe`

```
1  for(Indeks2D & I : GEN::indeksyLosowe(2,4))
2      cout << "{" << I.wiersz << ", " << I.kolumna << "}\n";
```

### Zadanie 11.2. Stworzenie klasy `Srodowisko`

Celem zadania jest stworzenie ostatniej klasy programu *Wirtualny Ekosystem* o nazwie `Srodowisko`. Klasa posiada dynamiczną tablicę obiektów `Nisza` (Rys.1.). Tablica ta jest tworzona w konstruktorze klasy i zwalniana w destruktorze. Konstruktor klasy przyjmuje rozmiar środowiska prostokątnego, czyli liczbę wierszy i kolumn. Metoda `lokuje` umieszcza wirtualny organizm we wskazanej niszy. Metoda `ustalSasiedztwo` określa sąsiedztwo wskazanej niszy. Metoda `liczba` wylicza ile jest w całym środowisku osobników zadanego rodzaju. Metoda `martwy` zwraca wartość `prawda`, jeżeli w całym środowisku nie ma żywych organizmów. Metoda `wykonajSkok` powoduje, że jeżeli we wskazanej niszy jest obiekt klasy `Grzyb` lub `Bakteria`, to ten obiekt zmienia losowo nisze na inną, jeżeli są nisze puste w sąsiedztwie. Metoda `wykonajAkcje` powoduje, że jeżeli we wskazanej niszy jest obiekt klasy `Glon`, `Grzyb` lub `Bakteria`, to wykonuje on swoją „funkcję życiową” w warunkach swojego sąsiedztwa. Metoda `krokSymulacji` przeprowadza całe środowisko z bieżącej konfiguracji do następnej. Metoda `doTekstu` tworzy obraz tekstowy środowiska.

Wykonaj polecenia:

- Dodaj do projektu klasę `Srodowisko`
- Dodaj definicję klasy jak na listingu 11.4.
- Otwórz w edytorze plik `srodowisko.cpp` i włącz pliki `nisza.h` `generatorlosowy.h` `ustawienia.h`.

- Dodaj definicje metod klasy **Srodowisko** wg listingów od 11.5 do 11.14.

*Listing 11.4. Definicja klasy Srodowisko*

```
1  #include <string>
2  #include "ustawienia.h"
3
4  class Nisza;
5  class Mieszkaniec;
6  class Sasiedztwo;
7
8  class Srodowisko{
9
10 public:
11     const unsigned int wiersze, kolumny;
12     const unsigned long liczbaNisz;
13
14 private:
15     Nisza ** nisza;
16
17 public:
18     Srodowisko(unsigned int _wiersze,
19                 unsigned int _kolumny);
20
21     ~Srodowisko();
22
23     void lokuj(Mieszkaniec * mieszkaniac,
24               unsigned int wiersz, unsigned int kolumna);
25
26
27     Sasiedztwo ustalSasiedztwo(unsigned int wiersz,
28                                 unsigned int kolumna) const;
29
30     unsigned long liczba(RodzajMieszkanca rodzaj) const;
31
32     bool martwy();
33
34     void wykonajSkok(unsigned int wiersz,
35                      unsigned int kolumna);
36
37     void wykonajAkcje(unsigned int wiersz,
38                       unsigned int kolumna);
39
40     void wykonajKrokSymulacji();
41
42     std::string doTekstu() const;
43 };
```



*Listing 11.5. Definicja konstruktora klasy Srodowisko*

```
1  Srodowisko::Srodowisko(unsigned int _wiersze,  
2                          unsigned int _kolumny)  
3      :wiersze(_wiersze),  
4        kolumny(_kolumny),  
5        liczbaNisz(_wiersze * _kolumny)  
6  {  
7  
8      niska = new Niska*[wiersze];  
9      for(unsigned int i=0; i<wiersze; i++)  
10         niska[i] = new Niska[kolumny];  
11 }
```

*Listing 11.6. Definicja destruktoru klasy Srodowisko*

```
1  Srodowisko::~~Srodowisko()  
2  {  
3      for(unsigned int i=0; i<wiersze; i++)  
4          delete [] niska[i];  
5      delete [] niska;  
6  }
```

*Listing 11.7. Definicja metody lokuj klasy Srodowisko*

```
1  void Srodowisko::lokuj(Mieszkaniec *mieszkaniec,  
2                          unsigned int wiersz, unsigned int kolumna)  
3  {  
4      if(wiersz<wiersze && kolumna < kolumny)  
5          niska[wiersz][kolumna].przyjmijLokatora(mieszkaniec);  
6  }
```



Listing 11.8. Definicja metody ustalSasiedztwo klasy Srodowisko

```
1  Sasiedztwo Srodowisko::ustalSasiedztwo(  
2      unsigned int wiersz, unsigned int kolumna) const  
3  {  
4      Sasiedztwo sasiedztwo(SCIANA);  
5  
6      long wiersz1, kolumna1;  
7  
8      for(Polozenie p : {P, PG, G, LG, L, LD, D, PD})  
9      {  
10         wiersz1 = wiersz;  
11         kolumna1 = kolumna;  
12  
13         Sasiedztwo::  
14             zmienIdeksyWgPolozenia(p,wiersz1,kolumna1);  
15  
16         if(wiersz1>=0 &&  
17             wiersz1<wiersze &&  
18             kolumna1>=0 &&  
19             kolumna1 < kolumny)  
20         {  
21             sasiedztwo.okreslSasiada(p,  
22                 nisza[wiersz1][kolumna1].ktoTuMieszka());  
23         }  
24     }  
25  
26     return sasiedztwo;  
27 }
```

Listing 11.9. Definicja metody liczba klasy Srodowisko

```
1  unsigned long Srodowisko::  
2      liczba(RodzajMieszkanca rodzaj) const  
3  {  
4      unsigned long licznik = 0;  
5  
6      for(unsigned int w=0; w<wiersze; w++)  
7          for(unsigned int k=0; k<kolumny; k++)  
8              if(nisza[w][k].ktoTuMieszka() == rodzaj)  
9                  licznik++;  
10  
11     return licznik;  
12 }
```

Listing 11.10. Definicja metody martwy klasy Srodowisko

```
1 bool Srodowisko::martwy()
2 {
3     return liczba(GLON)+liczba(GRZYB)+liczba(BAKTERIA)==0;
4 }
```

Listing 11.11. Definicja metody wykonajSkok klasy Srodowisko

```
1 void Srodowisko::wykonajSkok(unsigned int wiersz,
2                               unsigned int kolumna)
3 {
4
5     if(!nisza[wiersz][kolumna].lokatorZywy() ||
6         nisza[wiersz][kolumna].ktoTuMieszka()==GLON)return;
7
8     Sasiedztwo sasiedztwo =
9         ustalSasiedztwo(wiersz, kolumna);
10
11     if(sasiedztwo.ile(PUSTKA) > 0){
12
13         Polozenie polozenie =
14             sasiedztwo.losujSasiada(PUSTKA);
15
16         unsigned int wiersz1 = wiersz, kolumna1 = kolumna;
17
18         Sasiedztwo::zmienIdekisyWgPolozenia(polozenie,
19                                             wiersz1, kolumna1);
20
21         nisza[wiersz1][kolumna1] = nisza[wiersz][kolumna];
22     }
23 }
```

*Listing 11.12. Definicja metody wykonajAkcje klasy Srodowisko*

```
1 void Srodowisko::wykonajAkcje(unsigned int wiersz,  
2                               unsigned int kolumna)  
3 {  
4     if(!nisza[wiersz][kolumna].lokatorZywy()) return;  
5  
6     Sasiedztwo sasiedztwo  
7         = ustalSasiedztwo(wiersz,kolumna);  
8  
9     ZamiarMieszkanca zamiar =  
10        nisza[wiersz][kolumna].aktywujLokatora(sasiedztwo);  
11  
12    unsigned int wiersz1 = wiersz, kolumna1 = kolumna;  
13  
14    Sasiedztwo::zmienIdeksyWgPolozenia(zamiar.gdzie,  
15                                       wiersz1, kolumna1);  
16  
17    Mieszkaniec *m =nullptr;  
18  
19    switch (zamiar.akcja) {  
20    case NIC:  
21        wykonajSkok(wiersz,kolumna); break;  
22  
23    case POTOMEK:  
24        m = nisza[wiersz][kolumna].wypuscPotomka();  
25        nisza[wiersz1][kolumna1].przyjmijLokatora(m);  
26        break;  
27  
28    case POLOWANIE:  
29    case ROZKLAD:  
30        m = nisza[wiersz1][kolumna1].oddajLokatora();  
31        nisza[wiersz][kolumna].przyjmijZdobycz(m);  
32        break;  
33    }  
34 }
```

*Listing 11.13. Definicja metody wykonajKrokSymulacji klasy Srodowisko*

```
1 void Srodowisko::wykonajKrokSymulacji()  
2 {  
3     WektorIndeksow2D indeksyLosowe =  
4         GeneratorLosowy::indeksyLosowe(wiersze, kolumny);  
5  
6     for(Indeks2D indeks : indeksyLosowe)  
7         wykonajAkcje(indeks.wiersz, indeks.kolumna);  
8 }
```





Listing 11.14. Definicja metody doTekstu klasy Srodowisko

```

1  std::string Srodowisko::doTekstu() const
2  {
3      std::string tekst = "";
4      char sep = UstawieniaSymulacji::
5                  pobierzUstawienia().znakSeparator;
6
7      for(unsigned int w=0; w<wiersze; w++){
8          for(unsigned int k=0; k<kolumny; k++){
9              if(k>0) tekst += sep;
10
11              tekst += nisza[w][k].jakiSymbol();
12          }
13
14          tekst += '\n';
15      }
16
17      tekst += "\n  Glony * : "
18              + std::to_string(liczba(GLON))
19              + "\n  Grzyby # : "
20              + std::to_string(liczba(GRZYB))
21              + "\n  Bakterie @ : "
22              + std::to_string(liczba(BAKTERIA))
23              + "\n  Martwe + : "
24              + std::to_string(liczba(TRUP))
25              + '\n';
26
27      return tekst;

```

**Zadanie 11.3. Uruchomienie aplikacji *Wirtualny Ekosystem***

Wykonaj polecenia:

- Otwórz w edytorze plik `nisza.h` i włącz do niego plik `srodowisko.h`.
- Dodaj w definicji klasy `Nisza` deklaracje przyjaźni z metodą `wykonajAkcje` klasy `Srodowisko`.
- Listing 7.12, wiersz 28 → dodaj wpis:

```
friend void Srodowisko::wykonajAkcje(unsigned int wiersz,
                                     unsigned int kolumna);
```

- Przygotuj plik `main.cpp` wg listingu 11.15.
- Uruchom.
- Naciśnięcie klawisza Enter wykonuje krok symulacji.

Listing 11.15. Zawartość pliku main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  #include "nisza.h"
5  #include "osobniki.h"
6  #include "sasiedztwo.h"
7
8  int main()
9  {
10     Srodowisko ekoSystem(8,12);
11
12     ekoSystem.lokuj(new Glon(),0,10);
13     ekoSystem.lokuj(new Glon(),1,10);
14     ekoSystem.lokuj(new Glon(),1,13);
15     ekoSystem.lokuj(new Glon(),3,10);
16     ekoSystem.lokuj(new Grzyb(),1,11);
17     ekoSystem.lokuj(new Grzyb(),0,0);
18     ekoSystem.lokuj(new Bakteria(),3,3);
19     ekoSystem.lokuj(new Bakteria(),2,6);
20
21     unsigned long i = 0;
22
23     do{
24         system("clear");
25         cout << "Krok symulacji: " << i << endl
26             << endl << ekoSystem.doTekstu() << endl;
27
28         cin.ignore(1);
29         ekoSystem.wykonajKrokSymulacji();
30         i++;
31     } while(i < 200 && !ekoSystem.martwy());
32
33     cout << endl;
34     return 0;
35 }
```



#### Zadanie 11.4. Dodanie przeciążonych operatorów

Wykonaj polecenia:

- Otwórz w edytorze plik `srodowisko.h`.
- Dodaj wewnątrz definicji klasy `Srodowisko` definicje operatorów:

```
void operator++(int) { wykonajKrokSymulacji(); }
```

```
bool operator!() { return !martwy(); }
```

- Dodaj na zewnątrz definicji klasy `Srodowisko` deklaracje operatora:

```
std::ostream & operator<<(std::ostream & strumien,  
                           const Srodowisko & srodowisko);
```

- Dodaj definicję tego operatora w pliku `srodowisko.cpp` wg listingu 11.16

Listing 11.16 Definicja operatora << dla klasy Srodowisko

```
1 #include<iostream>  
2 std::ostream &operator<<(std::ostream &strumien,  
3                           const Srodowisko & srodowisko)  
4 {  
5     strumien << srodowisko.doTekstu();  
6     return strumien;  
7 }
```

- Zmień wewnątrz pętli w pliku `main.cpp` wg Listingu 11.17.

Listing 11.17 Zmiana kodu w pliku main.cpp

```
1 do{  
2     system("clear");  
3     cout << "Krok symulacji: " << i << endl;  
4     cout << endl << ekoSystem << endl;  
5     cin.ignore(1);  
6     ekoSystem++;  
7     i++;  
8 } while(i < 200 && !ekoSystem);
```

- Uruchom.

#### Dyskusja – odpowiedz na pytania:

- Listing 11.4, wiersz 15 → Co oznacza zapis `Nisza ** nisza` ?
- Listing 11.7 → Co się stanie przy próbie umieszczenia organizmu w już zajętej niszy?
- Listing 11.8 → Opisz algorytm metody.
- Listing 11.9 → Opisz algorytm metody.
- Listing 11.11 → Opisz algorytm metody.
- Listing 11.12 → Opisz algorytm metody.



## LABORATORIUM 12. KLASA NAPISÓW STRING, ZAPIS I ODCZYT PLIKÓW, FORMATOWANIE.

### Cel laboratorium:

Celem laboratorium jest wykonanie ćwiczeń programistycznych z zakresu obsługi standardowych strumieni dostępnych w środowisku języka C++ oraz uzupełnienie aplikacji *Wirtualny Ekosystem* o odczyt i zapis plików.

### Zakres tematyczny zajęć:

- Strumień jako abstrakcja sekwencyjnej wymiany danych.
- Strumień tekstowe.
- Strumień plikowe.

### Kompendium wiedzy:

- Cechą szczególną strumieni jest dostęp do ich zawartości. Jest to **dostęp sekwencyjny**, tzn. bajt po bajcie. Oznacza to, że nie można zapisać lub wydobyć ze strumienia bezpośrednio dowolnego bajtu, ale trzeba wcześniej przejść przez wszystkie znaki go poprzedzające.
- Jeżeli bajty w strumieniu są interpretowane jako typ `char` to mówimy, że strumień jest **znakowy**.
- Do strumienia można bajty/znaki dopisywać na koniec lub nadpisywać od wskazanego miejsca.
- Ze strumienia można pobierać bajty/znaki w takiej kolejności w jakiej są dostępne od wskazanego miejsca np. od początku.
- Można określić miejsce (pozycję) w strumieniu od której nastąpi pisanie lub czytanie. Służą do tego metoda **seekg**.
- Zapis do strumienia odbywa się za pomocą operatora `<<`, a odczyt za pomocą operatora `>>`.
- Strumienie posiadają wewnętrzny **bufor**, czyli tablicę na bajty / znaki, który zmienia automatycznie swój rozmiar w miarę potrzeb. Bufor można również wyczyścić.
- Strumieniami są klasy do obsługi plików: **ofstream**, **ifstream** oraz **fstream**.
- Strumieniami są klasy: standardowego wyjścia **cout** oraz standardowego wejścia **cin**.
- Strumienie tekstowe, czyli klasy **ostream**, **istream** oraz **stringstream** służą do wygodnego formatowania i parsowania tekstów.

### Pytania kontrolne:

1. Omów czym różni się klasa `cout` od klasy `cin`.
2. Wyłumacz czym są klasy `ofstream` oraz `ifstream` i do czego służą.
3. Wyjaśnij czym są klasy `stringstream` i `istream`.

### Zadanie 12.1. Demonstracja użycia strumieni

Wykonaj polecenia:

- Dodaj do projektu klasę `ZLab12`

- W definicji tej klasy zadeklaruj metody publiczne zgodnie z listingiem 12.1.
- Następnie w pliku źródłowym włącz pliki: `iostream` `sstream` `fstream`.
- Dodaj definicje metod w pliku źródłowym klasy wg listingów od 12.2. do 12.6.
- Uruchom po kolei metody w funkcji `main`.
- Odszukaj utworzone pliki na dysku.

Listing 12.1. Deklaracje metod w klasie ZLab12

```
1  #include <string>
2
3  class ZLab12
4  {
5  public:
6      void strumienTekstowy();
7      void zapisTekstowyDoPliku(std::string nazwaPliku);
8      void odczytPlikuTekstowego(std::string nazwaPliku);
9      void zapisBinarnyDoPliku(std::string nazwaPliku);
10     void odczytPlikuBinarnego(std::string nazwaPliku);
11 };
```

Listing 12.2. Definicja metody `strumienTekstowy` klasy ZLab12

```
1  void ZLab12::strumienTekstowy()
2  {
3      //Formatowanie
4      ostreamstream strumien1;
5
6      strumien1 << "Początek tekstu" << endl
7              << "Liczba1: " << 2 << endl
8              << "Liczba2: " << 3.14 << endl;
9
10     cout << strumien1.str() << endl;
11
12     //Parsowanie
13     string liniaDanych = "Liczby: 3.4 4.5 -3.2 aa";
14
15     string opis;
16     double x, y, z;
17
18     istringstream strumien2(liniaDanych);
19
20     strumien2 >> opis >> x >> y >> z;
21
22     if(!strumien2.fail())
23         cout << "opis: " << opis << endl
24             << "x=" << x << endl
25             << "y=" << y << endl
26             << "z=" << z << endl;
27     else cout <<"Nie udało się!" << endl << endl;
28 }
```



Listing 12.3. Definicja metody zapisTekstowyDoPliku klasy ZLab12

```
1 void ZLab12::zapisTekstowyDoPliku(string nazwaPliku)
2 {
3     ofstream plik;
4
5     cout << "Otwieram plik." << endl;
6     plik.open(nazwaPliku, ios_base::out);
7
8     if(plik.is_open()){
9         cout << "Zapisuje do pliku" << endl;
10        for(int i=1; i<=10; i++)
11            plik << i <<" " << i*i << " "
12            << i*i*i << endl;
13    }
14
15    plik.close();
16
17    cout << "Plik zamknięty" << endl;
18 }
```

Listing 12.4. Definicja metody odczytPlikuTekstowego klasy ZLab12

```
1 void ZLab12::odczytPlikuTekstowego(string nazwaPliku)
2 {
3     ifstream plik;
4     cout << "Otwieram plik." << endl;
5     plik.open(nazwaPliku, ios_base::in);
6
7     int a, b, c;
8
9     if(plik.is_open()){
10        while(!plik.eof()){
11            plik >> a >> b >> c;
12            if(plik.eof()) break;
13            cout << "a="<<a<<" b="<<b
14            <<" c="<<c <<endl;
15        }
16    }
17
18    plik.close();
19 }
```



Listing 12.5. Definicja metody zapisBinarnyDoPliku klasy ZLab12

```

1 void ZLab12::zapisBinarnyDoPliku(string nazwaPliku)
2 {
3     ofstream plik;
4
5     cout << "Otwieram plik." << endl;
6     plik.open(nazwaPliku, ios_base::out|ios_base::binary);
7
8     if(plik.is_open()){
9         cout << "Zapisuje do pliku" << endl;
10        for(int i=1; i<=10; i++)
11        {
12            int k = i;
13            plik.write(reinterpret_cast<char*>(&k),
14                       sizeof (i));
15            k = i*i;
16            plik.write(reinterpret_cast<char*>(&k),
17                       sizeof (i));
18            k = i*i*i;
19            plik.write(reinterpret_cast<char*>(&k),
20                       sizeof (i));
21        }
22    }
23
24    plik.close();
25
26    cout << "Plik zamknięty" << endl;
27 }

```

Listing 12.6. Definicja metody odczytPlikuBinarnego klasy ZLab12

```

1 void ZLab12::odczytPlikuBinarnego(string nazwaPliku)
2 {
3     ifstream plik;
4     cout << "Otwieram plik." << endl;
5     plik.open(nazwaPliku, ios_base::in |ios_base::binary);
6
7     int a, b, c;
8
9     if(plik.is_open()){
10        while(!plik.eof()){
11            plik.read(reinterpret_cast<char*>(&a),
12                      sizeof (a));
13            plik.read(reinterpret_cast<char*>(&b),
14                      sizeof (a));
15            plik.read(reinterpret_cast<char*>(&c),
16                      sizeof (a));
17            if(plik.eof()) break;

```



```
18         cout << "a="<<a<<" b="<<b<<" c="<<c <<endl;
19     }
20 }
21
22     plik.close();
23 }
```

### Dyskusja – odpowiedz na pytania:

- Listing 12.2, wiersz 10 → Co zwraca metoda `str` strumienia ?
- Listing 12.2, wiersz 22 → Dlaczego konieczne jest sprawdzanie warunku ?
- Listing 12.3, wiersz 6 → Co określa `aios_base::out` ?
- Listing 12.4, wiersz 5 → Co określa `ios_base::in` ?
- Listing 12.4, wiersz 10 → Po co jest metoda `eof` ?
- Listing 12.4, wiersz 12 → Dlaczego warunek jest sprawdzany jeszcze raz?
- Listing 12.5, wiersz 6 → Wy tłumacz zapis `ios_base::out|ios_base::binary` .
- Listing 12.5, wiersz 13 → Co robi metoda `write` ?
- Listing 12.5, wiersz 13 → Dlaczego nie można użyć funkcji `static_cast` ?
- Listing 12.6, wiersz 11 → Co robi metoda `read` ?

### Zadanie 12.2. Zapis wyniku symulacji *Wirtualnego Ekosystemu* do pliku tekstowego

W zadaniu zostanie zmieniony plik `main.cpp` z listingu 11.15 tak, aby kolejne kroki symulacji widoczne na ekranie były również zapisywane do pliku tekstowego. Zostanie dodany strumień plikowy `ofstream` i jego obsługa.

Wykonaj polecenia:

- Otwórz plik `main.cpp` i zmień go wg listingu 12.7.
- Uruchom i odszukaj utworzony plik na dysku.

Listing 12.7. Zmieniony plik `main.cpp`

```
1  #include "nisza.h"
2  #include "osobniki.h"
3  #include "sasiedztwo.h"
4  #include <fstream>
5
6  int main()
7  {
8      Srodowisko ekoSystem(8,12);
9
10     ekoSystem.lokuj(new Glon(),0,10);
11     ekoSystem.lokuj(new Glon(),1,10);
12     ekoSystem.lokuj(new Glon(),1,13);
13     ekoSystem.lokuj(new Glon(),3,10);
14     ekoSystem.lokuj(new Grzyb(),1,11);
15     ekoSystem.lokuj(new Grzyb(),0,0);
16     ekoSystem.lokuj(new Bakteria(),3,3);
17     ekoSystem.lokuj(new Bakteria(),2,6);
18 }
```





```

19     ofstream plikWynikowy("symulacja.txt");
20     if(!plikWynikowy.is_open()) return 1;
21     string stanSymulacji;
22
23     unsigned long i = 0;
24
25     do{
26         system("clear");
27         cout << "Krok symulacji: " << i << endl;
28         plikWynikowy << "Krok symulacji: " << i << endl;
29         stanSymulacji = ekoSystem.doTekstu();
30
31         cout << endl << stanSymulacji << endl;
32         plikWynikowy << stanSymulacji << endl;
33         cin.ignore(1);
34         ekoSystem++;
35         i++;
36     } while(i < 200 && !ekoSystem);
37
38     cout << endl;
39
40     plikWynikowy.close();
41     return 0;
42 }

```

### Zadanie 12.3. Odczytanie stanu początkowego symulacji z pliku

W zadaniu zostanie dodana metoda o nazwie `czytajZPliku` do klasy `Srodowisko`, która będzie czytała z pliku początkowy stan zasiedlenia środowiska.

Wykonaj polecenia:

- Otwórz plik `srodowisko.h` i dodaj do definicji klasy deklarację publicznej metody:  
`static Srodowisko czytajZPliku(std::string nazwaPliku)`
- Zdefiniuj metodę w pliku `srodowisko.cpp` wg listingu 12.8.

*Listing 12.8. Definicja metody `czytajZPliku` klasy `Srodowisko`*

```

1  #include <fstream>
2  #include<sstream>
3  #include "osobniki.h"
4
5  Srodowisko Srodowisko::czytajZPliku(std::string nazwaPliku)
6  {
7      std::ifstream plik(nazwaPliku);
8
9      std::stringstream tekst("");

```



```
10     if(plik){
11         tekst << plik.rdbuf();
12         plik.close();
13     }
14
15     std::string zapis = tekst.str();
16
17     unsigned int wiersze=0, kolumny=0;
18     bool pierwszaLinia = true;
19     for(char c : zapis){
20         if(c!='\n'){
21             if(pierwszaLinia && c !=' ') kolumny++;
22         } else{
23             pierwszaLinia = false;
24             if(c=='\n') wiersze++;
25         }
26     }
27
28     Srodowisko srodowisko(wiersze,kolumny);
29
30     char glon = UstawieniaSymulacji
31         ::pobierzUstawienia().znakGlon;
32     char grzyb = UstawieniaSymulacji
33         ::pobierzUstawienia().znakGrzyb;
34     char bakteria = UstawieniaSymulacji
35         ::pobierzUstawienia().znakBakteria;
36     char pusta = UstawieniaSymulacji
37         ::pobierzUstawienia().znakPustaNisza;
38
39     char znak;
40     for(unsigned int w=0; w<wiersze; w++){
41         getline(tekst,zapis);
42         for(unsigned int k=0; k<2*kolumny; k+=2){
43
44             znak = k<zapis.size() ? zapis[k] : pusta;
45
46             if(znak==glon)
47                 srodowisko.lokuj(new Glon(),w,k/2);
48             else if(znak==grzyb)
49                 srodowisko.lokuj(new Grzyb(),w,k/2);
50             else if(znak==bakteria)
51                 srodowisko.lokuj(new Bakteria(),w,k/2);
52         }
53     }
54
55     return srodowisko;
56 }
```



- Utwórz plik tekstowy o nazwie **start.txt** z zawartością identyczną jak na listingu 12.9. (Wklej do notatnika lub podobnego programu.)
- Zapisz plik tekstowy w katalogu bieżącym środowiska *Qt Creator* (to tam gdzie pojawił się plik **symulacja.txt**) lub zmień ścieżkę na listingu 12.10 w wierszu 4.
- Zmień metodę **main** tak, aby czytała ustawienia początkowe środowiska z tego pliku. (Listing 12.10)
- Uruchom i sprawdź czy aplikacja czyta plik **start.txt** i czy zapisuje plik **symulacja.txt**.

Listing 12.9. Zawartość pliku tekstowego **start.txt**

```

- - - - - * *
- - - - - * *
- - - - - @ - - - *
- # - - - - - *
- - - @ - - - - - #
- - - - -
- - - - -
- - - - -

```

Listing 12.10. Zmieniony plik **main.cpp**

```

1  int main() {
2      Srodowisko ekoSystem
3          =Srodowisko::czytajZPliku("start.txt");
4
5      ofstream plikWynikowy("symulacja.txt");
6      if(!plikWynikowy.is_open()) return 1;
7      string stanSymulacji;
8
9      unsigned long i = 0;
10
11     do{
12         system("clear");
13         cout << "Krok symulacji: " << i << endl;
14         plikWynikowy << "Krok symulacji: " << i << endl;
15         stanSymulacji = ekoSystem.doTekstu();
16
17         cout << endl << stanSymulacji << endl;
18         plikWynikowy << stanSymulacji << endl;
19         cin.ignore(1);
20         ekoSystem++;
21         i++;
22     } while(i < 200 && !ekoSystem);
23
24     cout << endl;
25
26     plikWynikowy.close();
27     return 0;
28 }

```



- Zmieniaj parametry symulacji i konfigurację początkową tak, aby wirtualny ekosystem pozostawał jak najdłużej w równowadze.

**Dyskusja – odpowiedz na pytania:**

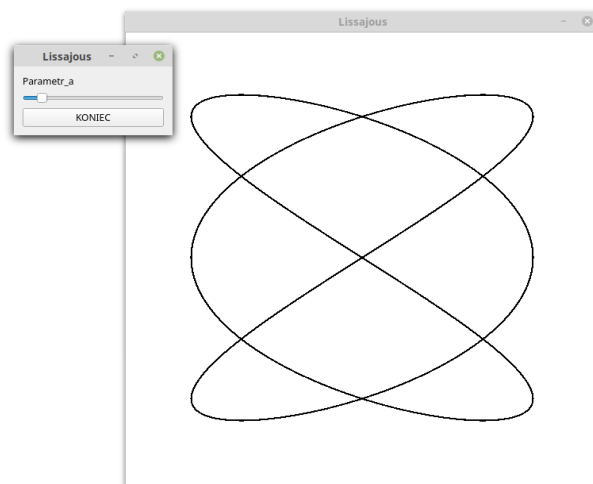
- Listing 12.7 → Jak zachowa się program, jeżeli nie uda się utworzyć pliku?
- Listing 12.8 → Dlaczego metoda `czytajZPliku` jest statyczna? Czy musi taka być?
- Listing 12.8, wiersz 9 → Jaka jest początkowa zawartość strumienia tekstowego?
- Listing 12.8, wiersz 11 → Co tutaj się dzieje ?
- Listing 12.8, wiersze od 17 do 26 → W tym fragmencie ustalone zostaje jaki rozmiar ma środowisko. Odtwórz i opisz ten algorytm?
- Listing 12.8, wiersze od 30 do 37 → Czy to jest konieczne?
- Listing 12.9, wiersze od 39 do 53 → W tym fragmencie dodawane są organizmy do środowiska. Odtwórz i opisz ten algorytm?



## **LABORATORIUM 13. BIBLIOTEKA QT, WYKORZYSTANIE KLAS BIBLIOTEKI DO BUDOWY APLIKACJI "WYKRESY FUNKCJI MATEMATYCZNYCH"**

### **Cel laboratorium:**

Celem laboratorium jest praktyczne wprowadzenie do biblioteki Qt poprzez wykonanie aplikacji z graficzny interfejsem użytkownika. Aplikacja służy do rysowania krzywych Lissajous, jej wygląd został przedstawiony na Rys. 13.1.



*Rys. 13.1. Wygląd aplikacji Lissajous*

### **Zakres tematyczny zajęć:**

- Tworzenie projektu programistycznego korzystającego z biblioteki graficznej Qt.
- Opracowanie modelu matematycznego do rysowania krzywych Lissajous.
- Opracowanie graficznego interfejsu użytkownika złożonego z dwóch oddzielnych okien.

### **Kompendium wiedzy:**

- Wystarczające na potrzeby laboratorium informacje na temat krzywych Lissajous można pozyskać z Wikipedii ([https://pl.wikipedia.org/wiki/Krzywa\\_Lissajous](https://pl.wikipedia.org/wiki/Krzywa_Lissajous)).
- Budowanie aplikacji z interfejsem graficznym polega na zaimportowaniu właściwych obiektów (komponentów) i odpowiednim połączeniu ich ze sobą. Do komponentów gotowych można (a nawet trzeba) dodać własne klasy.
- Praktyką inżynierską jest stosowanie narzędzi ułatwiających budowę graficznego interfejsu użytkownika. W przypadku środowiska Qt jest to program *Qt Designer*. Jednak narzędzia takie nie będą stosowane w trakcie tego laboratorium.
- Projektując aplikacje należy rozdzielić interfejs graficzny od części logicznej (algorytmicznej).



### Pytania kontrolne:

1. Wyjaśnij dlaczego należy rozdzielać interfejs użytkownika od części algorytmicznej aplikacji.

### Zadanie 13.1. Utworzenie projektu korzystającego z biblioteki Qt

Wykonaj polecenia:

- Uruchom środowisko *Qt Creator* .
- Menu Główne → Plik → Nowy plik lub projekt .
- Aplikacja → Qt Widgets Application → Przycisk Wybierz .
- Nazwa → Lissajous .
- Bez generowania formularza: → Generate form → Puste → Przycisk Dalej .
- Wybór narzędzi → Desktop → Przycisk Dalej .
- Przycisk → Zakończ .
- Uruchom i sprawdź czy pokazuje się okno programu.
- Dodaj do projektu zwykłą klasę *Model* bez żadnych dołączeń .
- Dodaj do projektu klasę *Sterowanie* tutaj dołącz *QWidget*.
- Dodaj do projektu klasę *Wyswietlanie* bez żadnych dołączeń.
- Usuń z projektu pliki *mainwindow.h* oraz *mainwindow.cpp* .
- Doprowadź plik *main.cpp* do postaci z listingu 13.1.

Listing 13.1. Wstępna postać pliku *main.cpp*

```
1  #include <QApplication>
2
3  int main(int argc, char *argv[])
4  {
5      QApplication a(argc, argv);
6
7      return a.exec();
8  }
```

### Zadanie 13.2. Utworzenie klasy *Model*

Informacje na temat użytych klas *QVector* *QPointF* oraz *QpolygonF* dostępne są pod adresami:

<https://doc.qt.io/qt-5/qvector.html>

<https://doc.qt.io/qt-5/qpointf.html>

<https://doc.qt.io/qt-5/qpolygonf.html>

Wykonaj polecenia:

- Przenieś kod z listingu 13.2. do pliku *model.h* .
- Przenieś kod z listingu 13.3. do pliku *model.cpp* .



Listing 13.2. Zawartość pliku model.h

```
1  #ifndef MODEL_H
2  #define MODEL_H
3
4  #include <cmath>
5  #include <QVector>
6  #include <QPointF>
7  #include <QPolygonF>
8
9  class Model {
10 private:
11     double a=1, b=2, d = M_PI/2, skala = 100,
12                                     dt = 1/skala;
13     QPointF p;
14     QVector<QPointF> punkty;
15     void wygenerujPunkty();
16
17 public:
18     Model(){ wygenerujPunkty();}
19     void ustawParemery(double _a, double _b, double _d);
20     void ustawSkale(double _skala);
21     double wartosc_a() const {return a;}
22     double wartosc_b() const {return b;}
23     double wartosc_d() const {return d;}
24
25     QPolygonF getQPolygnF() const
26                                     {return QPolygonF(punkty);}
27 };
28
29 #endif // MODEL_H
```

Listing 13.3. Zawartość pliku model.cpp

```
1  #include "model.h"
2
3  void Model::wygenerujPunkty()
4  {
5      double tmax = a < b ? 4*M_PI/a : 4*M_PI/b, t=0;
6
7      punkty.clear();
8      while(t<tmax){
9          p.setX(skala * sin(a*t+d));
10         p.setY(skala * sin(b*t));
11         punkty.push_back(p);
12         t+=dt;
13     }
14 }
15
```



```
16 void Model::ustawParametry(double _a, double _b, double _d)
17 {
18     a = _a;
19     b = _b;
20     d = _d;
21     wygenerujPunkty();
22 }
23
24 void Model::ustawSkale(double _skala)
25 {
26     if(skala > 0){
27         skala = _skala;
28         dt = 1.0/skala;
29         wygenerujPunkty();
30     }
31 }
```

### Zadanie 13.3. Utworzenie klasy Sterowanie

Informacje na temat użytych klas QWidget QPushButton QSlider QLabel oraz QVBoxLayout dostępne są pod adresami:

<https://doc.qt.io/qt-5/qwidget.html>

<https://doc.qt.io/qt-5/qpushbutton.html>

<https://doc.qt.io/qt-5/qslider.html>

<https://doc.qt.io/qt-5/qlabel.html>

<https://doc.qt.io/qt-5/qvboxlayout.html>

Wykonaj polecenia

- Przenieś kod z listingu 13.4. do pliku `sterowanie.h`.
- Przenieś kod z listingu 13.5. do pliku `sterowanie.cpp`.

Listing 13.4. Zawartość pliku `sterowanie.h`

```
1  #ifndef STEROWANIE_H
2  #define STEROWANIE_H
3  #include <QWidget>
4  #include <QPushButton>
5  #include <QSlider>
6  #include <QLabel>
7  #include <QVBoxLayout>
8
9  class Sterowanie : public QWidget{
10     Q_OBJECT
11
12 private:
13     QLabel * napis_a;
14     QSlider * slider_a;
15     QPushButton * przyciskKoniec;
16     QVBoxLayout * ukladacz;
17 }
```





```
18 public:
19     Sterowanie(QWidget * parent = nullptr);
20     const QPushButton * przycisk()
21         { return przyciskKoniec; }
22     const QSlider * slidera() { return slider_a;}
23 };
24 #endif // STEROWANIE_H
```

Listing 13.5. Zawartość pliku sterowanie.cpp

```
1  #include "sterowanie.h"
2
3  Sterowanie::Sterowanie(QWidget *parent)
4      :QWidget(parent)
5  {
6      przyciskKoniec = new QPushButton("KONIEC",this);
7      napis_a = new QLabel("Parametr_a",this);
8      slider_a = new QSlider(Qt::Horizontal, this);
9
10     slider_a->setRange (1, 20);
11     slider_a->setValue(1);
12
13     ukladacz = new QVBoxLayout();
14
15     ukladacz->addWidget(napis_a);
16     ukladacz->addWidget(slider_a);
17     ukladacz->addWidget(przyciskKoniec);
18
19     setLayout(ukladacz);
20 }
```

#### Zadanie 13.4. Utworzenie klasy Wyświetlanie

Informacje na temat użytych klas `QGraphicsPolygonItem` `QGraphicsScene` oraz `QGraphicsView` dostępne są pod adresami:

<https://doc.qt.io/archives/qt-4.8/qgraphicspolygonitem.html>

<https://doc.qt.io/qt-5/qgraphicsscene.html>

<https://doc.qt.io/qt-5/qgraphicsview.html>

Wykonaj polecenia:

- Przenieś kod z listingu 13.6. do pliku `wyswietlanie.h`.
- Przenieś kod z listingu 13.7. do pliku `wyswietlanie.cpp`.



*Listing 13.6. Zawartość pliku sterowanie.h*

```
1  #ifndef WYSWIETLANIE_H
2  #define WYSWIETLANIE_H
3
4  #include "model.h"
5
6  #include <QPointF>
7  #include <QPolygonF>
8  #include <QGraphicsPolygonItem>
9  #include <QGraphicsScene>
10 #include <QGraphicsView>
11
12 class Wyświetlanie : public QGraphicsView
13 {
14     Q_OBJECT
15
16 private:
17     Model * model;
18
19     QPolygonF * poligon;
20     QGraphicsPolygonItem * polItem;
21     QGraphicsScene * scene;
22
23 public:
24     Wyświetlanie(QWidget * parent = nullptr);
25     ~Wyświetlanie();
26     void podlaczModel(Model * _model) {model = _model;}
27     void show();
28
29 public slots:
30     void przerysuj(int a);
31 };
32
33 #endif // WYSWIETLANIE_H
```



Listing 13.7. Zawartość pliku sterowanie.cpp

```
1  #include "wyswietlanie.h"
2
3  Wyswietlanie::Wyswietlanie(QWidget * parent)
4      :QGraphicsView(parent)
5  {
6
7      poligon = new QPolygonF();
8      polItem = new QGraphicsPolygonItem();
9      scene = new QGraphicsScene();
10
11      scene->addItem(polItem);
12      setScene(scene);
13      setFixedSize(600,600);
14      fitInView(0, 0, 220, 220, Qt::KeepAspectRatio);
15  }
16
17  Wyswietlanie::~~Wyswietlanie()
18  {
19      delete poligon;
20      delete polItem;
21      delete scene;
22  }
23
24  void Wyswietlanie::show()
25  {
26      przerysuj(0);
27      QGraphicsView::show();
28  }
29
30  void Wyswietlanie::przerysuj(int a)
31  {
32      double aa = model->wartosc_a();
33      double bb = model->wartosc_b();
34      double dd = model->wartosc_d();
35
36      if(a>0) aa = static_cast<double>(a);
37
38      model->ustawParametry(aa, bb, dd);
39      *poligon=model->getQPolygonF();
40      polItem->setPolygon(*poligon);
41  }
```



### Zadanie 13.5. Połączenie komponentów aplikacji

Połączenie stworzonych komponentów następuje w funkcji `main` w pliku `main.cpp`.

Wykonaj polecenia

- Przenieś kod z listingu 13.8. do pliku `main.cpp`.
- Uruchom aplikację.

Listing 13.8. Zawartość pliku `main.cpp`

```
1  #include <QApplication>
2  #include "model.h"
3  #include "sterowanie.h"
4  #include "wyswietlanie.h"
5
6  int main(int argc, char *argv[])
7  {
8      QApplication app(argc, argv);
9
10     Model model;
11     Sterowanie sterowanie;
12     Wyswietlanie wyswietlanie;
13
14     //Podłączanie obiektów
15
16     wyswietlanie.podlaczModel(&model);
17
18
19     QObject::connect (sterowanie.przycisk(), //źródło
20                      SIGNAL (clicked()), //sygnał źródła
21                      &app, //odbiorca
22                      SLOT(quit ()) //akcja odbiorcy
23                      );
24
25     QObject::connect (sterowanie.slidera(), //źródło
26                      SIGNAL (valueChanged (int)), //sygnał źródła
27                      &wyswietlanie, //odbiorca
28                      SLOT(przerysuj(int)) //akcja odbiorcy
29                      );
30
31     wyswietlanie.show();
32     sterowanie.show();
33
34     return app.exec();
35 }
```



## **LABORATORIUM 14. REPETYTORIUM**

### **Cel laboratorium:**

Celem zajęć jest dyskusja analityczna pod kierunkiem prowadzącego zajęcia na temat wytworzonej w trakcie zajęć aplikacji *Wirtualny Ekosystem* ze szczególnym naciskiem na obiektywne techniki programowania.

### **Zakres tematyczny zajęć:**

- Laboratoria od 2 do 12.

### **Pytania kontrolne:**

1. Wszystkie pytania i problemy postawione w trakcie cyklu zajęć laboratoryjnych w sekcjach *Dyskusja* i *Pytania kontrolne*.

### **Zadania\***

Zajęcia mogą mieć również formę samodzielnej pracy studentów nad zaproponowanymi problemami:

- Wizualizacja graficzna aplikacji *Wirtualny Ekosystem* z wykorzystaniem biblioteki Qt.
- Rozbudowanie aplikacji o możliwość czytania z pliku ustawień symulacji, a nie tylko konfiguracji początkowej.
- Rozszerzenie aplikacji o nowe zachowania wirtualnych organizmów.



## **LABORATORIUM 15. KOŁOKWIUM 2**

### **Cel laboratorium:**

Weryfikacja nabytych umiejętności programowania z wykorzystaniem technik obiektowych w języku C++.

### **Zakres tematyczny kolokwium:**

- Laboratoria od 2 do 12.

### **Pytania kontrolne:**

- 1 . Warunkiem przystąpienia do kolokwium jest wykonanie wszystkich zadań z zakresu laboratoriów od 1 do 12 i wysłanie kopii prowadzącemu 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



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny

