

# Informatyka II dla kierunku Mechatronika (sem. 4)

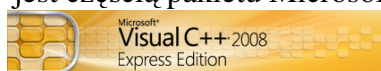
## Instrukcja do ćwiczeń laboratoryjnych – Visual C++

Opracowanie: dr inż. Marek Galewski

Uwaga: Niniejsze materiały przeznaczone są dla studentów jako pomoc do zajęć dydaktycznych prowadzonych przez pracowników i doktorantów Katedry Mechaniki i Mechatroniki Wydziału Mechanicznego Politechniki Gdańskiej. Jakikolwiek ich wykorzystywanie przez osoby trzecie do celów naukowych, dydaktycznych oraz komercyjnych jest zabronione z mocy Ustawy o prawie autorskim i prawach pokrewnych.

### 1 Wstęp

Podczas zajęć laboratoryjnych wykorzystywane jest środowisko programowania Microsoft Visual C++ 2008 Express, który jest częścią pakietu Microsoft Visual Studio 2008.



Prace należy zapisywać w folderze c:\Prace\XYZ\ gdzie za XYZ wybrać dowolną nazwę własnego programu. Programy umieszczone w innych folderach będą usuwane! Po zakończeniu poszczególnych zajęć zalecane jest kopiowanie wykonanych programów – nie ma gwarancji, że ktoś innych ich nie skasuje lub zmodyfikuje. Wykonane ćwiczenia będą przydatne przy pisaniu programów zaliczających zajęcia laboratoryjne.

Dodatkowo, w kilku ćwiczeniach wykorzystany zostanie System Zarządzania Bazą Danych Microsoft SQL Server 2008 R2 Express.



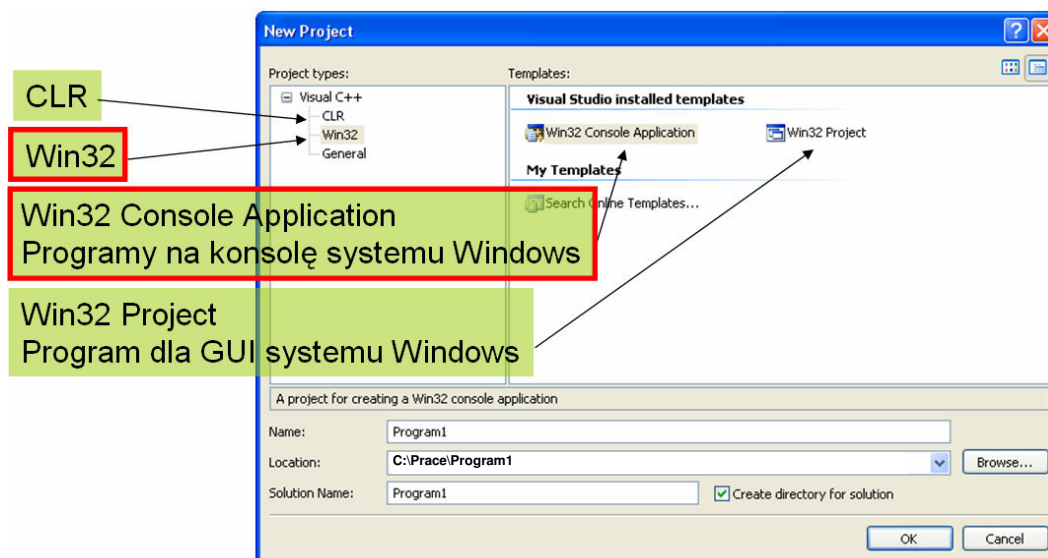
Do wykonania ćwiczeń mogą być przydatne materiały z wykładów (prezentacje 2, 3, 4, 5 i 7).

### 2 Aplikacje dla konsoli systemu MS Windows – C++

#### 2.1 Tworzenie nowego projektu

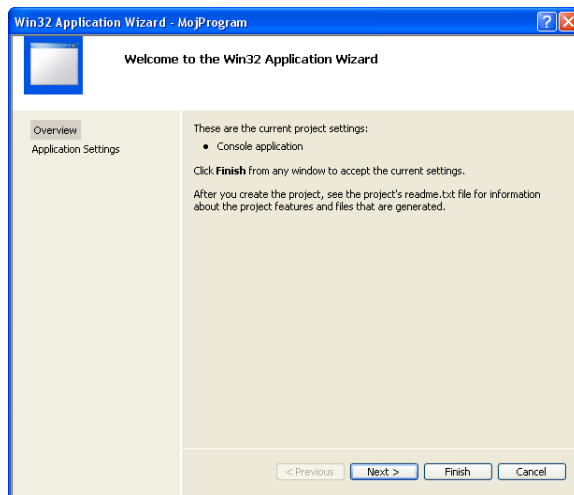
Z Menu wybierz File > New > Project...

W oknie kreatora projektu wybierz „Win32” > „Win32 Console Application”, a następnie podaj nazwę projektu (Rys. 1)

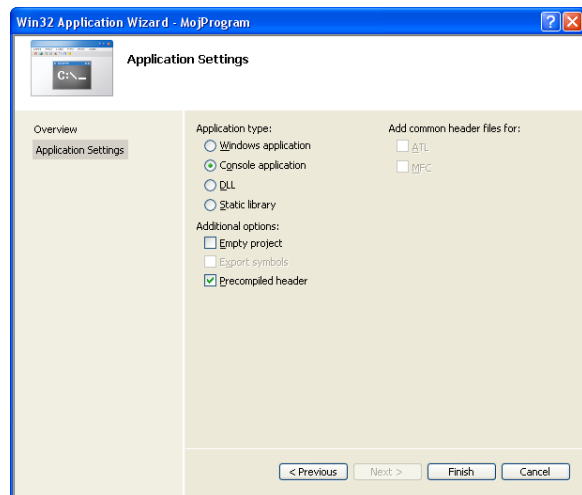


Rys. 1

W kolejnych oknach pozostaw opcje bez zmian (Rys. 2 i Rys. 3)

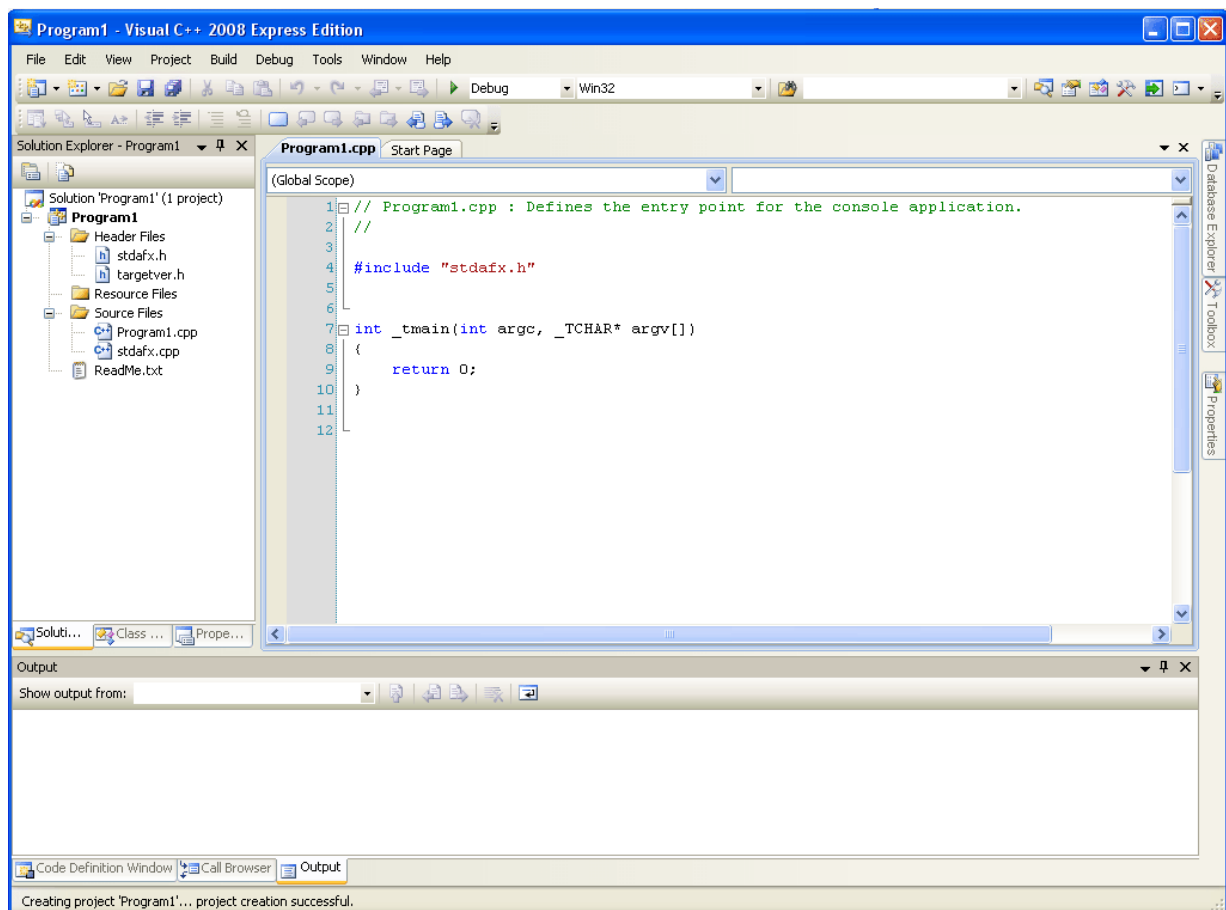


Rys. 2



Rys. 3

Po zakończeniu pracy kreatora tworzony jest pierwszy, „pusty” program (Rys. 4)



Rys. 4

Uruchomienie programu następuje po wykonaniu jednego z poniższych:

- Kliknięciu zielonej strzałki na pasku narzędzi
- Wciśnięciu klawisza F5
- Wybranie z menu *Debug > Start Debugging*

## 2.2 Ćwiczenie 1 – Pierwszy program

### 2.2.1 Ćwiczenie 1a

Zmodyfikuj program według podanego wzorca tak, by wyświetlał komunikat „Program działa” i oczekiwał na wciśnięcie klawisza. Oczekiwanie można zrealizować z użyciem funkcji `_getch()`.

---

```
#include "stdafx.h"
#include <conio.h>

int _tmain(int argc, _TCHAR* argv[])
{
    printf("Program dziala!");
    _getch();
    return 0;
}
```

---

Uruchom napisany program.

### 2.2.2 Ćwiczenie 1b

Zmodyfikuj program według podanego wzorca tak, by podawał sinus zadanego w stopniach kąta.

---

```
#include "stdafx.h"
#include <conio.h>
#include <math.h>

const double pi=3.14159265;

int _tmain(int argc, _TCHAR* argv[])
{
    double wynik, kat_stopnie;    //definicje zmiennych

    kat_stopnie=45;
    wynik=sin(kat_stopnie*pi/180);
    printf("sin %5.2f = %5.3f\n\\0",kat_stopnie,wynik);
    _getch();
    return 0;
}
```

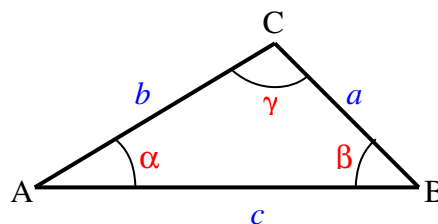
---

Uruchom napisany program.

Sprawdź jego działanie dla innych kątów np.  $0^\circ$ ,  $180^\circ$ ,  $30^\circ$ ,  $60^\circ$ ,  $90^\circ$ ,  $33^\circ$ .

### 2.2.3 Ćwiczenie 1c

Zmodyfikuj program tak, by na podstawie długości boków  $a$  i  $b$  trójkąta oraz kąta  $\gamma$  między nimi (podanego w stopniach), program obliczał długość boku  $c$  oraz pole powierzchni  $S$  trójkąta. Przyjmij oznaczenia jak na Rys. 5. Wynik obliczeń ma być pokazany na ekranie.



Rys. 5

$$S = \frac{ab \sin(\gamma)}{2}$$

$$c^2 = a^2 + b^2 - 2ab \cos(\gamma) \quad \text{tw. Carnota, tw. cosinusów}$$

W programie potrzebne będą następujące zmienne:

- a, b, c – długości boków trójkąta
  - gamma\_stopnie, gamma\_rad – kąt gamma wyrażony w stopniach i radianach
  - S – pole powierzchni trójkąta
- 

```
#include "stdafx.h"
#include <conio.h>
#include <math.h>

const double pi=3.14159265;

int _tmain(int argc, _TCHAR* argv[])
{
    //definicje zmiennych
    double a, b ,c;
    double gamma_stopnie, gamma_rad;
    double S;

    //wartosci zmiennych
    a=5;
    b=10;
    gamma_stopnie=90;

    //obliczenia
    gamma_rad=gamma_stopnie*pi/180;
    c=sqrt(a*a+b*b-2*a*b*cos(gamma_rad));
    S=a*b*sin(gamma_rad)/2;

    //prezentacja wynikow
    printf("Trojkat o bokach a=%5.3f, b= %5.3f →
           → i kacie gamma=%5.3fst\n\0", a, b, gamma_stopnie);
    printf("ma bok c=%5.3f i powierzchnie S=%5.3f", c, S);

    _getch();
    return 0;
}
```

---

Po uruchomieniu i przetestowaniu programu zamień część związaną z wyświetlaniem wyników tak, by wykorzystywała strumienie. W tym celu na początku programu należy dołączyć bibliotekę obsługi strumieni

---

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

---

Następnie, w dalszej części zamiast funkcji *printf()* użyć strumieni:

---

```
cout<<"Trojkat o bokach a="<<a<<" b="<<b;
cout<<" i kacie gamma="<<gamma_stopnie<<"st"<<endl;
cout<<"ma bok c="<<c<<" i powierzchnie S="<<S<<endl;
```

---

Po uruchomieniu i przetestowaniu programu zmodyfikuj go tak, by wartości danych wejściowych były podawane przez użytkownika (tzn. w czasie działania programu, a nie wpisane na stałe w kodzie źródłowym).

---

```
cout<<"podaj a=";
cin>>a;
cout<<"podaj b=";
cin>>b;
```

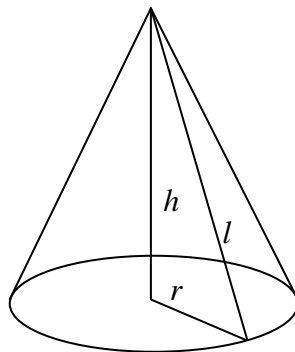
```
cout<<"podaj kat gamma=";
cin>>gamma_stopnie;
```

Uruchom napisany program.

## 2.3 Ćwiczenie 2 – Proste obliczenia i komunikacja z użytkownikiem

Korzystając z doświadczeń z ćwiczenia 1 utwórz nowy projekt i napisz program obliczający na podstawie długości tworzącej  $l$  i wysokości  $h$  stożka:

- Pole powierzchni stożka  $P$
- Objętość stożka  $V$



Rys. 6

Przydatne wzory:

Pole podstawy:  $P_p = \pi \cdot r^2$

Pole boku:  $P_b = \pi \cdot r \cdot l$

Objętość:  $V = \frac{1}{3} \cdot P_p \cdot h$

Dane powinny być wprowadzane przez użytkownika podczas działania programu.

Uwaga: potrzebne jest wprowadzenie tylko dwóch (a nie trzech) danych wejściowych!

## 2.4 Ćwiczenie 3 - Funkcje

### 2.4.1 Ćwiczenie 3a

Utwórz nowy projekt i napisz program obliczający wynik wyrażenia

$$w = \frac{\sqrt{2 \cdot 5}}{2 \cdot 0,6} + \frac{\sqrt{3 \cdot 8}}{3 \cdot 0,6} + \frac{\sqrt{8 \cdot 11}}{8 \cdot 0,6} + \frac{\sqrt{10 \cdot 50}}{10 \cdot 0,6}.$$

Zauważ, że wzór ogólny opisujący poszczególne wyrazy to  $\frac{\sqrt{a \cdot b}}{a \cdot 0,6}$ . W celu obliczenia

wartości  $w$  napisz najpierw funkcję obliczającą wartości wyrazów, korzystającą ze wzoru ogólnego. Funkcja ta powinna przyjmować dwa parametry:  $a$  i  $b$  oraz zwracać wynik obliczeń dla pojedynczego składnika sumy. Jej deklaracja przyjmie postać:

```
float element(float a, float b)
```

Obliczenie sumy będzie polegało na zsumowaniu wyników uzyskanych z kilku wywołań funkcji `element( , )`;

Gotowy program powinien wyglądać tak jak poniżej:

```
#include "stdafx.h"
#include <math.h>
#include <conio.h>
#include <iostream>
using namespace std;
```

```
float element(float a, float b)
{
    //funkcja obliczajaca jeden element sumy
    float e;

    e=sqrt(a*b)/(a*0.6);
    return e;
}

float oblicz_wzor()
{
    float w;
    //obliczenie sumy elementow
    w=element(2,5)+element(3,8)+element(8,11)+element(10,50);
    return w;
}

int _tmain(int argc, _TCHAR* argv[])
{
    float wynik;
    wynik=oblicz_wzor();
    cout<<"wynik sumy ="<<wynik<<endl;
    _getch();
    return 0;
}
```

---

Uruchom i sprawdź działanie programu.

Uzyskany wynik sumy  $w=11.038$ .

### 2.4.2 Ćwiczenie 3b

Zmodyfikuj program tak, by obliczał sumę poniższego wyrażenia:

$$w = \frac{1 \cdot x^2 \cdot \sin\left(\frac{\pi}{|y|+1}\right)}{\sqrt{|y|+0,5}} + \frac{4 \cdot x^2 \cdot \sin\left(\frac{\pi}{|y|+1}\right)}{\sqrt{|y|+2}} + \frac{5 \cdot x^2 \cdot \sin\left(\frac{\pi}{|y|+1}\right)}{\sqrt{|y|+2,5}} + \frac{10 \cdot x^2 \cdot \sin\left(\frac{\pi}{|y|+1}\right)}{\sqrt{|y|+5}}.$$

Wartości  $x$  i  $y$  mają być zadawane przez użytkownika. Do obliczenia poszczególnych składowych sumy użyj funkcji wywoływanej z odpowiednimi parametrami. Pobranie tej wartości powinno odbywać się na początku funkcji `_tmain()`. Wartości te powinny być przekazywane do funkcji obliczającej wzór, co oznacza, że deklaracja tej funkcji musi przyjąć postać:

---

```
float oblicz_wzor(float x, float y);
```

---

Następnie, wartości  $x$  i  $y$  muszą, obok innych potrzebnych wartości, być przekazane do funkcji `element()`. Jej deklaracja także będzie wymagała zmian. Wprowadź je i sprawdź działanie programu.

Przykładowe wyniki:

dla  $x = 5$  i  $y = 5 \rightarrow w = 86,5785$   
dla  $x = -5$  i  $y = -5 \rightarrow w = 86,5785$   
dla  $x = 10$  i  $y = 5 \rightarrow w = 346,314$   
dla  $x = 20$  i  $y = 10 \rightarrow w = 615,249$   
dla  $x = -5$  i  $y = -10 \rightarrow w = 38,4530$

### 2.4.3 Ćwiczenie 3c

Zmień kolejność funkcji w kodzie programu 3b tak, by pierwszą funkcją była funkcja `_tmian()`. Pamiętaj o umieszczeniu wcześniej, przed funkcją `_tmian()` prototypów funkcji `element()` i `oblicz_wzor()`, dzięki którym kompilator będzie wiedział, że funkcje będą zdefiniowane w dalszej części kodu programu:

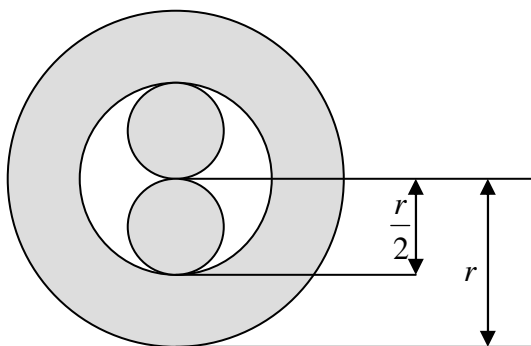
```
float element(float m, float x, float y);
float oblicz_wzor(float x, float y);
```

Uruchom napisany program.

## 2.5 Ćwiczenie 4 – Wykorzystanie funkcji

### 2.5.1 Ćwiczenie 4a

Utwórz nowy projekt i napisz program obliczający pole powierzchni szarej części figury pokazanej na Rys. 7.



Rys. 7

**Napisz funkcje** `p_kola()` i wykorzystaj ją do obliczania pól elementów składowych wywołując ją kilkakrotnie z odpowiednimi parametrami.

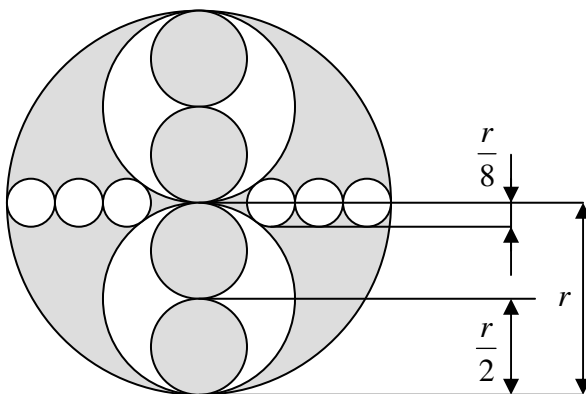
Przykładowe wyniki:

dla  $r = 10$  – pole = 274.889

dla  $r = 20$  – pole = 1099.56

### 2.5.2 Ćwiczenie 4b

Zmodyfikuj program tak, by dodatkowo obliczał pole powierzchni szarej części figury pokazanej na Rys. 8.



Rys. 8

Przykładowe wyniki:

dla  $r = 10$  – pole = 206.167

dla  $r = 20$  – pole = 824.668

## 2.6 Ćwiczenie 5 – Instrukcja warunkowa *if...else*

Utwórz nowy projekt i napisz program, który na podstawie wartości współczynników  $a$ ,  $b$  i  $c$  określi ile pierwiastków rzeczywistych ma równanie  $ax^2 + bx + c = 0$  oraz poda ich wartości.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    float a,b,c;    //wspolczynniki rownania
    float delta;
    float x1,x2;    //pierwiastki rownania

    cout<<"Podaj wspolczynniki rownania a*x^2+b*x+c=0"<<endl;;
    cout<<"Podaj a= ";
    cin>>a;
    cout<<"Podaj b= ";
    cin>>b;
    cout<<"Podaj c= ";
    cin>>c;

    if (a!=0){        //kontrola, czy rowanie jest kadratowe
        delta=pow(b,2)-4*a*c;

        if (delta>0){
            x1=(-b+sqrt(delta))/(2*a);
            x2=(-b-sqrt(delta))/(2*a);
            cout<<"Rownanie ma 2 pierwiastki"<<endl;
            cout<<"x1= "<<x1<<endl;
            cout<<"x2= "<<x2<<endl;
        }
        if (delta==0){
            x1=(-b+sqrt(delta))/(2*a);
            cout<<"Rownanie ma 1 pierwiastek"<<endl;
            cout<<"x1= "<<x1<<endl;
        }
        if (delta<0){
            cout<<"Rownanie nie ma pierwiastkow rzeczywistych"<<endl;
        }
    }
    else {
        cout<<"Rownanie ma a=0, nie jest rownaniem kwadratowym"<<endl;
    }

    _getch();
    return 0;
}
```

---

Uruchom napisany program.

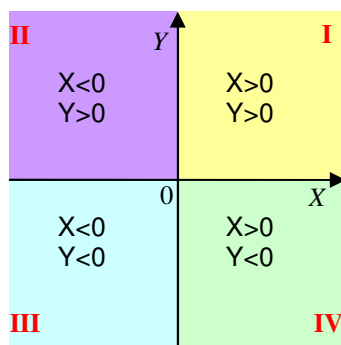


## 2.7 Ćwiczenie 6 – Instrukcja warunkowa *if...else*

Utwórz nowy projekt i napisz program, który na podstawie współrzędnych  $X$  i  $Y$  punktu określi, w której ćwiartce układu współrzędnych jest on położony (Rys. 9). Jeśli punkt leży na osi lub w środku układu współrzędnych - program powinien o tym poinformować.

Użyj konstrukcji *if...else....*. Konieczne będzie jej kilkukrotne zagnieżdżenie.

Aby połączyć dwa warunki, które mają być spełnione jednocześnie użyj operatora iloczynu logicznego  $\&\&$ , np.: `if ((x>0) && (y>0))`. W przypadku, gdy wystarczy spełnienie jednego z kilku warunków, użyj operatora sumy logicznej  $\|$  np.: `if ((x==0) || (y==0))`.



Rys. 9

## 2.8 Ćwiczenie 7 – Instrukcja warunkowa *if...else*

Utwórz nowy projekt i napisz program, który na podstawie długości trzech odcinków (boków) sprawdzi:

Wersja rozgrzewkowa a) Czy da się z nich zbudować trójkąt?

Wersja podstawowa b) Czy da się z nich zbudować trójkąt? Jeśli jest możliwe zbudowanie trójkąta – określi czy jest to trójkąt równoboczny, równoramienny czy zwykły. Dodatkowo sprawdzi, czy trójkąt jest prostokątny

Wersja rozszerzona c) – jak wersja b, ale dodatkowo sprawdzi, czy trójkąt może być ostro- lub rozwartokątny

### Uwagi:

- Kolejność wpisywania boków przez użytkownika może być dowolna!
- Pamiętaj, że trójkąt może mieć kilka cech jednocześnie, np. równoramienny i prostokątny.

Uruchom napisany program i sprawdź poprawność jego działania.

Przykłady wywołań i wyników:

Podaj dlugosc boku a= 1 Podaj dlugosc boku b= 1 Podaj dlugosc boku c= 2 Z podanych bokow nie da sie zbudowac trojkata.	Podaj dlugosc boku a= -1 Podaj dlugosc boku b= 9 Podaj dlugosc boku c= 2 Z podanych bokow nie da sie zbudowac trojkata.
Podaj dlugosc boku a= 3 Podaj dlugosc boku b= 4 Podaj dlugosc boku c= 5 Z podanych bokow da sie zbudowac trojkat prostokatny.	Podaj dlugosc boku a= 51 Podaj dlugosc boku b= 100 Podaj dlugosc boku c= 51 Z podanych bokow da sie zbudowac trojkat rownoramienny, rozwartokatny.
Podaj dlugosc boku a= 6 Podaj dlugosc boku b= 6 Podaj dlugosc boku c= 6 Z podanych bokow da sie zbudowac trojkat rownoboczny, ostrokątny.	Podaj dlugosc boku a= 23 Podaj dlugosc boku b= 32 Podaj dlugosc boku c= 11 Z podanych bokow da sie zbudowac trojkat rozwartokatny.

## 2.9 Ćwiczenie 8 – Instrukcja warunkowa *switch*

Utwórz nowy projekt i napisz program, który na podstawie numeru dnia poda jego nazwę. Dla 6 i 7 zamiast nazwy dnia napisze „Weekend”

---

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    unsigned short int nr_dnia;

    cout<<"podaj numer dnia tygodnia: ";
    cin>>nr_dnia;
    switch (nr_dnia){
        case 1: {cout<<"Poniedzialek"<<endl;} break;
        case 2: {cout<<"Wtorek"<<endl;} break;
        case 3: {cout<<"Sroda"<<endl;} break;
        case 4: {cout<<"Czwartek"<<endl;} break;
        case 5: {cout<<"Piatek"<<endl;} break;
        case 6:
        case 7: {cout<<"Weekend"<<endl;} break;
        default: {cout<<"Tydzien ma 7 dni"<<endl;} break;
    }

    _getch();
    return 0;
}
```

---

Uruchom napisany program.

## 2.10 Ćwiczenie 9 – Instrukcja warunkowa *switch*

Korzystając z doświadczeń z poprzedniego ćwiczenia, napisz program, który zapyta użytkownika o dwie liczby, a następnie o znak operacji (+,-,\*,/) i na tej podstawie wykonywa odpowiednie działanie. Użyj konstrukcji *switch*. Dla dzielenia, program powinien przed wykonaniem dzielenia sprawdzić (instrukcja *if*), czy  $y \neq 0$ . Jeśli  $y == 0$ , powinien zostać wyświetlony odpowiedni komunikat.

Do pobrania znaku operacji potrzeba będzie zmienna typu *char*

---

```
char znak_op;
```

---

Można jej użyć bezpośrednio jako warunku instrukcji *switch*, wówczas wybór przypadku *case* może mieć np. postać:

---

```
case '+':
```

---

Uruchom napisany program.

## 2.11 Ćwiczenie 10 – Instrukcje warunkowe

Napisz program, który wczyta liczby całkowite  $B$ ,  $W$ ,  $Z$ , a następnie poprawnie napisze tekst: „Na łące [pasł / pasły / pasło] się  $B$  [baran / barany / baranów]. Wieczorem [przyszedł / przyszedł / przyszedł]  $W$  [wilk / wilki / wilków] i [zjadł / zjadł] [barana /  $Z$  barany /  $Z$  baranów]. Rano na łące [nie było / był / były / było już tylko  $B-Z$ ] [baran / barany / baranów].”

Rozwiązanie powinno być tak skonstruowane, że wpisać można dowolne liczby naturalne – należy znaleźć ogólne zasady dopasowania formy rzeczowników i czasowników do podanych liczb. Program powinien także zawierać kontrolę danych:  $B \geq 1$ ,  $W \geq 1$ ,  $Z \geq 0$ ,  $B \geq Z$ .

Zadanie można wykonać zarówno korzystając z instrukcji *if* jak i *switch*. Przydatny będzie także operator modulo  $rd = x \% y$  wyznaczający resztę  $rd$  z dzielenia liczby  $x$  przez  $y$ . Tę samą

operację wykonać można także z użyciem funkcji  $rd=fmod(x, y)$ . Funkcja ta znajduje się w bibliotece `math.h`, którą należy dołączyć do programu (polecenie `#include`). Konieczne może być także rzutowanie typów zmiennych na typ całkowity `static_cast <int>( )` lub zmiennoprzecinkowy `static_cast <double>( )`.

## 2.12 Ćwiczenie 11 – Instrukcje warunkowe, prosty szkielet programu

### 2.12.1 Ćwiczenie 11a

Utwórz nowy projekt i napisz program, który będzie wyświetlał kody wciśniętych klawiszy. Program ma działać tak długo, aż użytkownik wciśnie klawisz ESC.

**Uwagi:**

#### Obsługa klawiszy

- Każdy klawisz / znak na klawiaturze ma przypisany kod.
- Kod należy odczytywać funkcją `_getch()`.
- Kody dla małej i wielkiej litery są różne.
- Część klawiszy (np. klawisze kursora, funkcyjne, PgUp, PgDown itp.) ma podwójne kody, tzn. pierwszy kod jest == 0 lub 224, a po nim następuje drugi kod. Dopiero na podstawie pary kodów można określić, który klawisz został wciśnięty. Oznacza to, że jeżeli pierwszy odczytany kod == 0 lub 224 należy dodatkowo odczytać kod jeszcze raz i na tej podstawie określić wciśnięty klawisz.
- Przy naciskaniu „zwykłych” klawiszy, sterownik klawiatury, umieszcza w buforze klawiatury ich kody. W przypadku klawiszy „specjalnych” umieszczane są 2 kody. Oba muszą być odczytane. Jeśli w programie nie znajdzie się fragment kodu odpowiedzialny za odczytywanie kodów takich klawiszy (nawet gdy nie są potrzebne do obsługi danego programu) wówczas po odczytaniu pierwszego kodu drugi kod nadal pozostanie w buforze klawiatury i w kolejnej pętli programu zostanie zinterpretowany jako zupełnie inny, „zwykły” klawisz.
- Do przechowywania kodów w pamięci należy użyć zmiennej typu `wchar_t`. jest to odpowiednik typu `char` dla znaków kodowanych w kodzie Unicode.

#### Działanie programu

- Program ma działać tak długo, aż użytkownik wciśnie ESC. Oznacza to, że czytanie klawiszy i wyświetlanie ich kodu musi odbywać się w pętli. W tym przypadku najlepiej użyć pętli `do...while`, która działa tak samo jak pętla `while`, z tą różnicą, że warunek trwania pętli sprawdzany jest na końcu każdej iteracji, a nie na początku. Powoduje to, że pętla wykonywana jest zawsze co najmniej jeden raz.

Gotowy kod programu znajduje się poniżej. Czyta i wyświetla on kody klawiszy, dodatkowo w przypadku wciśnięcia A, B, F1 lub F2 wyświetla odpowiedni napis.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    wchar_t klawisz1, klawisz2;

    cout<<"Wciskaj klawisze; koniec = ESC"<<endl;
    do {

        cout<<endl;
        klawisz1=_getch();           //pobranie kodu klawisza
        cout<<"kod klawisza = "<<klawisz1;
```

```

if ((klawisz1==0)|| (klawisz1==224)) {
    klawisz2=_getch();           //pobranie drugiego kodu klawisza
    cout<<"", "<<klawisz2;
}
cout<<endl;

switch (klawisz1){               //kod wcisnietego klawisza
    case 65:                     //a lub A
    case 97: {
        cout<<"A"<<endl;
        } break;

    case 66:                     //b lub B
    case 98: {
        cout<<"B"<<endl;
        } break;

    case 27: break;             //ESC
    case 0: {                   //klawisze specjalne
        switch (klawisz2){
            case 59: cout<<"F1"<<endl; break; //F1
            case 60: cout<<"F2"<<endl; break; //F2
        } break;
    }
    case 224: break;            //klawisze specjalne
    default: cout<<endl; break;
}
} while (klawisz1!=27);         //powtarzaj gdy klawisz nie jest ESC

return 0;
}

```

Uruchom napisany program.

### 2.12.2 Ćwiczenie 11b

Zmodyfikuj program tak, by wyświetlał wprowadzane samogłoski jako wielkie litery (a nie ich kody), a w miejsce spółgłosek i innych znaków wstawiał znak „\_”.

**Uwagi:**

- Przy modyfikacji ćw. 11a będzie trzeba usunąć część programu
- Kody samogłosek:

	Wielka litera	Mała litera
A, a	65	97
E, e	69	101
I, i	73	105
O, o	79	111
U, u	85	117
Y, y	89	121

- Do zamiany kodu małej litery na wielką służy funkcja *toupper()*
- Aby wyświetlić literę, a nie jej kod, należy skorzystać z rzutowania typów `static_cast <char>( )`.

Uruchom napisany program.

Przykładowy wynik działania programu dla tekstu „Alicja ma białego kota”

```

Napisz tekst; koniec = ESC
A_I_A_A_IA_E_O_O_A

```

Sprawdź działanie programu także dla przypadku, gdy wciskane są klawisze specjalne, w szczególności: klawisze kursora, F4, F5, F7. Ich naciśnięcie również powinno generować na ekranie pojedynczy znak „\_”.

### 2.13 Ćwiczenie 12 – Pętla *for*

Utwórz nowy projekt i napisz program odliczający czas do „autodestrukcji”.

#### Uwagi:

- Wykorzystana zostanie pętla *for*, która pozwala wykonać blok instrukcji zadaną liczbę razy. W momencie rozpoczęcia pętli liczba powtórzeń musi być znana. W naszym przypadku warunek ten jest spełniony, gdyż wcześniej użytkownik podaje na ile sekund „ustawić zapalnik”.
- Funkcja *sleep(czas)* z biblioteki *windows.h* służy do odczekania określonego w milisekundach czasu.
- Funkcja *beep(czestotliwosc, czas)* generuje dźwięk o określonej częstotliwości i czasie trwania (uwaga: nie na wszystkich komputerach działa / generuje dźwięk).

---

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <windows.h>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int licznik;
    int zakres;

    cout<<" Ustaw zapalnik - czas w sekundach:";
    cin>>zakres;
    for (licznik=zakres;licznik>0;licznik--)
    {
        cout<<licznik<<" ";
        Beep(1000,100);
        Sleep(1000);
    }
    cout<<"Autodestrukcja!\n\0";
    Beep(100,300);

    _getch();
    return 0;
}
```

---

Uruchom napisany program.

#### 2.13.1 Ćwiczenie 13 – wykrywanie błędów w programie, *debugger*

Utwórz nowy projekt i napisz program sprawdzający poprawność numeru PESEL oraz podający czy numer należy do mężczyzny czy do kobiety.

PESEL (Powszechny Elektroniczny System Ewidencji Ludności) jest to 11-cyfrowy, stały symbol numeryczny, identyfikujący jednoznacznie określoną osobę fizyczną. Numer PESEL zawiera informację o dacie urodzenia i płci. Dodatkowo, w jego skład wchodzi liczba porządkowa oraz cyfra kontrolna.

Data urodzenia zapisana jest w następującym porządku: dwie ostatnie cyfry roku, miesiąc i dzień. Dla rozróżnienia stuleci przyjęto następującą metodę kodowania:

- dla lat 1900-1999 miesiąc zapisywany jest w sposób naturalny
- dla lat 1800-1899 do numeru miesiąca dodawana jest liczba 80
- dla lat 2000- 2099 do numeru miesiąca dodawana jest liczba 20

- dla lat 2100- 2199 do numeru miesiąca dodawana jest liczba 40
- dla lat 2200- 2299 do numeru miesiąca dodawana jest liczba 60

Informacja dotycząca płci osoby zawarta jest na 10 pozycji numeru PESEL:

- cyfry 0, 2, 4, 6, 8 – płeć żeńska
- cyfry 1, 3, 5, 7, 9 – płeć męska

Ostatnia cyfra numeru PESEL ma charakter kontrolny. Każdej pozycji numeru PESEL nadany został odpowiedni współczynnik – waga pozycji (patrz tabelka). Algorytm kontrolny numeru PESEL jest następujący: każdą cyfrę numeru mnoży się przez jego wagę i sumuje wyniki mnożenia. Następnie, otrzymany wynik należy podzielić modulo 10 i odjąć wynik od 10.

Numer Cyfry numeru PESEL	1	2	3	4	5	6	7	8	9	10
Znaczenie	Rok		Miesiąc		Dzień		L. Porz.		Płeć	
Waga	1	3	7	9	1	3	7	9	1	3

Przykładowo, dla numeru PESEL 85052203076:

$$suma = 8*1 + 5*3 + 0*7 + 5*9 + 2*1 + 2*3 + 0*7 + 3*9 + 0*1 + 7*3 = 124$$

$$124 \bmod 10 = 4$$

$$10 - 4 = 6$$

Ostatnia cyfra numeru jest równa 6, więc numer jest poprawny, jest to numer należący do mężczyzny ponieważ przedostatnia cyfra jest nieparzysta.

Przykładowy numer należący do kobiety: 86062801487.

Poniżej znajduje się kod programu, który powinien realizować postawione zadanie. Przepisz go. Niestety, w programie wprowadzono 3 błędy. Twoim zadaniem będzie ich znalezienie i poprawienie tak, by program działał prawidłowo. W celu znalezienia błędów skorzystaj z debuggera. Sposób obsługi debuggera opisany jest poniżej kodu programu.

```
#include "stdafx.h"
#include "stdlib.h"
#include <iostream>
#include <conio.h>
using namespace std;

bool sprawdz_numer_PESEL(char PESEL[12]) {
    //sprawdzanie poprawnosci numeru PESEL
    int w[12]={1,3,7,9,1,3,7,9,1,3,0}; //tablica wag
    int i, k, s;
    char cyfraT;
    int cyfra;
    //pojedynczy znak z numeru (jako tekst)
    //pojedyncza cyfra z numeru (jako liczba)

    s=0;
    for (i=0;i<=9;i++){
        strncpy(&cyfraT,&PESEL[i+1],1); //pobranie jednego znaku z numeru (jako tekst)
        cyfra=atoi(&cyfraT);           //konwersja tekstu z cyfrą na jej
        //wartosc numeryczna
        s=s+cyfra*w[i];                 //wyznaczanie sumy
    }
    s=s%10;                             //dzielenie modulo 10
    k=(10-s)%10;                        //wyznaczenie cyfry kontrolnej

    //porównanie cyfry kontrolnej z ostatnia cyfra numeru
    if (atoi(&PESEL[10])==k) return true;
    else return false;
}

bool sprawdz_plec(char PESEL[12]) {
    //sprawdzanie plci, true=mezczyzna
    char cyfraT;

    strncpy(&cyfraT,&PESEL[9],1);
    if ((atoi(&cyfraT)/2)!=0) return true;
}
```

```

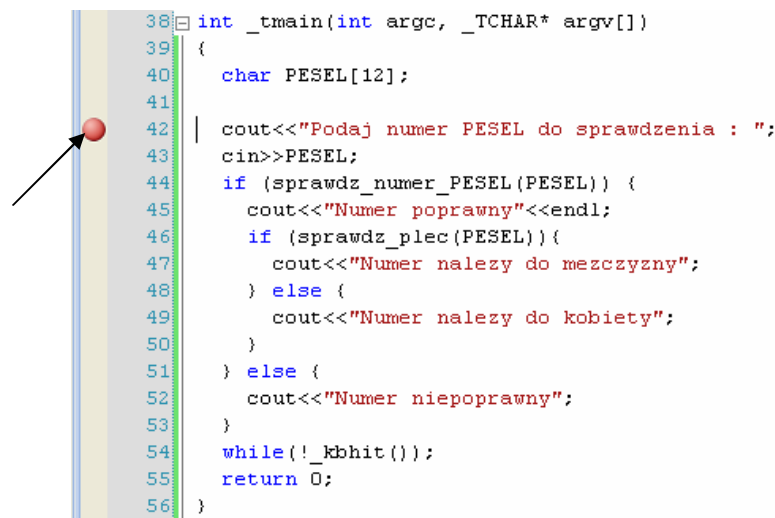
        else return false;           //jesli nieparzysta - mezczyzna, zwracamy true(prawda)
        //w przeciwnym wypadku - kobieta, zwracamy false(falsz)
    }

int _tmain(int argc, _TCHAR* argv[])
{
    char PESEL[12]={0};              //12 znaków, ponieważ ostatni powinien
                                    //byc znakiem pustym


    cout<<"Podaj numer PESEL do sprawdzenia : ";
    cin>>PESEL;
    if (sprawdz_numer_PESEL(PESEL)) {
        cout<<"Numer poprawny"<<endl;
        if (sprawdz_plec(PESEL)){
            cout<<"Numer należy do mezczyzny";
        } else {
            cout<<"Numer należy do kobiety";
        }
    } else {
        cout<<"Numer niepoprawny";
    }
    while(!_kbhit());
    return 0;
}

```

W funkcji `_tmain()` ustaw kursor na linii „`cout<<"Podaj numer PESEL do sprawdzenia : ";`” i wciśnij F9 lub kliknij myszą na pasku, na lewo od numeracji linii. Pojawi się duży, czerwony punkt (Rys. 10). Ustawiony punkt to tzw. *breakpoint* – miejsce, w którym wykonywanie programu zostanie wstrzymane.



Rys. 10

Uruchom napisany kod – jego wykonanie zatrzyma się na początku programu. Przechodzenie do następnej linii następuje po wciśnięciu klawisza F10 lub kliknięciu ikonki . Po wykonaniu każdej linii obserwuj zmiany na ekranie konsoli (konieczne będzie przełączanie się między ekranem konsoli, a ekranem Visual C++). Obserwuj także zmiany wartości poszczególnych zmiennych. W tym celu najedź kursorem myszy na nazwę zmiennej (nie klikaj jej). Po chwili pojawi się jej aktualna wartość. Można także dodać zmienną do okienka podglądu (*Watch*) – w tym celu kliknij nazwę zmiennej prawym klawiszem myszy i wybierz „*Add watch*”. By przerwać pracę debuggera i zakończyć program wciśnij Shift+F5. Podczas poszukiwania błędu sprawdzaj w szczególności wartości zmiennych `s`, `cyfra` oraz `cyfraT`. Kontroluj także, czy wyniki poszczególnych operacji są takie jak powinny. Sprawdź działanie programu dla numerów należących do kobiet i do mężczyzn.



## 2.14 Ćwiczenie 14 – Pętla *for*

Napisz program wypisujący wszystkie dzielniki zadanej liczby naturalnej, liczbę tych dzielników oraz ich sumę. Program powinien podawać komunikat, jeżeli podana liczba jest liczbą pierwszą.

np.     liczba 123  
           dzielniki naturalne liczby: 1, 3, 41, 123  
           znaleziono dzielników: 4  
           suma dzielników = 168

liczba 13  
           dzielniki naturalne liczby: 1, 13  
           znaleziono dzielników: 2  
           suma dzielników = 14  
           podana liczba jest liczbą pierwszą

liczba 52  
           dzielniki naturalne liczby: 1, 2, 4, 13, 26, 52  
           znaleziono dzielników: 6  
           suma dzielników = 98

Algorytm: w pętli należy sprawdzić czy reszty z dzielenia zadanej liczby  $a$  przez kolejne liczby z zakresu od 1 do  $a/2$  są równe 0. Jeśli reszta jest równa 0, wówczas należy wyświetlić znaleziony dzielnik, zwiększyć wartość zmiennej, na której zliczana jest liczba dzielników, oraz dodać znaleziony dzielnik do sumy poprzednich dzielników. Po zakończeniu pętli należy jeszcze uwzględnić dodatkowo fakt, że liczba  $a$  jest zawsze także swoim własnym dzielnikiem. Następnie należy wyświetlić odpowiednie komunikaty zgodnie z poleceniem oraz, dodatkowo, na podstawie liczby znalezionych dzielników sprawdzić czy liczba  $a$  jest liczbą pierwszą.

### Uwagi:

- W programie przydatne mogą być następujące funkcje i operacje:  
 Operator  $a \% b$  lub funkcja *fmod()* – wyznaczanie reszty z dzielenia  $a/b$ , funkcja *fmod()* znajduje się w bibliotece *math.h*  
 $x = \text{ceil}(y)$  – zaokrąglanie w górę (funkcja z biblioteki *math.h*)  
`static_cast <double>( )` – rzutowanie typów na typ zmiennoprzecinkowy *double*

## 2.15 Ćwiczenie 15 – Pętla *for*

Utwórz nowy projekt i napisz program, który wygeneruje zadaną liczbę wyrazów ciągu Fibonacciego. Ciąg ten ma postać 1, 1, 2, 3, 5, 8, 13, 21, ... i jest opisany wzorem ogólnym:  $n_k = n_{k-2} + n_{k-1}$ , tzn. każdy kolejny wyraz jest sumą dwóch poprzednich.

Przykład wywołania

Podaj liczbę wyrazów ciągu Fibonacciego: 12  
 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,

Uruchom napisany program.

## 2.16 Ćwiczenie 16 – Pętla *for*

Utwórz nowy projekt i napisz program, który wyświetli tabliczkę mnożenia (konieczne będzie użycie dwóch zagnieżdżonych jedna w drugiej pętli *for* oraz funkcji *printf()*). Uruchom napisany program.

Wynik działania programu:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100



## 2.17 Ćwiczenie 17 – Pętla *for*, tablice

### 2.17.1 Ćwiczenie 17a

Utwórz nowy projekt i napisz program, który ponumeruje pola w tablicy o rozmiarze 4x4 wg wzoru:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

W programie użyj pętli *for*.

---

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;

// deklaracja stałych globalnych i ich wartości
const int lw=4;           //rozmiary tablicy, liczba wierszy i kolumn
const int lk=4;

void wyswietlTab(int tablica[lw][lk])
//wyswietlanie tablicy
{
    int w, k;              //numer kolumny i wiersza, zmienne lokalne

    for (w=0;w<lw;w++){    //wyswietlanie gotowej tablicy
        for (k=0;k<lk;k++){
            printf("%3i",tablica[w][k]);
        }
        printf("\n");
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    //deklaracje zmiennych lokalnych
    int tablica[lw][lk]; //tablica do wypełnienia
    int w, k;            //numer wiersza i kolumny
    int nr;              //numer do wpisania

    nr=0;
    for (w=0;w<lw;w++){  //wstawianie wartosci do tablicy
        for (k=0;k<lk;k++){
            nr++;
            tablica[w][k]=nr;
        }
    }

    wyswietlTab(tablica); //wyswietlanie gotowej tablicy

    _getch();
    return 0;
}
```

---

Uruchom napisany program.

Najważniejszą częścią programu jest para zagnieżdżonych pętli *for* wpisujących wartości do tablicy. Pętla zewnętrzna (z licznikiem *w*) odpowiedzialna jest za zmianę numeru wiersza. Pętla wewnętrzna (z licznikiem *k*) odpowiedzialna jest za zmianę numeru kolumny. Wpisywanie wartości odbywa się w ten sposób, że w kolejnych iteracjach (wykonaniach)

pętli wewnętrznej wpisywane są kolejne wartości w danym wierszu. Po wypełnieniu wiersza program przechodzi do następnej iteracji pętli zewnętrznej, co powoduje „przejsie” do kolejnego wiersza i ponowne rozpoczęcie pętli wewnętrznej. Wyświetlenie wyników zawarte w funkcji `wyswietlTab()` oparte jest na tej samej zasadzie. Jedynie, zamiast wpisywać wartości do tablicy, są one z niej pobierane, zaś po wypisaniu elementów jednego wiersza wstawiany jest znak końca linii by przejść do następnej linii na ekranie.

### 2.17.2 Ćwiczenie 17b

Zmodyfikuj program, tak by ponumerował pola tablicy wg. wzoru:

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

Nowa tablica powinna być wyświetlana oprócz poprzedniej.

Uruchom napisany program.

### 2.17.3 Ćwiczenie 17c

Zmodyfikuj program, tak by ponumerował pola tablicy wg wzorów:

A			
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

b			
1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

c			
0	2	3	4
4	0	4	5
9	16	0	6
16	25	36	0

Uruchom napisany program (powinien wyświetlać kolejno wszystkie 5 tablic, w tym 2 z poprzednich części ćwiczenia).

**Uwagi:**

- Konieczne jest odpowiednie operowanie zakresami pętli i instrukcjami warunkowymi wewnątrz nich. Może być też wymagane zbudowanie dwóch pętli wewnętrznych, uruchamianych w zależności od numeru wiersza lub kolumny. Program powinien być napisany w taki sposób, by działał dla tablicy o dowolnych rozmiarach (a nie tylko 4x4). Zmiana rozmiaru będzie następowała poprzez zmianę wartości stałych `lw` i `lk`.
- W przypadku c) wypełnianie opiera się na innych zasadach niż w pozostałych przypadkach.

### 2.17.4 Ćwiczenie 17d

Zmodyfikuj program, tak by ponumerował pola tablicy wg wzorów:

a			
1	3	6	10
2	5	9	13
4	8	12	15
7	11	14	16

b			
1	4	9	16
2	3	8	15
5	6	7	14
10	11	12	13

c			
1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

Uruchom napisany program.

**Uwagi:**

- Konieczne jest odpowiednie operowanie zakresami pętli i instrukcjami warunkowymi wewnątrz nich. Może być też wymagane zbudowanie kilku pętli wewnętrznych uruchamianych w zależności od aktualnych wartości liczników wierszy lub kolumn. Program powinien być napisany w taki sposób, by działał dla tablicy o dowolnych rozmiarach (a nie tylko 4x4).

## 2.18 Ćwiczenie 18 – Pętla *while*

Utwórz nowy projekt i napisz program zbierający od użytkownika datki tak długo, aż uzbiera określoną kwotę.

### Uwagi:

- Wykorzystana zostanie pętla *while*, która pozwala wykonać blok instrukcji tak długo, jak długo spełniony jest zadany warunek (warunek trwania pętli, a nie jej zakończenia). Pętli *while* używamy, gdy w momencie rozpoczęcia pętli liczba powtórzeń nie jest znana (np. zależy od danych lub decyzji podawanych / podejmowanych przez użytkownika w trakcie trwania pętli). W naszym przypadku nie wiemy, jakie datki będzie przekazywać użytkownik i dlatego nie można użyć pętli *for*.

---

```
#include "stdafx.h"
#include <conio.h>
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int suma=0, cel, datek;

    cout<<"Podaj kwotę do zebrania:"<<endl;
    cin>>cel;
    while (suma<=cel){
        cout<<"Proszę o datek: ";
        cin>>datek;
        suma=suma+datek;
        cout<<"Dziękuję!"<<endl;
    }
    cout<<"Zebrałem= "<<suma<<endl;
    _getch();
    return 0;
}
```

---

Uruchom napisany program.

## 2.19 Ćwiczenie 19 – Pętla *while*

Utwórz nowy projekt i napisz program „Zgadywanka”, który wylosuje pewną liczbę całkowitą z zakresu <0; 100>, a następnie będzie pytał użytkownika jaka liczba została wylosowana. Po podaniu przez użytkownika liczby, program powinien komunikować, czy liczba ta jest większa czy mniejsza od liczby wylosowanej. Dodatkowo program powinien po zakończeniu zgadywania podać w ilu próbach użytkownikowi udało się zgadnąć wylosowaną liczbę.

Do wylosowania liczby możesz użyć poniższego fragmentu kodu

---

```
int liczbaZagadka;
srand(unsigned int(time(NULL))); //inicjalizacja funkcji losującej
liczbaZagadka=double((rand()+rand())/2)/double(RAND_MAX + 1)*100;
```

---

Podwójne użycie funkcji *rand()* pozwala zwiększyć rozrzut losowań przy kolejnych uruchomieniach programu. Do działania powyższego kodu konieczne jest dołączenie do programu biblioteki *time.h*.

Uruchom napisany program.

## 2.20 Ćwiczenie 20 – Pętle i instrukcje warunkowe

Utwórz nowy projekt i napisz program, który w podanym tekście znajdzie pozycję drugiej spacji. Rozwiązanie w ćwiczeniu 20a i 20b.

### 2.20.1 Ćwiczenie 20a

Dygresja - gdyby zadanie brzmiało „znajdź pozycję pierwszej spacji” można by wykorzystać gotową funkcję `strchr(tekst, znak)`, która zwraca wskaźnik do komórki pamięci, w której znajduje się pierwsze wystąpienie znaku w tekście.

Rozwiązanie:

```
#include "stdafx.h"
#include <string.h>
#include <stdio.h>
#include <conio.h>

int _tmain(int argc, _TCHAR* argv[])
{
    char tekst[100]; //tablica z wpisanym tekstem
    char *wsk_z;
    int poz_z;
    char znak=' '; //znak spacji

    printf("Wpisz tekst: ");
    gets_s(tekst); //pobranie linii tekstu
    wsk_z=strchr(tekst,znak); //wyszukanie pozycji znaku (zwracany jest wskaźnik)
    if (wsk_z==NULL) //wskaźnik NULL znaczy, że znak nie został znaleziony
        printf("Nie znaleziono spacji");
    else {
        poz_z = (int)(wsk_z - tekst + 1); //wyliczenie pozycji znaku
        //wskaźnik do znaku - wskaźnik do początku tekstu +1
        //następnie rzutowanie na typ int
        printf("Spacja na pozycji %i",poz_z);

        _getch();
        return 0;
    }
}
```

Uruchom napisany program.

### 2.20.2 Ćwiczenie 20b

Ponieważ w zadaniu znaleźć należy drugą (a nie pierwszą) spację, trzeba napisać własną funkcję. Dwa przykładowe rozwiązania podane są poniżej – różnią się one sposobem przechowywania tekstu na zmiennej. W rozwiązaniu 1 jest to tablica znaków, w 2 – zmienna typu *string*. W obu przypadkach napisano funkcję, która jako zmienną wejściową przyjmuje tekst, zaś na wyjściu zwraca pozycję drugiej znalezionej spacji. Jeśli w tekście są mniej niż dwie spacje – zwracana jest wartość 0.

Przepisz jedno z rozwiązań.

Rozwiązanie 1	Rozwiązanie 2
<pre>#include "stdafx.h" #include &lt;stdio.h&gt; #include &lt;conio.h&gt; #include &lt;string&gt; #include &lt;iostream&gt; using namespace std;  unsigned int  znajdz2sp_v1(char tekst[100]){     unsigned int poz_z; //pozycja zmiennej     char znak=' '; //szukany znak (spacja)     unsigned int nr_spacji;//nr znalezionej spacji</pre>	<pre>#include "stdafx.h" #include &lt;stdio.h&gt; #include &lt;conio.h&gt; #include &lt;string&gt; #include &lt;iostream&gt; using namespace std;  unsigned int  znajdz2sp_v2(string tekst){     string t_pom; //pomocniczy string     unsigned int poz_z; //pozycja zmiennej     string znak(" "); //szukany znak (spacja)     unsigned int nr_spacji; //nr znalezionej spacji</pre>

<pre> poz_z=0; nr_spacji=0; while ((poz_z&lt;strlen(tekst))&amp;&amp;(nr_spacji&lt;2)){      if (tekst[poz_z]==znak){         //porównanie jednej litery z tekstu         //ze znakiem (spacja)         nr_spacji++;         //zwiększenie licznika znalezionych spacji     }     poz_z++;     //zwiększenie pozycji przed następna petla }  if ((poz_z==0)   (nr_spacji&lt;2))     return 0; //0 gdy nie ma 2 spacji else     return poz_z; }  int _tmain(int argc, _TCHAR* argv[]) {     char tekst[100];     unsigned int poz_sp2;      printf("Wpisz tekst: ");     gets_s(tekst);     //pobranie linii tekstu i przypisanie do     //tablicy znaków char      poz_sp2=znajdz2sp_v1(tekst); //wywołanie funkcji      if (poz_sp2==0)         printf("Nie znaleziono spacji");     else         printf("Spacja nr 2 na pozycji %i",poz_sp2);      _getch();     return 0; } </pre>	<pre> poz_z=0; nr_spacji=0; while ((poz_z&lt;tekst.length())&amp;&amp;(nr_spacji&lt;2)){     t_pom=tekst.substr(poz_z,1);     //pobranie pojedynczej litery ze stringa     if (t_pom.compare(znak)==0){         //porównanie stringa (w tym wypadku         //jednej litery) z znakiem (spacja)         nr_spacji++;         //zwiększenie licznika znalezionych spacji     }     poz_z++;     //zwiększenie pozycji przed następna petla }  if ((poz_z==0)   (nr_spacji&lt;2))     return 0; //0 gdy nie ma 2 spacji else     return poz_z; }  int _tmain(int argc, _TCHAR* argv[]) {     string tekst2;     unsigned int poz_sp2;      printf("Wpisz tekst: ");     std::getline(cin,tekst2,'\n');     //pobranie lini tekstu     //uwaga: cin&gt;&gt; pobiera tekst tylko do pierwszej     //spacji, dalej ucina, stad użycie getline()     poz_sp2=znajdz2sp_v2(tekst2); //wywołanie funkcji      if (poz_sp2==0)         printf("Nie znaleziono spacji");     else         printf("Spacja nr 2 na pozycji %i",poz_sp2);      _getch();     return 0; } </pre>
---	--

Uruchom napisany program.

### 2.20.3 Ćwiczenie 20c

Zmodyfikuj program tak, aby funkcja wyszukiwała spację o zadanym numerze. Program powinien prosić o wpisanie tekstu, następnie o podanie numeru spacji do wyszukania. Funkcję z poprzedniego ćwiczenia należy zmodyfikować tak, by miała dwa parametry wejściowe: tekst oraz numer spacji do wyszukania. Pamiętaj o zachowaniu poprawnego „stylu kodowania”.

Uruchom napisany program.

### 2.20.4 Ćwiczenie 20d

Napisz program (korzystając z gotowej funkcji z poprzedniego ćwiczenia), który:

- poprosi o podanie tekstu
- poprosi o podanie numeru wyrazu
- wyświetli wyraz o podanym numerze

Podpowiedź: Na podstawie numeru wyrazu można określić, która spacja znajduje się przed i za wyrazem. Należy znaleźć ich pozycje, a następnie wyświetlić fragment tekstu pomiędzy tymi spacjami. Do „wycięcia” fragmentu tekstu można użyć:

- pętli
- funkcji *strncpy*(wskaźnik\_do\_celu, wskaźnik\_do\_źródła, liczba\_znaków)
  - jako celu można użyć dodatkowej zmiennej pomocniczej
  - wskaźnik do źródła można wyznaczyć znając wskaźnik do początku tekstu oraz pozycję spacji przed szukanym słowem (można je dodać)

- źródłem jest tekst, z którego „wycinamy” ciąg znaków; „wycięty” ciąg kopiowany jest do celu

- funkcji *tekst.substr*(pozycja, liczba\_znaków)

Uwaga: przemyśl program, tak by działał poprawnie także dla pierwszego i ostatniego wyrazu oraz przypadku, gdy tekst ma mniej wyrazów niż numer wyrazu zadanego do wyświetlenia.

Uruchom napisany program i przetestuj jego działanie dla różnych przypadków.

## 2.21 Ćwiczenie 21 – Operacje na tablicach, pętle

### 2.21.1 Ćwiczenie 21a

Utwórz nowy projekt i napisz program, który pomnoży dwa wektory o rozmiarze 1x5.

Program powinien najpierw wyświetlić oba wektory, a następnie ich iloczyn. Do wypełnienia wektorów wartościami początkowymi można użyć funkcji *RangedRand*(min, max), która zwraca liczby całkowite z zakresu min ~ max, lub (na etapie testowania) na stałe przypisać wartości przy deklarowaniu zmiennych tablicowych.

**Uwagi:**

- Na początku funkcji *\_tmain*() znajduje się wywołanie funkcji *srand*(). Funkcja ta inicjalizuje generator liczb losowych. Jeżeli nie zostanie on zainicjalizowany, przy każdym uruchomieniu programu losowany ciąg liczb zawsze będzie taki sam.

---

```
#include "stdafx.h"
#include <conio.h>
#include <time.h>
#include <iostream>
using namespace std;

int RangedRand(int range_min, int range_max)
{    //funkcja losująca liczby całkowite z podanego zakresu
    int u;
    u = (double)rand() / (RAND_MAX + 1) * (range_max - range_min) + range_min;
    return u;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int x[5], y[5];           //deklaracja tablic, rozmiar 5x5
    long int wynik;
    int i;

    srand((int) time(NULL)); //inicjalizacja funkcji losującej

    printf("x: ");           //inicjalizacja x
    for (i=0; i<5; i++){
        x[i]=RangedRand(-10,10);
        printf("%8i", x[i]);
    }
    printf("\n");

    printf("y: ");           //inicjalizacja y
    for (i=0; i<5; i++){
        y[i]=RangedRand(-10,10);
        printf("%8i", y[i]);
    }
    printf("\n");

    printf("x*y: ");
    wynik=0;
```

```
for (i=0;i<5;i++){           //obliczenia
    wynik=wynik+x[i]*y[i];
}
printf("%5i",wynik);
printf("\n");

_getch();
return 0;
}
```

Uruchom napisany program.

### 2.21.2 Ćwiczenie 21b

Utwórz nowy projekt i napisz program, który pomnoży dwie macierze o rozmiarze 5x5. Do wypełnienia macierzy wartościami początkowymi można użyć funkcji *RangedRand*(min, max) podanej poprzednio, lub (na etapie testowania) na stałe przypisać wartości przy deklarowaniu zmiennych tablicowych. Na etapie testowania programu ogranicz rozmiary macierzy do 3x3 by łatwiej sprawdzić poprawność obliczeń.

Uruchom napisany program.

### 2.21.3 Ćwiczenie 21c

Zmodyfikuj program tak, by zamiast tablic o z góry znanych rozmiarach wykorzystywał tablice dynamiczne, a ich rozmiar był zadawany przez użytkownika.

Pamiętaj o sposobie deklarowania zmiennych dynamicznych i wskaźników do nich:

- deklaracja wskaźnika  
`int *x, *y;`
- utworzenie zmiennej dynamicznej i nadanie wskaźnikowi adresu do niej  
`x=new int [dl_wekt];`
- usunięcie zmiennej tablicowej  
`delete[] x;`

#### Uwagi:

- W przypadku zmiennych dynamicznych nie można zadeklarować tablicy dwuwymiarowej. Należy zarezerwować obszar pamięci o odpowiednim rozmiarze będącym iloczynem obu wymiarów tzn:  $rozmiar = rozmiar\_x * rozmiar\_y$ . Następnie, przy odwoływaniu się do poszczególnych komórek zarezerwowanego obszaru należy pamiętać o odpowiednim korygowaniu adresów względem początku obszaru pamięci wskazywanego przez wskaźnik. Np. by w macierzy o 3 kolumnach odwołać się do elementów z wiersza 0 korekta =0, dla wiersza 1 korekta = +3, dla wiersza 2 korekta =+6 itd.

Adresy w tablicy statycznej

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

Adresy względne w tablicy dynamicznej

+0	+1	+2
+3	+4	+5
+6	+7	+8

- Innym sposobem rozwiązania jest zadeklarowanie tablicy dynamicznej o liczbie elementów równej liczbie wierszy w tablicy dwuwymiarowej. Elementami tej tablicy będą wskaźniki do dalszych tablic, odpowiadających wierszom tablicy dwuwymiarowej. Przykład dla przypadku tablicy o 2 wierszach i 3 kolumnach realizuje poniższy kod:

---

```
int **x;           //wskaźnik do wskaźnika
x = new int*[2]; //rezerwacja pamięci dla tablicy wskaźników
x[0] = new int[3]; //rezerwacja pamięci dla kolejnych wierszy tablicy
x[1] = new int[3];
x[2] = new int[3];
```

---

Po zakończeniu operacji należy pamiętać o zwolnieniu pamięci w odwrotnej kolejności niż była rezerwowana, tj. najpierw zwolnić pamięć rezerwowaną dla kolejnych wierszy, a dopiero potem usunąć zmienną wskaźnikową *x*.

Wybierz jedno z rozwiązań, wprowadź modyfikacje do programu i uruchom go.

## 2.22 Ćwiczenie 22 – Operacje na tablicach, pętle

### 2.22.1 Ćwiczenie 22a

Utwórz nowy projekt i napisz program, który dla tablicy liczb całkowitych o wymiarach 8x8 (wartości wpisywane przez użytkownika lub losowe) obliczy:

- Sumy liczb w wierszach (dla każdego wiersza osobno)
- Sumy liczb w kolumnach (dla każdej kolumny osobno)
- Sumy liczb na obu przekątnych (dla każdej przekątnej osobno)

Użyj pętli.

Uruchom napisany program.

### 2.22.2 Ćwiczenie 22b

Zmodyfikuj program tak, by wykorzystywał zmienne dynamiczne i wskaźniki do nich a rozmiar tablicy był podawany przez użytkownika.

Uruchom napisany program.

## 2.23 Ćwiczenie 23 – Operacje na tablicach, pętle

Utwórz nowy projekt i napisz program, który wczyta do tablicy (najlepiej dynamicznej) *k* liczb całkowitych z przedziału  $\langle -5, 5 \rangle$ , a następnie wyznaczy dla nich histogram (poda ile razy poszczególne liczby występują w tablicy wejściowej). Liczniki poszczególnych wartości umieść w tablicy o długości / rozmiarze = 11.

Np. dla liczb -5, 5, -5, 2, -2, 1, 4, 5, 0, 2, 1, 1, 1, -2

Wynik działania programu będzie następujący:

-5	-4	-3	-2	-1	0	1	2	3	4	5
2	0	0	2	0	1	4	2	0	1	2

## 2.24 Ćwiczenie 24 – Operacje plikach binarnych

Napisz program, który zapisze w pliku binarnym kilka informacji, a następnie odczyta je i wyświetli na ekranie.

Założmy, że do zapisania będą następujące zmienne:

- Liczba całkowita typu *int*
- Liczba zmiennoprzecinkowa typu *double*
- Krótki tekst

Zadanie to można wykonać na dwa sposoby. Pierwszy z nich korzysta z funkcji i jest zgodny z metodami programowania w języku C i C++. Drugi – korzysta z klasy obsługującej strumienie plikowe i jest zgodne z obiektowymi metodami programowania w języku C++.

### Uwagi do pierwszej metody rozwiązania:

Przed zapisaniem pliku należy otworzyć strumień do niego (funkcja *fopen()* z biblioteki *iostream.h*) i określić, czy plik ma być otwarty do zapisu czy do odczytu. Następnie, przy



użyciu funkcji *fwrite()* lub *fread()* zapisuje się lub odczytuje kolejne wartości do / z pliku. Po zakończeniu operacji plik musi zostać zamknięty funkcją *fclose()*.

**Składnia funkcji *fopen()*:**

```
wskaznik_do_pliku=fopen(nazwa_pliku,tryb_dostepu);
```

- nazwa\_pliku – tablica z tekstem – nazwą pliku
- tryb\_dostepu – ciąg tekstowy opisujący tryb dostępu, „w” – oznacza, że do pliku będziemy tylko zapisywać dane (nie można ich w tym samym czasie czytać), „r” – tylko odczyt (bez jednoczesnego zapisu), „a” – dodanie danych do istniejącego pliku, „t” – otwarcie pliku jako pliku tekstowego, „b” – otwarcie pliku jako pliku binarnego. W większości przypadków wygodniejszy jest dostęp do pliku w sposób binarny, nawet jeśli zawiera on wyłącznie tekst.
- wskaznik\_do\_pliku – jeśli otwarcie pliku powiodło się (plik do odczytu istnieje lub plik do zapisu udało się otworzyć), funkcja zwraca wskaznik do strumienia związanego z tym plikiem. W przeciwnym razie, wskaznik ma wartość NULL.

**Składnia funkcji *fwrite()*:**

```
fwrite(adres_zmiennej,rozmiar_zmiennej,liczba_elementow,strumien);
```

- adres\_zmiennej – referencja do zmiennej, którą chcemy zapisać. By przekazać referencję (podać adres zmiennej) należy użyć operatora `&`, np.: `&zmienna`.
- rozmiar\_zmiennej – liczba bajtów, które zajmuje zmienna. Najbezpieczniejszym rozwiązaniem jest wyznaczenie rozmiaru z użyciem funkcji *sizeof()*. Możemy podać nazwę zmiennej, np.: *sizeof(zmienna)* lub wprost, nazwę typu zmiennej np.: *sizeof(double)*. Ten drugi sposób jest zwykle wygodniejszy w przypadku, gdy chcemy zapisać do pliku zmienną tablicową.
- liczba\_elementow – liczba elementów które chcemy zapisać do pliku, przy zapisie pojedynczej zmiennej będzie to wartość 1. Przy zapisie tablicy, będzie to liczba elementów tablicy, które chcemy zapisać.
- strumien – wskaznik do strumienia do pliku

**Składnia funkcji *fread()*:**

```
fread(adres_bufora,rozmiar_zmiennej,liczba_elementow,strumien);
```

- adres\_zmiennej – referencja do zmiennej, będącej buforem, na który chcemy odczytać dane z pliku.
- rozmiar\_zmiennej – liczba bajtów, które zajmuje pojedynczy element, który chcemy odczytać.
- liczba\_elementow – liczba elementów, które chcemy odczytać z pliku, przy odczycie pojedynczej zmiennej będzie to wartość 1. Przy odczycie np.: tablicy, będzie to liczba elementów tablicy, które chcemy odczytać.
- strumien – wskaznik do strumienia do pliku

#### **Uwagi do drugiej metody rozwiązania:**

Podobnie jak poprzednio, przed zapisaniem pliku należy otworzyć strumień do niego (metoda *open()* z klasy *fstream* z biblioteki *fstream.h*) i określić tryb dostępu do pliku. Następnie, przy użyciu metod *write()* lub *read()* zapisuje się lub odczytuje kolejne wartości do / z pliku. Po zakończeniu operacji plik musi zostać zamknięty metodą *close()*.

### Składnia funkcji *fopen()*:

```
strumien.fopen(nazwa_pliku, tryb_dostępu);
```

- nazwa\_pliku – tablica z tekstem – nazwą pliku. Jeżeli nazwa pliku przechowywana jest na zmiennej typu String, do pobrania tablicy znaków należy użyć metody *c\_str()* z klasy String.
- tryb\_dostępu – kombinacja stałych z przestrzeni nazw *ios* opisujących tryb dostępu, *ios::out* – oznacza, że do pliku będziemy tylko zapisywać dane (nie można ich w tym samym czasie czytać), *ios::in* – tylko odczyt (bez jednoczesnego zapisu), *ios::text* – otwarcie pliku jako pliku tekstowego, *ios::binary* – otwarcie pliku jako pliku binarnego. Do łączenia poszczególnych stałych należy użyć operatora sumy bitowej „|”.
- – Do sprawdzenia, czy otwarcie strumienia do pliku powiodło się, służy metoda *is\_open()* z klasy *fstream*.

### Składnia metody *write()*:

```
strumien.write(adres_zmiennej, rozmiar_zmiennej);
```

- adres\_zmiennej – referencja do zmiennej, którą chcemy zapisać. By przekazać referencję (podać adres zmiennej) należy użyć operatora *&*, np.: *&zmienna*. Uwaga: w metodzie *write()* przyjęto, że zmienna, którą zapisujemy jest typu *char* i zajmuje 1 bajt. Powoduje to, że jeżeli chcemy zapisać zmienną jakiegokolwiek innego typu, musimy przekształcić wskaźnik do niej na wskaźnik typu *char\**. Do konwersji typów wskaźników służy rzutowanie typu *reinterpret\_cast*. Np.: *reinterpret\_cast<char\*>(&zz1)* spowoduje, że wskaźnik do zmiennej *zz1* typu *int* zostanie potraktowany jak wskaźnik do zmiennej typu *char*.
- rozmiar\_zmiennej – liczba bajtów, które zajmuje zmienna. Najbezpieczniejszym rozwiązaniem jest wyznaczenie rozmiaru z użyciem funkcji *sizeof()*. Możemy podać nazwę zmiennej, np.: *sizeof(zmienna)* lub wprost, nazwę typu zmiennej np.: *sizeof(double)*. Ten drugi sposób jest zwykle wygodniejszy w przypadku, gdy chcemy zapisać do pliku zmienną tablicową. Zapisanie np. jednej zmiennej typu *int* (4 bajtowej) odpowiada zapisaniu 4-ch jednobajtowych (typ *char*) fragmentów tej zmiennej. Dlatego w metodzie *write()* jej twórcy przyjęli, że potrafi ona zapisac tylko zmienne typu *char* – wszystkie inne można łatwo przekształcić do tej postaci.

### Składnia metody *read()*:

```
strumien.read(adres_bufora, rozmiar_zmiennej, liczba_elementow, strumien);
```

- adres\_zmiennej – referencja do zmiennej, będącej buforem, na który chcemy odczytać dane z pliku. Uwaga: w metodzie *read()* przyjęto, że zmienna, którą odczytujemy jest typu *char* i zajmuje 1 bajt. Powoduje to, że jeżeli chcemy odczytać zmienną jakiegokolwiek innego typu, musimy przekształcić wskaźnik do niej na wskaźnik typu *char\**. Do konwersji typów wskaźników służy rzutowanie typu *reinterpret\_cast*.
- rozmiar\_zmiennej – liczba bajtów, które zajmuje pojedynczy element, który chcemy odczytać.

**Uwaga:** w przypadku zapisu do pliku tablicy, w szczególności umieszczonego w tablicy znakowej tekstu, są dwie możliwości zapisu.

1. Zapisać całą tablicę – wówczas zapisywana jest zawsze cała tablica, nawet wtedy, gdy nie wszystkie jej elementy są wypełnione. W przypadku tekstu oznacza to, że gdy tekst jest krótszy niż maksymalny, w pliku zapisywane są także niepotrzebne znaki znajdujące się w tablicy „za” tekstem. Tak zapisywany plik będzie miał zawsze taki sam rozmiar, niezależnie od długości użytecznego tekstu. W takim wypadku, przy odczycie pliku również należy odczytywać taką samą liczbę elementów tablicy.
2. Zapisać fragment tablicy – wówczas zapisywanych jest tylko tyle elementów tablicy, ile w rzeczywistości jest wykorzystanych. Rodzi to jednak problem przy odczycie pliku gdyż nie wiadomo ile elementów należy odczytać. Aby ominąć ten problem można np. przed zapisem elementów tablicy, zapisać w pliku dodatkową informację o liczbie elementów tablicy, które zapisano. Później, przy odczycie pliku należy najpierw odczytać tę informację, a następnie wykorzystać ją do określenia liczby elementów, którą należy odczytać przy odczycie danych z tablicy.

W obu przykładach zawartych poniżej zastosowano drugie rozwiązanie.

Wybierz jedno z przedstawionych poniżej rozwiązań i przepisuj je, a następnie sprawdź jego działanie.

#### Rozwiązanie 1

---

```
#include "stdafx.h"
#include <conio.h>
#include <iostream>
using namespace std;

char nazwa[256]="plik1.dat\0"; //nazwa pliku

void zapis1(){
    FILE *strumien; //wskaznik do strumienia do pliku
    int zz1=4; //pierwsza zmienna do zapisania
    double zz2=3.14; //druga zmienna do zapisania
    char ztekst[10]={0}; //tekst do zapisania
    size_t rt; //rozmiar tekstu

    strcpy(ztekst, "-TEXT-\0"); //inicjalizacja tekstu

    strumien=fopen(nazwa, "wb"); //otwarcie pliku binarnego do zapisu
    if (strumien!=NULL){
        fwrite(&zz1, sizeof(zz1), 1, strumien); //zapis zmiennej 1
        fwrite(&zz2, sizeof(zz2), 1, strumien); //zapis zmiennej 2
        rt=strlen(ztekst); //sprawdzenie dlugosci tekstu
        fwrite(&rt, sizeof(rt), 1, strumien); //zapisanie dlugosci tekstu
        fwrite(&ztekst, sizeof(char), strlen(ztekst), strumien); //zapisanie tekstu
        fclose(strumien); //zamkniecie pliku
        cout<<"Zapisano plik "<<nazwa<<endl;
    } else {
        cout<<"Nie mozna utworzyc pliku";
    }
}

void odczyt1(){
    FILE *strumien; //wskaznik do strumienia do pliku
    int oz1; //bufor dla pierwszej odczytanej wartosci
    double oz2; //bufor dla drugiej odczytanej wartosci
    char otekst[10]={0}; //bufor dla odczytanego tekstu
    size_t rt; //bufor dla odczytanego rozmiaru tekstu
```

```

strumien=fopen(nazwa,"rb");//otwarcie pliku binarnego do odczytu
if (strumien!=NULL){
    fread(&oz1 ,sizeof(oz1) ,1 ,strumien); //odczyt pierwszej wartosci
    fread(&oz2 ,sizeof(oz2) ,1 ,strumien); //odczyt drugiej wartosci
    fread(&rt ,sizeof(rt) ,1 ,strumien); //odczyt dlugosci tekstu
    fread(&otekst,sizeof(char),rt,strumien); //odczytanie tesktu
    fclose(strumien); //zamkniecie pliku

    cout<<oz1<<endl; //wyswietlenie odczytanych wartosci
    cout<<oz2<<endl;
    cout<<otekst<<endl;
} else {
    cout<<"Nie mozna odczytac pliku";
}
}

int _tmain(int argc, _TCHAR* argv[])
{
    zapis1();
    cout<<endl<<"-----"<<endl<<endl;
    odczyt1();
    cout<<endl;
    _getch();
    return 0;
}

```

---

## Rozwiązanie 2

---

```

#include "stdafx.h"
#include <conio.h>
#include <iostream>
#include <fstream> //obsługa strumieni plikowych
using namespace std;

string nazwa2="plik2.dat"; //nazwa pliku

void zapis2(){
    fstream strumien; //obiekt obsługi pliku
    int zz1=4; //pierwsza zmienna do zapisania
    double zz2=3.14; //druga zmienna do zapisania
    char ztekst[10]={0}; //tekst do zapisania
    size_t rt; //rozmiar tekstu

    strcpy(ztekst,"-TEXT-\0"); //inicjalizacja tekstu

    //otwarcie pliku binarnego do zapisu
    strumien.open(nazwa2.c_str(),ios::out|ios::binary);
    if (strumien.is_open()){ //zapis kolejnych zmiennych
        strumien.write(reinterpret_cast<char*>(&zz1) ,sizeof(zz1));
        strumien.write(reinterpret_cast<char*>(&zz2) ,sizeof(zz2));
        rt=strlen(ztekst); //sprawdzenie dlugosci tekstu
        strumien.write(reinterpret_cast<char*>(&rt) ,sizeof(rt) );
        strumien.write(reinterpret_cast<char*>(&ztekst),rt );
        strumien.close(); //zamkniecie pliku
        cout<<"Zapisano plik "<<nazwa2.c_str()<<endl;
    } else {
        cout<<"Nie mozna utworzyc pliku";
    }
}

```

```
void odczyt2(){
    fstream strumien;           //obiekt obsługi pliku
    int    oz1;                 //bufor dla pierwszej odczytanej wartosci
    double oz2;                 //bufor dla drugiej odczytanej wartosci
    char   otekst[10]={0};      //bufor dla odczytanego tekstu
    size_t rt;                  //bufor dla odczytanego rozmiaru tekstu

                                //otwarcie pliku binarnego do odczytu
    strumien.open(nazwa2.c_str() ,ios::in|ios::binary);
    if (strumien.is_open()){//odczyt kolejnych zmiennych
        strumien.read(reinterpret_cast<char *>(&oz1) , sizeof(oz1));
        strumien.read(reinterpret_cast<char *>(&oz2) , sizeof(oz2));
        strumien.read(reinterpret_cast<char *>(&rt) , sizeof(rt) );
        strumien.read(reinterpret_cast<char *>(&otekst),rt );
        strumien.close();       //zamknięcie pliku

        cout<<"Odczytana zawartosc pliku "<<nazwa2.c_str()<<endl;
        cout<<oz1<<endl;
        cout<<oz2<<endl;
        cout<<otekst<<endl;
    } else {
        cout<<"Nie mozna odczytac pliku";
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    zapis2();
    cout<<endl;
    cout<<"-----"<<endl<<endl;
    odczyt2();
    cout<<endl;
    _getch();
    return 0;
}
```

---

Uruchom wybraną przez siebie wersję programu.

## 2.25 Ćwiczenie 25 – Operacje na plikach

Napisz program, który odczyta z pliku binarnego pewną tablicę danych o 4 kolumnach

- 1-sza kolumna to liczba całkowita, 2. bajtowa (*short int*)
- 3 kolejne kolumny to liczby rzeczywiste, 8. bajtowe (*double*)

Program powinien odczytać plik i podać ile wierszy zawiera tabela danych w nim zawarta oraz określić dla każdej kolumny: wartość maksymalną, minimalną, średnią oraz odchylenie standardowe.

### Uwagi:

- Dane w pliku zapisane są w ten sposób, że najpierw zapisany jest pierwszy wiersz tabeli danych, następnie drugi, trzeci, itd.
- Dane nie mogą być czytane do zmiennej tablicowej (nawet dynamicznej), ponieważ nie wiadomo jak dużo jest ich w pliku, a w ogólności może to być np. kilka TB danych. Należy tak zorganizować czytanie danych i obliczenia, by nie trzeba było wczytywać całego pliku do pamięci. Oznacza to, że odczytywać należy po jednym wierszu danych.
- Ponieważ nie wiemy ile danych zawiera plik, nie można tu użyć pętli *for*. Zamiast niej należy zastosować pętlę *do-while*, której warunkiem będzie test osiągnięcia końca pliku. Do sprawdzania, czy osiągnięto koniec pliku służy funkcja *feof()* lub metoda *eof()* z klasy *fstream*. Pętla ta będzie miała postać:

```
do {
    //odczyt porcji danych
    if (!feof(strumien)){
        //przetworzenie porcji danych, pod warunkiem,
        //ze nie osiagnieto konca pliku
    }
} while (!feof(strumien));
```

lub

```
do {
    //odczyt porcji danych
    if (!strumien.eof()){
        //przetworzenie porcji danych, pod warunkiem,
        //ze nie osiagnieto konca pliku
    }
} while (!strumien.eof());
```

- Estymator nieobciążony odchylenia standardowego jest opisany zależnością:

$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}$ . W tej postaci jest ona jednak nieprzydatna, gdyż przy sumowaniu wymaga znajomości wartości średniej  $\bar{x}$ , którą znać będziemy dopiero po przejrzaniu całego pliku. W związku z tym, należy tę zależność zmodyfikować tak, by była możliwość obliczania jej składników na bieżąco:

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2} = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i^2 - 2x_i\bar{x} + \bar{x}^2)} = \sqrt{\frac{1}{n-1} \left( \sum_{i=1}^n x_i^2 - 2\bar{x} \sum_{i=1}^n x_i + n\bar{x}^2 \right)}$$

Zauważmy, że w tej postaci na bieżąco należy obliczać tylko sumę kolejnych wartości (którą i tak należy obliczyć w celu późniejszego wyznaczenia średniej) oraz sumę kwadratów kolejnych wartości (tylko te dwie wielkości są zależne od  $i$ ). Natomiast po zakończeniu przeglądania pliku, wartości średnie oraz liczba elementów  $n$  będą już znane i będzie można je podstawić do wzoru.

Pliki dane1.bin, dane2.bin oraz dane3.bin z przykładowymi danymi pobierz ze strony [www](#) z materiałami dydaktycznymi. Pisząc program najpierw spraw, by poprawnie działał z plikiem dane1.bin (plik ten jest krótszy więc testowanie programu będzie łatwiejsze), a następnie sprawdź jego działanie na drugim pliku.

Na etapie testowania programu przydatna może być informacja, iż w obu plikach, pierwsza kolumna danych zawiera kolejne liczby całkowite, można więc je traktować pomocniczo jako numer wiersza w oryginalnej tablicy danych.

Tworzenie programu warto rozpocząć od napisania programu, który po prostu wyświetli dane odczytane z pliku i dopiero w dalszych krokach modyfikować go do postaci docelowej, realizującej postawione w ćwiczeniu zadanie.

Wyniki dla pliku plik1.bin

Kolumna	1	2	3	4
Min	0,000000	-0,998063	0,000861	0,000000
Średnia	9,500000	-0,001537	1,013001	1,235000
Max	19,000000	0,999785	2,000000	3,610000
Odchylenie std	5,916080	0,720787	0,730012	1,164431

Wyniki dla pliku plik2.bin

Kolumna	1	2	3	4
Min	0,000000	-3,073745	-2,792194	-5,493841
Średnia	1249,500000	0,015448	-0,019803	-3,979581
Max	2499,000000	3,412755	2,933931	-2,500397
Odchylenie std	721,832160	0,998613	0,715790	0,786501

Wyniki dla pliku plik3.bin

Kolumna	1	2	3	4
Min	0,000000	-9,999541	-10,000000	-13,024977
Średnia	499,500000	-8,995037	6,666650	-6,489613
Max	999,000000	0,000000	15,000000	-0,620078
Odchylenie std	288,819436	2,011035	7,457327	3,671724

## 2.26 Ćwiczenie 26 – Typ strukturalny

### 2.26.1 Ćwiczenie 26a

Utwórz nowy projekt i napisz program, który utworzy mini rejestr studentów (3 rekordy). Każdy opis ma zawierać Nazwisko, Imię, numer grupy i ocenę. Przy wczytywaniu danych użyj pętli. Po wczytaniu danych program powinien zapytać o numer rekordu do wyświetlenia i wyświetlić jego zawartość. Dane należy umieszczać w tablicy 3-elementowej. Każdy element tablicy to struktura o określonych polach. Rozwiązanie podano poniżej:

```
#include "stdafx.h"
#include <conio.h>
#include <iostream>
using namespace std;

struct student
{
    char    imie[30], nazwisko[50];
    int     grupa;
    float   ocena;
} ;

student Tab_studenci[3]; //zmienna globalna, tablica struktur

int _tmain(int argc, _TCHAR* argv[])
{
    int i;
    int nr_rek;

    for (i=0;i<3;i++){ //wypełnianie rekordów danymi
        cout<<"Podaj imie: ";
        cin>>Tab_studenci[i].imie;
        cout<<"Podaj Nazwisko: ";
        cin>>Tab_studenci[i].nazwisko;
        cout<<"Podaj numer grupy: ";
        cin>>Tab_studenci[i].grupa;
        cout<<"Podaj ocene (liczba): ";
        cin>>Tab_studenci[i].ocena;
        cout<<endl;
    }
    cout<<endl;
    cout<<"Podaj numer rekordu [0-2]: ";
    cin>>nr_rek;
```

```

cout<<Tab_studenci[nr_rek].imie<< →
                                → " "<<Tab_studenci[nr_rek].nazwisko<<endl;
cout<<"grupa "<<Tab_studenci[nr_rek].grupa<< →
                                → <<" , ocena: "<<Tab_studenci[nr_rek].ocena;

_getch();
return 0;
}

```

Uruchom napisany program.

### 2.26.2 Ćwiczenie 26b

Wykorzystując wiadomości z ćw. 11, 24 i 26a napisz program, który będzie umożliwiał:

- Wpisanie danych studenta (dodanie jednej osoby, maksymalnie 20 osób)
- Wyświetlenie danych studenta o podanym numerze
- Wyświetlenie nazwisk wszystkich studentów z podanej grupy
- Wyświetlenie średniej oceny wszystkich studentów
- Wyświetlenie studentów z zadanej grupy, którzy mają zadaną ocenę
- Da możliwość zapisu danych do pliku
- Da możliwość odczytu danych z pliku
- Opcja dodatkowa zadania: Ustalcie z koleżanką / kolegą pracującą / pracującym, na innym komputerze wspólny format pliku danych (sposób i kolejność zapisu danych w pliku). Zapiszcie swoje pliki, a następnie wymieńcie się nimi i wczytajcie nowe dane do swoich programów.

Program powinien działać tak długo, aż użytkownik zechce go zakończyć (tak jak w ćw. 11). Poszczególne zadania powinny być realizowane przez osobne funkcje. Można założyć, że studentów będzie max. 20.

#### Uwagi:

- W przypadku zapisu do pliku zmiennej strukturalnej można albo zapisywać każdy składnik struktury osobno, albo zapisać całą strukturę od razu, tak jak zwykłą zmienną. Należy mieć jednak na uwadze, że w przypadku, gdy elementami struktury są tablice, zapis całej struktury będzie oznaczał zapisanie całych tablic, niezależnie od ich faktycznego wypełnienia danymi / tekstem (patrz rozważania w ćw. 24).
- Na początku pliku należy zapisać liczbę studentów, których dane znajdują się w pliku. Nie należy zapisywać do pliku pustych struktur / rekordów nie zawierających danych studentów. W szczególności oznacza to, że nie należy zapisywać do pliku całej tablicy *Tab\_studenci*.

## 2.27 Ćwiczenie 27 – Klasy i obiekty

### 2.27.1 Ćwiczenie 27a

Utwórz nowy projekt i napisz program, który będzie zawierać strukturę opisującą pewien pojazd. Struktura ta powinna zawierać pola opisujące: liczbę pasażerów, aktualny poziom paliwa, zużycie paliwa na 100 km. Na podstawie tych danych program powinien wyliczyć, na jaką odległość da się przewieźć pasażerów. Rozwiązanie podano poniżej:

---

```

#include "stdafx.h"
#include <conio.h>
#include <iostream>
using namespace std;

```



```
struct Pojazd          //deklaracja typu strukturalnego pojazd
{
    string  Nazwa;
    int     LPas;
    float   PozPaliwa;   //aktualny poziom paliwa w litrach
    float   ZuzyciePaliwa; //zuzycie litrow paliwa na 100km
};

int _tmain(int argc, _TCHAR* argv[])
{
    Pojazd osobowy1;      //utworzenie zmiennej strukturalnej typu Pojazd

    osobowy1.Nazwa="Merc";
    osobowy1.LPas=4;
    osobowy1.PozPaliwa=50;
    osobowy1.ZuzyciePaliwa=9.5;

    cout<<"Osobowy 1 przewiezie "<<osobowy1.LPas;
    cout<<" os, na odleglosc ";
    cout<<osobowy1.PozPaliwa/osobowy1.ZuzyciePaliwa*100<<" km";

    _getch();
    return 0;
}
```

---

Uruchom napisany program.

### 2.27.2 Ćwiczenie 27b

W programie z ćw. 27a zamień strukturę na klasę *CPojazd* z elementami publicznymi.

```
class CPojazd          //deklaracja klasy pojazd
{
public:
    string  Nazwa;
    int     LPas;
    float   PozPaliwa;   //aktualny poziom paliwa w litrach
    float   ZuzyciePaliwa; //zuzycie litrow paliwa na 100km
};
```

---

Pozostały kod programu nie ulegnie zmianie z wyjątkiem linii deklaracji zmiennej *osobowy1* gdzie typ strukturalny *Pojazd* należy zastąpić klasą *CPojazd*.

Uruchom napisany program.

### 2.27.3 Ćwiczenie 27c

W programie z ćw. 27b uzupełnij deklarację klasy pojazd o publiczne metody:

- Konstruktor *CPojazd()*
- Metodę *setLadunek()* która zwróci tekst opisujący liczbę i rodzaj ładunku pojazdu
- Metodę *ObliczZasieg()* zwracającą wyliczony zasięg pojazdu

Deklaracja klasy powinna wyglądać tak jak poniżej:

```
class CPojazd          //deklaracja klasy pojazd
{
public:
    string  Nazwa;
    int     LPas;
    float   PozPaliwa;   //aktualny poziom paliwa w litrach
    float   ZuzyciePaliwa; //zuzycie litrow paliwa na 100km
private:
    float   Droga;
public:
    CPojazd();
    string  getLadunek();
}
```

```
float ObliczZasieg();  
};
```

Na podstawie deklaracji napisz kod poszczególnych metod wymienionych w klasie. Np.:

- Konstruktor klasy:

```
CPojazd::CPojazd() {  
}
```

- Metoda *getLadunek*:

```
string CPojazd::getLadunek(void) {  
    char bufor[20];  
    _itoa(LPas, bufor, 10);           //konwersja liczby na ciąg znakowy  
    return strcat(bufor, " os.");     //dodanie napisu „os.” i zwrócenie całości  
}
```

Napisz brakujący jeszcze kod metody *ObliczZasieg*. Zasięg powinien być uzależniony od liczby pasażerów w ten sposób, że zużycie paliwa rośnie o 5% dla każdego kolejnego pasażera. Pamiętaj o rzutowaniu zmiennych całkowitych na typ *double* przy wykonywaniu obliczeń (w szczególności dzielenia).

W kodzie programu, w części, w której wyświetlany jest komunikat o zasięgu pojazdu użyj odpowiednich metod zamiast wprost odwoływać się do właściwości obiektu. Pamiętaj, że aby wyświetlić z użyciem strumieni zawartość ciągu tekstowego ze zmiennej obiektowej typu *string* należy wywołać metodę *c\_str*. Np.: `cout<<osobowy1.Nazwa.c_str();`

Uruchom napisany program.

#### 2.27.4 Ćwiczenie 27d

W programie z ćw. 27c uzupełnij klasę o przeciążony publiczny konstruktor o deklaracji:

```
CPojazd(string NInic, int LPInic, float PPInic, float ZPInic);
```

Napisz kod konstruktora, a następnie użyj go w celu zdefiniowania 2 dodatkowych zmiennych obiektowych klasy *pojazd*:

```
CPojazd osobowy2("Fiacik", 2, 45, 6);  
CPojazd bus1("Solarek", 52, 600, 20);
```

W kodzie programu, dodaj kod wyświetlający komunikaty dotyczące zasięgu dodanych pojazdów.

Uruchom napisany program.

#### 2.27.5 Ćwiczenie 27e

W programie z ćw. 27d dodaj klasę opisującą pojazd ciężarowy jako klasę pochodną od klasy *pojazd*. Klasa taka powinna mieć:

- Dodatkowe pole prywatne *Cargo*
- Publiczną metodę *getLadunek()*, która przeciąży metodę *getLadunek()* z klasy bazowej
- Konstruktor pozwalający zainicjalizować wartości wszystkich pól klasy (w tym także tych odziedziczonych z klasy bazowej)

Deklaracja klasy powinna wyglądać tak jak poniżej:

```
class CPojazdCiezarowy : public CPojazd {  
private:  
    int Cargo; //ladunek w kg;  
public:  
    CPojazdCiezarowy(string NInic, int LPInic, float PPInic, →  
        → float ZPInic, float LInic);  
    string getLadunek();  
    float ObliczZasieg();  
};
```

Kod konstruktora powinien wyglądać następująco:

```
CPojazdCiezarowy::CPojazdCiezarowy(string NInic, int LPInic, →  
                                →float PPInic, float ZPInic, float CInic):→  
                                →CPojazd(NInic, LPInic, PPInic, ZPInic) {  
    Cargo=CInic;  
}
```

---

Taka budowa konstruktora umożliwia przekazanie części parametrów przez konstruktor klasy pochodnej do konstruktora klasy bazowej.

Należy jeszcze dodać kod źródłowy metod *getLadunek* i *ObliczZasieg*.

Kod metody *getLadunek* podany jest poniżej:

```
string CPojazdCiezarowy::getLadunek(void) {  
    char bufor1[50]={0};           //główny bufor  
    char bufor2[10]={0};          //pomocniczy bufor  
    string tekst;                 //końcowy tekst  
    _itoa(LPas, bufor1, 10);  
    _itoa(Cargo, bufor2, 10);  
  
    strcat(bufor1, " os. i ");  
    strcat(bufor1, bufor2);  
    tekst=strcat(bufor1, " kg ładunku");  
    return tekst;  
}
```

---

W metodzie *ObliczZasieg*, wpływ zużycia paliwa będzie inny niż dla innych pojazdów, tzn. zużycie paliwa wzrasta o 1% dla każdego kolejnego pasażera i o 20% na każdą tonę ładunku.

W kodzie programu utwórz dodatkową zmienną obiektową

```
CPojazdCiezarowy ciezarowy1("Manik", 1, 600, 25, 10000);
```

oraz dodaj kod wyświetlający komunikaty dotyczące zasięgu dodanego pojazdu.

Uruchom napisany program i przetestuj jego działanie.

## 2.27.6 Ćwiczenie 27f

W programie z ćw. 27e zadeklaruj tablicę obiektów klasy *CPojazd*:

```
CPojazd *TabPojazdow[4];
```

a następnie przypisz wskaźniki do utworzonych wcześniej obiektów do kolejnych pól w tablicy, np.:

```
TabPojazdow[0]=&osobowy1;
```

Po wypełnieniu tablicy wyświetl w pętli tekst zawierający nazwę pojazdu, jego ładunek i zasięg. (Nie usuwaj wcześniej napisanego fragmentu kodu odpowiedzialnego za wyświetlanie tego opisu – będzie potrzebny do porównania rezultatów uzyskanych dwiema metodami)

```
int i;  
for (i=0; i<4; i++){  
    cout<<TabPojazdow[i]->Nazwa.c_str() <<" przewiezie ";  
    cout<<TabPojazdow[i]->getLadunek().c_str() <<" na odleglosc ";  
    cout<<TabPojazdow[i]->ObliczZasieg()<<" km"<<endl;  
}
```

---

Uruchom program.

Zauważ, że przy takiej realizacji wyświetlania tekstu, informacje o samochodzie ciężarowym nie są pełne (brak informacji o ładunku), a podany zasięg nie zgadza się z podanym wcześniej, co oznacza, że zależy tylko od liczby pasażerów, a nie zależy od ładunku. Dzieje się tak ponieważ przy deklaracji tablicy podano, że ma zawierać wskaźniki do obiektów *CPojazd*. Wskaźnik taki może wskazywać jednak zarówno obiekt klasy *CPojazd* jak i obiekt klasy pochodnej od *CPojazd* (czyli *CPojazdCiezarowy*). Kompilator nie wie, które obiekty będą wskazywane w czasie pracy programu przez wskaźniki z tablicy, w związku z tym, gdy napotyka na wywołanie metod *getLadunek* oraz *ObliczZasieg* zakłada, że wywołane mają być

metody z klasy bazowej. Aby rozwiązać ten problem, w deklaracji klasy bazowej przed nazwami tych metod dopisz słowo `virtual`. Dzięki temu kompilator pozostawi programowi decyzję o wywołaniu odpowiedniej metody w trakcie jego działania.

Uruchom zmodyfikowany kod i sprawdź, czy opis pojazdu ciężarowego wykonany w pętli zgadza się z opisem wyświetlanym wcześniej.

Okno programu powinno prezentować informacje jak poniżej:

---

```
Merc przewiezie 4 os., na odleglosc 438.596 km
Fiacik przewiezie 2 os., na odleglosc 681.818 km
Solarek przewiezie 52 os., na odleglosc 833.333 km
Manik przewiezie 1 os. i 10000 kg ladunku, na odleglosc 797.342 km
```

```
Merc przewiezie 4 os., na odleglosc 438.596 km
Fiacik przewiezie 2 os., na odleglosc 681.818 km
Solarek przewiezie 52 os., na odleglosc 833.333 km
Manik przewiezie 1 os. i 10000 kg ladunku, na odleglosc 797.342 km
```

---

### 2.27.7 Ćwiczenie 27g

Korzystając z elementów napisanych w poprzednich podpunktach ćw. 27 napisz program, który na początku zapyta o stan paliwa 6 różnych pojazdów (w tym minimum 2 ciężarowe), a następnie (w pętli) będzie pytał o przebytą drogę. Na podstawie informacji o aktualnym stanie paliwa, przebytej w danym cyklu drodze i zużyciu paliwa pokaże:

- Całkowitą przebytą drogę
- Stan paliwa poszczególnych pojazdów
- Określi, czy dany pojazd może jeszcze dalej jechać

Program ma powtarzać pętlę tak długo, aż zabraknie paliwa we wszystkich pojazdach.

Konieczne będzie dodanie do klasy pojazd dodatkowej zmiennej o nazwie *Droga*. Przyjmijmy, że zmienna ta powinna być prywatna. Jej wartość początkowa powinna być inicjalizowana w konstruktorze klasy *CPojazd*. Ponadto, należy uzupełnić klasę o dwie publiczne metody `float getDroga();` oraz `float Przejedz(float);`. Pierwsza z nich powinna zwracać aktualnie przebytą przez pojazd drogę, zaś druga zwiększać przebytą drogę o zadaną wartość i również ją zwracać.

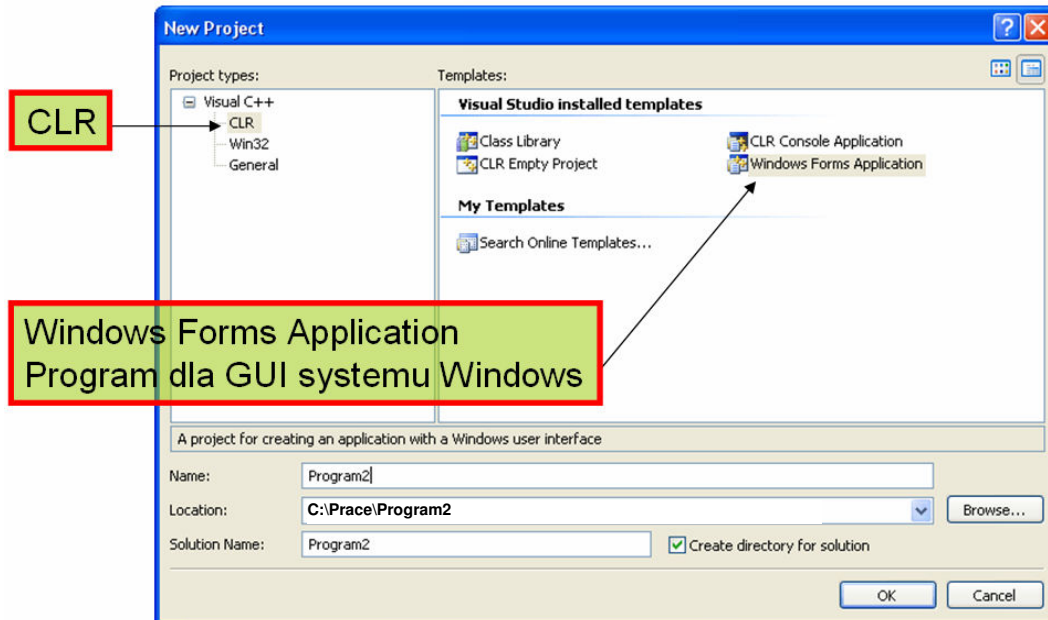
Uruchom napisany program.

### 3 Aplikacje dla GUI systemu MS Windows – C++/CLI

#### 3.1 Tworzenie nowego projektu

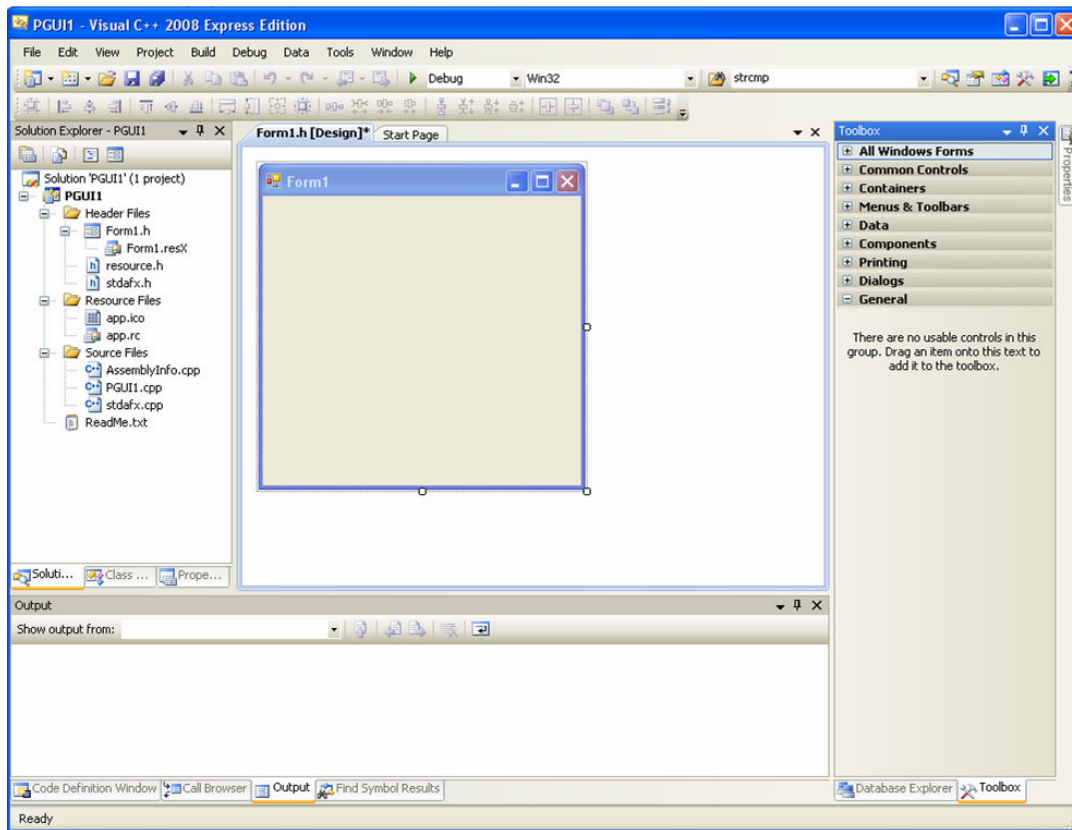
Z Menu wybierz File > New > Project...

W oknie kreatora projektu wybierz „CLR” > „Windows Forms Application”, a następnie podaj nazwę projektu (Rys. 11)



Rys. 11

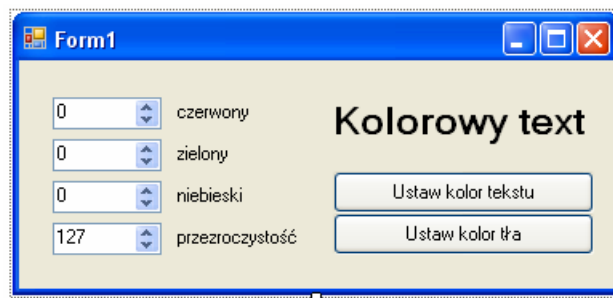
Po kliknięciu OK. pojawi się pusty formularz automatycznie wygenerowanego programu (Rys. 12).



Rys. 12

### 3.2 Ćwiczenie 28 – Prosty program dla GUI Windows – „Kolorowy tekst”

Utwórz nowy projekt dla GUI. Na początku zaprojektuj formularz programu, tak jak na Rys. 13.



Rys. 13

Potrzebne będą następujące kontrolki:

- *NumericUpDown*
- *Label*
- *Button*

Dla kontrolki *Label* i *Button* ustaw odpowiednie napisy we właściwości *Text*. Wartości właściwości ustawia się w oknie *Properties* po zaznaczeniu kontrolki, której właściwości chcemy zmienić. Jeśli okno to nie jest otwarte, kliknij na kontrolce prawym klawiszem myszy i wybierz polecenie *Properties*. W przypadku kontrolki *Label* zawierającej tekst „Kolorowy tekst” należy zmienić rozmiar czcionki na 18 i włączyć jej pogrubienie. W tym celu w oknie edytora właściwości rozwiń właściwość *Font* i zmień wartość *Size* na 18 i *Bold* na *True*. Zmień także nazwę tej kontrolki (właściwość *Name*) na „labelTekst”. Dla pól

*NumericUpDown* ustaw dopuszczalne wartości minimalne (0) i maksymalne (255). Dla kontrolki, która będzie odpowiedzialna za przezroczystość ustaw wartość początkową 127 (właściwość *Value*).

Następnie ustaw metodę, która będzie wywoływana dla zdarzenie przyciśnięcia przycisku „Ustaw kolor tekstu”. W tym celu w oknie *Properties* przejdź do zakładki *events* (🔗) i przypisz zdarzeniu *Click* metodę *button1\_Click*. Metoda jest tworzone automatycznie po podwójnym kliknięciu przy odpowiednim zdarzeniu w zakładce *events*. W podobny sposób przypisz metodę *button2\_Click* do drugiego przycisku. Uzupełnij kod metod:

```
private: System::Void button1_Click(System::Object^ sender, →
                                   → System::EventArgs^ e) {
    int a,r,g,b;

    a=static_cast<int> (numericUpDown4->Value);
    r=static_cast<int> (numericUpDown1->Value);
    g=static_cast<int> (numericUpDown2->Value);
    b=static_cast<int> (numericUpDown3->Value);
    labelTekst->ForeColor=Color::FromArgb(a,r,g,b);
}
```

W powyższym kodzie założono, że pole *numericUpDown1* jest pierwszym od góry i odpowiada za składową czerwoną koloru. Kolejne pola mają kolejne numery i odpowiadają za składową zieloną i niebieską oraz za przezroczystość.

Kod metody *button2\_Click* będzie różnił się tylko jedną linią:

```
labelTekst->BackColor=Color::FromArgb(a,r,g,b);
```

Uruchom napisany program. Zauważ, że w przypadku koloru czcionki zmiana przezroczystości nie zmienia koloru, natomiast w przypadku tła – zmienia. Uwaga: wartości należy zmieniać o duże wartości by zauważyć efekty.

### 3.3 Ćwiczenie 29 – Prosty program dla GUI Windows – „Kalkulator czasu”

Utwórz nowy projekt dla GUI i napisz program „Kalkulator czasu” pozwalający dodawać do siebie i odejmować godziny, minuty i sekundy.

Interfejs programu powinien wyglądać jak na Rys. 14, na którym pokazano wybrane przypadki wyglądu okna programu w różnych sytuacjach.



Rys. 14

Wygląd okna ma być taki jak na rysunkach tzn. zachowane mają być pozycje elementów, ich rozmiary, kolory, czcionki.



Podczas pisania kodu przydatne będą następujące informacje i odpowiedzi:

- Wygląd interfejsu
  - Poszczególne fragmenty wyniku najlepiej umieszczać w osobnych kontrolkach typu *label*. Oprócz 3 elementów *label* przeznaczonych do wyświetlania godzin, minut i sekund potrzebne też będzie pole do wyświetlania znaku „-” oraz pola do wyświetlania znaku „:”.
  - W polach *label* odpowiedzialnych za wyświetlanie wyniku użyta jest czcionka o rozmiarze 12 pt.
  - Właściwość *AutoSize* pól *label* powinna być wyłączona, a rozmiar pól powinien być tak dobrany, by poprawnie wyświetlić mogły:
    - Pole godzin: 3 znaki
    - Pola minut i sekund: 2 znaki.

Uwaga: Najlepiej najpierw nadać właściwości *Text* wartość „00” lub „000”, a następnie ustawić *AutoSize* na *False*. Wówczas kontrolki automatycznie i na stałe przyjmą właściwe rozmiary.

  - Po dodaniu każdej z kontrolki dobrze jest zmienić jej nazwę na lepiej oddającą przeznaczenie danej kontrolki. Np. zamiast *label1*, *label2*, *label3* itd. lepiej użyć nazw *label\_h*, *label\_m*, *label\_s*. Nazwę zmienić można edytując właściwość *Name* wybranej kontrolki.
  - Dla kontrolki, które można edytować lub klikać ustawić właściwość *TabStop* tak, by można było pomiędzy nimi przechodzić w sensownej kolejności z użyciem klawisza tabulatora. Kolejność przechodzenia najłatwiej jest ustawić korzystając z narzędzie *TabOrder* z menu *View*. Po jego włączeniu wystarczy klikać poszczególne kontrolki na formularzu w odpowiedniej kolejności. Po zakończeniu należy wyłączyć narzędzie ponownie wybierając z menu *View > TabOrder*.- Ograniczenia zakresu danych i ich kontrola
  - Zakresy dopuszczalnych wartości kontrolki *numericUpDown* powinny być:
    - Pole godzin: <0; 999>
    - Pola minut i sekund: <0; 60>.
  - Zakresy danych wejściowych powinny być kontrolowane poprzez określenie odpowiednich wartości właściwości kontrolki *numericUpDown*.
  - Wynik nie powinien przekraczać wartości +/-999h59m59s, jeśli po operacji wartość wynikowa przekroczy ten, zakres program powinien wyświetlić okno z ostrzeżeniem i powrócić do wyniku sprzed operacji
- Wyświetlanie okna komunikatu
  - Do wyświetlania okna komunikatu służy metoda: *MessageBox::Show*(„Komunikat”, „Tytuł okna”, specyfikacja przycisków, specyfikacja ikonki). Np.:  

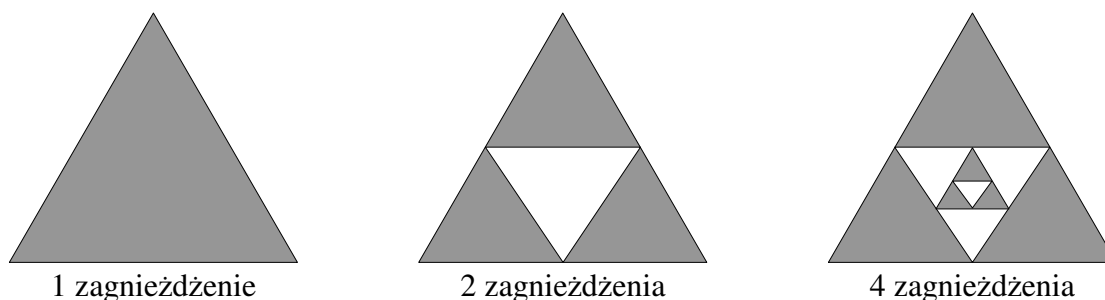
```
MessageBox::Show("Przekroczenie zakresu", "Przekroczenie zakresu", MessageBoxButtons::OK, MessageBoxIcon::Exclamation);
```
- Przechowywanie wyniku
  - Kiedy w programie chcemy wykonywać operacje związane z przechowywaniem i obliczeniami zależności czasowych można wykorzystać klasę *DateTime*. Niestety, w naszej sytuacji nie będzie ona przydatna. Klasa ta wymaga podania konkretnej daty i godziny – nas interesuje zaś pewien abstrakcyjny czas, nie związany z konkretną godziną ani tym bardziej konkretnym dniem z kalendarza. Ponadto, klasa *DateTime* automatycznie koryguje datę, jeśli np. po dodaniu pewnej liczby godzin do aktualnej godziny liczba godzin przekracza 24, data jest zmieniana na następny dzień. W naszym programie liczba godzin może być większa od 24.



- Mając na uwadze rozważania z poprzedniego podpunktu, aktualny wynik obliczeń najwygodniej przechowywać w postaci liczby sekund, na zmiennej globalnej typu *int*. Zadeklarować ją można jako składnik klasy *Form1*:  
`private: int czas;`
  - Podpowiedź: deklaracja klasy *Form1* zaczyna się od linii:  
`public ref class Form1 : public System::Windows::Forms::Form`
- Inicjalizację wartości tej zmiennej najlepiej umieścić w kodzie konstruktora klasy *Form1*.
  - Podpowiedź: poszukaj w kodzie programu linii:  
`//TODO: Add the constructor code here'`  
i umieść inicjalizację wartości zmiennej *czas* zaraz za nią.
- Konwersja typów danych
  - Właściwość *Value* kontrolki *numericUpDown* ma typ *Decimal*. Jest to typ przeznaczonych dla liczb stałoprzecinkowych. Do jego konwersji na typ *int* można użyć metod z klasy *Convert*, np.:  
`wartosc=Convert.ToInt32(numericUpDown1->Value)`
  - Do konwersji z typu *int* na *String* najlepiej użyć metody *ToString()* z klasy *int*, np.: `tekst=godziny.ToString()`  
Metoda ta pozwala także podać format konwersji. Na przykład, aby liczba po konwersji była uzupełniana zerami z przodu można wywołać metodę tak jak poniżej:  
`tekst=minuty.ToString("00")`
- Obliczenia
  - Jeśli wynik jest ujemny, przed wynikiem powinien być wyświetlony znak „-”
  - Podczas obliczeń przydatne będą metody:
    - Zaokrąglenie w stronę 0:  
`System::Math::Truncate()`
    - Wartość bezwzględna:  
`System::Math::Abs()`
  - Rzutowanie typów:  
`static_cast<System::Double>(zmienna)`
  - Obliczenia najlepiej wykonywać w metodach *button\_Click()* odpowiednich przycisków.
- Wyjście z aplikacji
  - Metoda *Application::Exit()*;

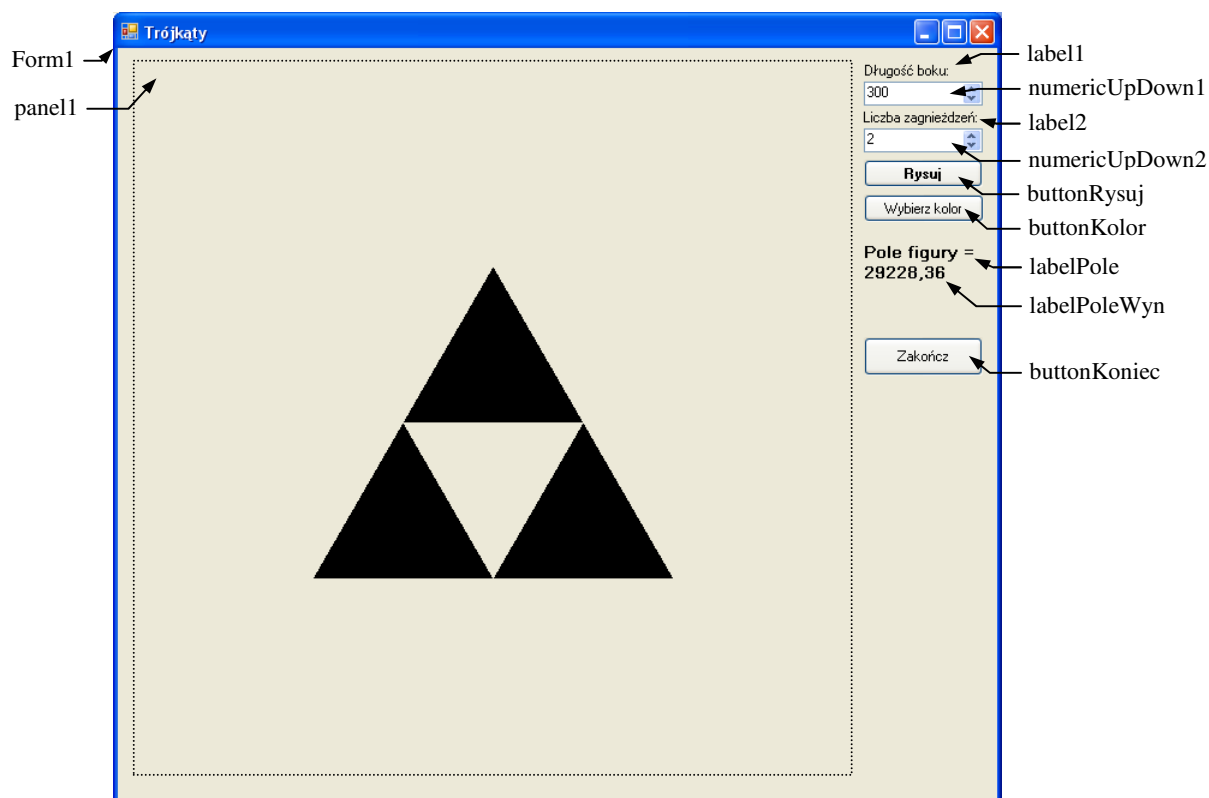
### 3.4 Ćwiczenie 30 – Obsługa prostej grafiki – „Trójkąty”

Utwórz nowy projekt dla GUI i napisz program „Trójkąty” rysujący figurę jak na Rys. 15 i obliczający jej pole. Figura ta budowana jest w ten sposób, że umieszcza się kolejne trójkąty jeden w drugim. Trójkąty skierowane w dół „wycinają” część pola z trójkąta skierowanego w górę. Użytkownik ma mieć możliwość zdania długości boku w pikselach (10 ~ 600 pikseli), liczby zagnieżdżeń (1 ~ 10) i koloru figury. Oprócz narysowania figury program powinien obliczyć także jej pole.



Rys. 15

Interfejs programu powinien wyglądać jak na Rys. 16. Na rysunku podane są także nazwy, jakie należy nadać poszczególnym kontrolkom poprzez ustawienie właściwości *Name*.  
**Uwaga: ustaw nazwy dokładnie takie, jak pokazano.** Jeśli kontrolki będą nazywać się inaczej, elementy kodu źródłowego podane w dalszej części ćwiczenia nie będą działać.



Rys. 16

Uwaga: panel jest niewidoczny w czasie działania programu, jego granice widać tylko w edytorze GUI.

Oprócz widocznych wyżej kontrolek, umieść na formularzu także kontrolkę *colorDialog*. Będzie ona widoczna pod formularzem. Jest ona odpowiedzialna za wyświetlenie dodatkowego okienka, w którym użytkownik będzie miał możliwość wybrania koloru figury.

Ustawienia właściwości poszczególnych kontroltek oraz znaczenie niektórych z nich podane jest poniżej:

Kontrolka	Właściwość	Wartość do ustawienia	Uwagi dodatkowe
Form1	Text	Trójkąty	Tytuł okna programu
	Size	740; 660	Rozmiar okna
	MaximumSize	740; 660	Minimalny i maksymalny rozmiar okna, ustawienie obu spowoduje, że okno zawsze będzie miało stały rozmiar
	ManimumSize	740; 660	
	MaximizeBox	False	Wyłączenie wyświetlania przycisku maksymalizacji okna
panel1	Size	600; 600	Rozmiar panelu
label1	Text	Długość boku:	
label2	Text	Liczba zagnieżdżeń:	
numericUpDown1	Value	300	Wartość początkowa
	Maximum	600	
	Minimum	10	
labelPole	Text	Pole figury =	
	Font.Size	10	
	Font.Bold	True	
numericUpDown2	Value	2	Wartość początkowa
	Maximum	10	
	Minimum	1	
buttonRysuj	Text	Rysuj	
	Font.Bold	True	
labelPoleWyn	Text	0	
	Font.Size	10	
	Font.Bold	True	
buttonKolor	Text	Wybierz kolor	
buttonKoniec	Text	Zakończ	

Do kontroltek typu button przypisz obsługę zdarzeń *Click*:

- *buttonRysuj\_Click*
- *buttonKolor\_Click*
- *buttonKoniec\_Click*

Kod źródłowy tych metod znajduje się poniżej:

```
private: System::Void buttonRysuj_Click(System::Object^ sender, →
                                     →System::EventArgs^ e) {
    panel1->Refresh();
}

private: System::Void buttonKolor_Click(System::Object^ sender, →
                                     →System::EventArgs^ e) {
    colorDialog1->ShowDialog();
    panel1->Refresh();
}

private: System::Void buttonKoniec_Click(System::Object^ sender, →
                                     →System::EventArgs^ e) {
    Application::Exit();
}
```

Uwagi dodatkowe i wyjaśnienia do kodu źródłowego metody *buttonRysuj\_Click*.

- Metoda ta wywołuje wewnętrznie metodę *Refresh()* obiektu *panel1*, która odpowiedzialna jest za odświeżenie panelu. Z kolei metoda *Refresh()* wywołuje metodę *Paint()*, odpowiedzialną za rysowanie elementów graficznych. Kod bezpośrednio odpowiedzialny za rysowanie należy umieścić w metodzie *Paint()* obiektu *panel1*.

Uwagi dodatkowe i wyjaśnienia do kodu źródłowego metody *buttonRysuj\_Click*.

- Metoda ta wywołuje wewnętrznie metodę *showDialog()*, która powoduje otwarcie dodatkowe okienka, w którym użytkownik może wybrać kolor. Wybrany kolor dostępny jest we właściwości *Color* obiektu *colorDialog1*. Po wybraniu koloru należy odświeżyć panel w celu ponownego narysowania figury.

Uwagi dodatkowe i wyjaśnienia do kodu źródłowego metody *buttonKoniec\_Click*.

- Metoda ta wywołuje wewnętrznie metodę *Exit()*, która zamyka program.

Z punktu widzenia rysowania figury najważniejszą metodą jest metoda *panel1\_Paint()*. Należy ją przypisać do obsługi zdarzenia *Paint* kontrolki *panel1*.

### Algorytm rysowania figury.

Jak napisano wcześniej, figura powstaje poprzez rysowanie kolejnych trójkątów, jeden wewnątrz drugiego. Trójkąty „nieparzyste”, skierowane w górę są rysowane zadany kolorem i ich pola są sumowane. Natomiast trójkąty „parzyste”, skierowane w dół, są rysowane kolorem takim samym, jak kolor panelu (stąd wrażenie przezroczystości), a ich pola są odejmowane od całkowitego pola.

Aby narysować trójkąt, konieczne będzie wyznaczenie współrzędnych jego narożników. Potrzebny jest także jakiś układ odniesienia. Przyjmijmy, że chcemy, by figura była rysowana na środku panelu. Przy tych założeniach, na pierwszy rzut oka wydawać się może, że najlepiej będzie wszystkie obliczenia wykonywać względem środka panelu. Problem pojawi się jednak dla trójkątów na 2 i dalszych poziomach zagnieżdżenia gdyż ich środki nie pokrywają się ze środkiem panelu. Lepszym rozwiązaniem będzie wyznaczenie pozycji środka trójkąta, a następnie uzależnienie położenia narożników od tej pozycji środkowej. Tylko dla pierwszego trójkąta pozycja jego środka będzie odpowiadać pozycji środka panelu. Do wyznaczenia współrzędnych narożników potrzebna będzie długość boku trójkąta oraz jego wysokość. Ponieważ współrzędne narożników trójkąta mają być wyznaczone względem środka, potrzebne nam będą także wartości połowy długości boku i wysokości. Dla uproszczenia i przyspieszenia obliczeń wygodniej będzie wyznaczać te wielkości przed dalszymi operacjami i później wykorzystać je zamiast wielokrotnego obliczania połówek długości boków i wysokości.

W metodzie *panel1\_Paint()* zadeklaruj następujące zmienne:

---

```
double bok, wys;           //długość boku i wysokość trójkąta
int     polbok, polwys;     //połowy długości boku i wysokości trójkąta,
                           //zaokrąglone do wartości całkowitych

double pole;
int     stX, stY;           //środek trójkąta, współrzędna X i Y
int     i;
```

---

Początkowe wartości zmiennych można określić w następujący sposób:

---

```
bok=Convert.ToDouble(numericUpDown1->Value); //bok zewnętrznego trójkąta
wys=bok*Math.Sqrt(3)/2;                       //wysokość zewnętrznego trójkąta
stX=panel1->Size.Width/2;                     //wsp. X środka zewnętrznego trójkąta
stY=panel1->Size.Height/2;                    //wsp. Y środka zewnętrznego trójkąta
```

---

Metody rysujące obiekty graficzne wymagają zdefiniowania tzw. pędzla wypełniającego. Jest to obiekt określający m.in. kolor wypełnienia figury. Utworzymy dwa pędzle – jeden o kolorze wybranym przez użytkownika w oknie dialogowym *colorDialog*; drugi o kolorze takim samym jak kolor tła okna programu.

```
SolidBrush^ B1 = gcnew SolidBrush( colorDialog1->Color );//wybrany kolor
SolidBrush^ B2 = gcnew SolidBrush( Form1::BackColor ); //kolor tła okna
```

Do rysowania wypełnionego kolorem trójkąta wykorzystamy metodę *FillPolygon* z klasy *Graphics*. Metoda ta wymaga podania wskaźnika do obiektu pędzla oraz tablicy współrzędnych kolejnych narożników. Jej przykładowe wywołanie ma postać:

```
e->Graphics->FillPolygon(B1,punkty);
```

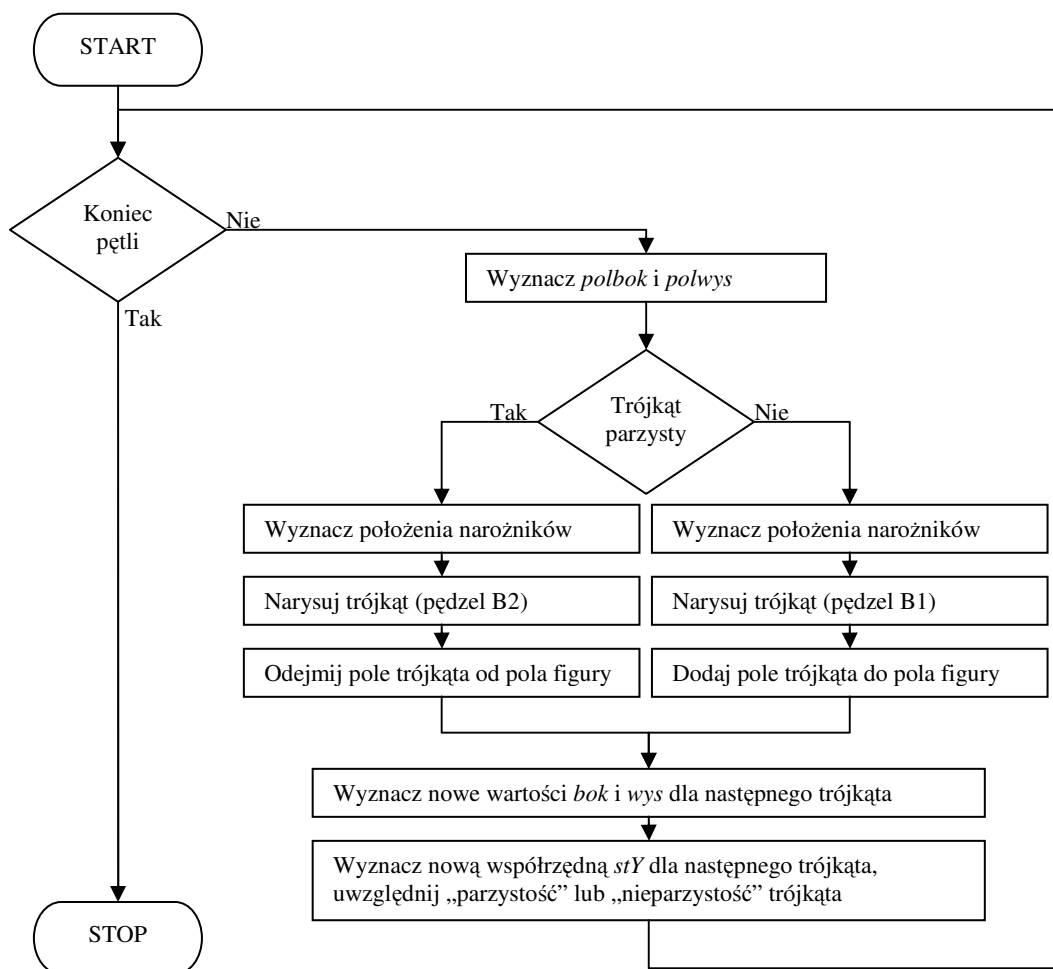
Do zadeklarowania tablicy punktów posłużymy się nadzorowaną tablicą dynamiczną, której elementami są zmienne strukturalne typu *Point*. Deklaracja takiej tablicy, wygląda następująco:

```
array<Point> ^punkty = gcnew array<Point>(3);
```

Tablicę będziemy następnie wypełnić punktami. Np.:

```
punkty[0]=Point(stX-polbok,stY+polwys);
```

Wyznaczanie i rysowanie kolejnych trójkątów należy umieścić w pętli. Algorytm zawarty w pętli ma następujący uproszczony schemat blokowy:



Rys. 17

W kolejnych iteracjach, pozycja X (*stX*) środka trójkąta jest stała, natomiast pozycja Y (*stY*) zmienia się w górę lub w dół (zależnie od tego, czy trójkąt jest „parzysty” czy „nieparzysty”) o połowę wysokości poprzedniego trójkąta.

Pamiętaj, że obliczenia dotyczące długości boków i wysokości należy wykonywać na liczbach zmiennoprzecinkowych w celu zachowania dokładności obliczeń. Natomiast do określania współrzędnych należy użyć liczb całkowitych. Konieczna będzie więc konwersja typów zmiennych. Np.:

---

```
polbok=static_cast<int>(Math::Round(bok/2));
```

---

Wynik obliczeń pola figury wyświetl w kontrolce *labelPoleWyn* z dokładnością do dwóch miejsc po przecinku. Można to zrobić podając odpowiedni format w wywołaniu metody *ToString()*, np.: `pole.ToString("0.00")`

Uzupełnij kod programu o brakujące elementy, uruchom go i przetestuj jego działanie.

### 3.5 Ćwiczenie 31 – Prosta animacja – „Zegar”

#### 3.5.1 Ćwiczenie 31a

Utwórz nowy projekt dla GUI i napisz program „Zegar” wyświetlający zegar analogowy. Oprócz godziny wskazywanej przez wskazówki, zegar na także podawać godzinę i datę w postaci tekstowej.

Formularz programu będzie składał się (oprócz samego formularza) tylko z panelu, na którym będzie rysowany zegar. Ustaw rozmiar okna formularza na 530x530 pikseli. Ustal także minimalny rozmiar formularza na te same wartości. Rozmiar panelu ustaw na 475x475 pikseli. Dla zdarzenia *Paint* panelu utwórz i przypisz metodę *panel1\_Paint*. Na razie pozostaw ją pustą, bez kodu.

Zegar będzie odświeżany co sekundę. Aby uzyskać automatyczne wywoływanie jakiejś operacji co określony czas należy skorzystać z kontrolki *timer*. Dodaj do formularza kontrolkę *timer* i ustaw częstotliwość jej wywoływania (właściwość *Interval*) na 1000 ms. Pamiętaj także o włączeniu timera (właściwość *Enabled* ustaw na *True*). Następnie do zdarzenia *Tick* dodaj metodę *timer1\_Tick*. W kodzie metody wpisz:

---

```
panel1->Refresh();
```

---

Pozwoli to przy każdym „tyknięciu” timera odświeżyć panel. Metoda *Refresh* wywołuje metodę *Paint* i to w niej należy zawrzeć kod odpowiedzialny za rysowanie zegara.

Teoretycznie, skoro wiemy, że *timer* wywołuje zdarzenie co sekundę, można by np. Raz pobrać godzinę początkową, a następnie co sekundę tylko przesuwając wskazówki o odpowiednie wartości. Jest to jednak rozwiązanie złe. Nie ma żadnej gwarancji, że zdarzenie *Tick* jest rzeczywiście zgłaszane co sekundę. W praktyce, dokładność odliczania czasu jest bardzo przybliżona. Już po kilkunastu sekundach może się okazać, że czas zliczany przez nas różni się zauważalnie od rzeczywistego. Znacznie lepszym rozwiązaniem w każdym programie, w którym wykorzystywany jest *timer* i potrzebne jest wykonanie dokładnych operacji na czasie lepiej pobierać aktualny czas systemowy przy każdym wywołaniu timera. Do obsługi czasu wykorzystamy klasę *DateTime*.

Poniższe fragmenty kodu należy umieścić wewnątrz metody *panel1\_Paint*.

Najpierw należy zadeklarować uchwyt do obiektu klasy *DateTime*, następnie utworzyć ten obiekt i pobrać aktualną datę i godzinę.

---

```
DateTime ^DataGodzina;
```

---

```
DataGodzina = gcnew DateTime();
```

```
DataGodzina=DateTime::Now; //aktualna data i godzina
```

---

Zarówno elementy zegara jak teksty najlepiej będzie rysować względem środka panelu. Należy go więc wyznaczyć. Skorzystamy przy tym z typu strukturalnego *PointF*, który pozwala zapisać współrzędne ekranu w postaci zmiennoprzecinkowej. Struktura ta jest przyjmowana jako parametr przez wiele różnych metod związanych z rysowaniem elementów graficznych. Wartości zmiennoprzecinkowe są automatycznie zaokrąglane do całkowitych dopiero w momencie rysowania co pozwala zachować dokładność obliczeń i operować np. na pozycjach w rodzaju 2,3 piksela.

Kolejnym krokiem będzie określenie pozycji środka panelu

---

```
PointF srodek = PointF(static_cast<double>(panel1->Width)/2, →  
→static_cast<double>(panel1->Height)/2);
```

---



Następnie, narysowany zostanie okrąg tarczy zegara. Wcześniej należy jednak utworzyć obiekt pióra pozwalający określić kolor i grubość linii okręgu.

```
//rysowanie okręgu
Pen^ piorol = gnew Pen(Color::Black, 3);
e->Graphics->DrawEllipse(piorol, static_cast<double>(srodek.X-200), →
→static_cast<double>(srodek.Y-200), 400, 400);
```

Kolejnymi elementami, które zostaną narysowane będą godzina i data wyświetlone w postaci tekstowej. Najpierw należy pobrać z obiektu *DataGodzina* odpowiednie informacje i przekształcić je na obiekty typu *String*. Służą do tego metody *ToLongTimeString* oraz *ToLongDateString*. Obie konwertują godzinę i datę do postaci tekstowej z użyciem formatów daty i czasu ustawionych w systemie operacyjnym. Zanim będzie można narysować teksty, trzeba najpierw jeszcze określić czcionkę, wyrównanie tekstu i jego kolor. Do określenia czcionki służy obiekt klasy *Font*. W jego konstruktorze określamy czcionkę (wyberzemy Arial), podajemy rozmiar w punktach oraz styl (wyberzemy pogrubienie *Bold*). Do określenia formatowania tekstu, w tym także jego wyrównania, służy obiekt klasy *StringFormat*. W naszym przypadku właściwość *Alignment* obiektu tej klasy ustawimy na *Center* co pozwoli rysować teksty symetrycznie względem środka zegara. Do określenia koloru tekstu potrzebny będzie obiekt klasy *SolidBrush*. Ostatnim brakującym elementem, jaki jest potrzebny do narysowania tekstu jest określenie jego pozycji. Wykorzystamy w tym celu kolejną zmienną strukturalną typu *PointF*. Tekst zawierający godzinę narysowany zostanie w osi x symetrycznie względem środka zegara, zaś w osi y – o 70 pikseli powyżej środka. Z kolei tekst zawierający datę zostanie narysowany 45 pikseli nad środkiem zegara. Do rysowania tekstu służy metoda *DrawString*, przy wywołaniu której należy podać tekst, czcionkę, pędzel, punkt odniesienia oraz format. W przypadku, gdy tekst ma być wyrównany do prawej, formatu można nie podawać.

```
//rysowanie tekstu z aktualną godziną
String^ godzinaText = DataGodzina->ToLongTimeString(); //tekst z godziną
String^ dataText    = DataGodzina->ToLongDateString();  //tekst z datą

//ustawienia czcionki
System::Drawing::Font^ datagodzFont = →
→ gnew System::Drawing::Font ("Arial",20,FontStyle::Bold )
StringFormat^ format1 = gnew StringFormat(); //ustawienia orientacji napisu
format1->Alignment=StringAlignment::Center;
SolidBrush^ pedzell = gnew SolidBrush(Color::Black);

PointF punkt;
punkt = PointF(srodek.X, srodek.Y-70); //pozycja tekstu z godziną
//narysuj tekst
e->Graphics->DrawString(godzinaText, datagodzFont, pedzell, punkt, format1);

punkt = PointF(srodek.X, srodek.Y-45); //pozycja tekstu z datą
//narysuj tekst
e->Graphics->DrawString(dataText, datagodzFont, pedzell, punkt, format1);
```

Najważniejszym elementem zegara są wskazówki. Pozycja każdej ze wskazówek musi być wyznaczona przy każdym odświeżeniu rysunku zegara, a ich pozycje zależą od aktualnej godziny. Sposób wyznaczania pozycji zostanie pokazany na przykładzie wskazówki sekundnika.

Wskazówka ma punkt początkowy w punkcie środkowym zegara i jest to punkt stały. Pozycję zmienia natomiast druga końcówka wskazówki. Aktualną pozycję drugiego końca sekundnika będziemy przechowywać na zmiennej *sK2* typu *PointF*. Jej składowe *X* i *Y* będą wyznaczone wg wzorów:

$$sK2.X = \cos\left(s \cdot \frac{360 \cdot \pi}{60 \cdot 180} - \frac{\pi}{2}\right) \cdot dlWsk + srodek.X = \cos\left(s \cdot \frac{\pi}{30} - \frac{\pi}{2}\right) \cdot dlWsk + srodek.X$$

$$sK2.Y = \sin\left(s \cdot \frac{360 \cdot \pi}{60 \cdot 180} - \frac{\pi}{2}\right) \cdot dlWsk + srodek.Y = \sin\left(s \cdot \frac{\pi}{30} - \frac{\pi}{2}\right) \cdot dlWsk + srodek.Y$$

gdzie:  $s$  – aktualna sekunda

$dlWsk$  – długość wskazówki w pikselach (przyjmujemy 190 pikseli).

W programie, wzory te zostaną podzielone na części, ponieważ niektóre elementy są stałe i w celu przyspieszenia obliczeń można ich wartości wyznaczyć tylko raz. Ponadto, występująca we wzorze wartość  $\pi/2$  będzie przydatna do innych obliczeń w dalszej części programu. Obie części składowe wzoru ( $pi30$  oraz  $pi2$ ) zadeklarowane zostaną jako statyczne stałe. Dzięki temu ich wartości będą zachowane pomiędzy kolejnymi wywołaniami metody *Paint* i nie będą ponownie obliczane.

Do narysowania wskazówki trzeba jeszcze określić jej kolor i grubość, do czego służy obiekt klasy *Pen*.

```
//wyznaczanie pozycji sekundnika
PointF sK2; //pozycja drugiego końca wskazówki sekundnika
static const double pi30=Math::PI/30; //stała - kąt obrotu wskazówki na 1 sekundę
static const double pi2=Math::PI/2; //stała - pi/2

sK2.X=Math::Cos(static_cast<double>(DataGodzina->Second)*pi30-pi2)*190+srodek.X;
sK2.Y=Math::Sin(static_cast<double>(DataGodzina->Second)*pi30-pi2)*190+srodek.Y;

Pen^ pioro2 = gcnew Pen(Color::Blue, 3);
e->Graphics->DrawLine(pioro2, srodek, sK2); //rysowanie wskazówki sekundnika
```

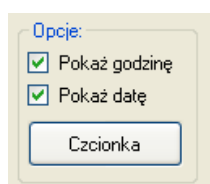
Uruchom program i sprawdź jego działanie.

Uzupełnij kod programu o rysowanie pozostałych wskazówek, liczb oznaczających godziny i innych elementów dekoracyjnych zegara. Pamiętaj, że kolejność rysowania elementów ma znaczenie. Element narysowany jako ostatni jest „na wierzchu” i przysłania elementy narysowane wcześniej.

### 3.5.2 Ćwiczenie 31b

Uzupełnij program o możliwość wyboru czy data oraz godzina mają być wyświetlane, możliwość wyboru czcionki oraz automatyczne skalowanie zegara przy zmianie rozmiaru okna.

W celu umożliwienia wyboru opcji uzupełnij formularz o elementy jak na Rys. 18.



Rys. 18

Aby dać użytkownikowi możliwość wyboru czcionki najlepiej skorzystać ze standardowego, systemowego dialogu wyboru czcionki. W tym celu należy umieścić na formularzu kontrolkę *fontDialog*. Następnie do przycisku „Czcionka” należy dodać metodę obsługi zdarzenia *Click* i umieścić w niej polecenie otwarcia dialogu oraz odświeżenia panelu, na którym rysowany jest zegar.

```
fontDialog1->ShowDialog();
panel1->Refresh();
```

Dla kontrolki *fontDialog* najlepiej ustawić następujące właściwości:



*FontMustExist* = *True* – ogranicza wybór czcionki tylko do czcionek zainstalowanych w danych systemie  
*ShowApply* = *True* – włącza wyświetlanie przycisku „Zastosuj” w oknie dialogowym  
*Font* = *Arial; 18pt; style=Bold* – określa domyślnie wybraną czcionkę  
*MaxSize* = *30* – ogranicza maksymalny rozmiar czcionki, który można wybrać  
*MinSize* = *10* – ogranicza minimalny rozmiar czcionki, który można wybrać

Ponadto, w kodzie metody *panell\_Paint* należy zmienić linię określającą czcionkę wykorzystywaną do rysowania napisów na:

---

```
System::Drawing::Font^ godzinaFont = →  
→gcnew System::Drawing::Font(fontDialog1->Font,FontStyle::Bold );
```

---

Ponieważ zegar ma być skalowany w zależności od rozmiaru okna, należy przy zmianie rozmiaru okna zmieniać także rozmiar panelu, a następnie wszystkie obliczenia (położenia, pozycje, długości, rozmiary kontrole oraz rysowanych elementów) uzależnić od aktualnych rozmiarów panelu. W celu uzyskania dostępu do właściwości formularza *Form1* należy użyć uchwytu *this*. Jest on przekazywany w sposób niejawny do każdej wywoływanej metody. Np. by zmienić szerokość panelu w zależności od szerokości głównego okna programu można to zrobić następująco:

---

```
panell->Width=this->Width-20;
```

---

Inne właściwości kontrolek, które trzeba będzie modyfikować to: *Height* (wysokość) *Location.X* i *Location.Y* (położenie kontrolki względem elementu nadrzędnego).

Wprowadź odpowiednie modyfikacje do kodu, uruchom program i sprawdź jego działanie.

## 4 Komunikacja z relacyjną bazą danych

W przypadku systemu wykorzystującego bazę danych, pierwszym etapem jego budowy jest zaprojektowanie bazy danych oraz wybranie odpowiedniego systemu zarządzania bazą danych (DBMS). Następnie należy DBMS uruchomić – tj. zainstalować i skonfigurować. Kolejnym krokiem jest utworzenie struktury bazy danych i ewentualne wypełnienie jej początkowymi danymi. Tworzenie aplikacji korzystającej z bazy danych prowadzone jest po uruchomieniu bazy danych lub równolegle z tym procesem (głównie w przypadku zaawansowanych, rozbudowanych systemów).

W laboratorium rolę serwera pełni komputer o nazwie ECO8 o numerze IP 192.168.68.17. Uruchomiony jest na nim serwer Microsoft SQL Server 2008 R2 Express. Jest to nieco ograniczona i uproszczona wersja profesjonalnego, bardzo zaawansowanego narzędzia, jakim jest MS SQL Server.

Komunikacja pomiędzy aplikacją, a serwerem odbywa się na zasadzie zapytanie-odpowiedź. Zapytania przesyłane są w postaci ciągów tekstowych, które pisane są w języku SQL. Po wykonaniu polecenia SQL, serwer zwraca rezultat jego wykonania – w szczególności, są to dane, o które „prosiła” aplikacja. Typowa sekwencja operacji, jakie trzeba wykonać w programie, by nawiązać połączenie z bazą danych jest następująca:

1. Konfiguracja połączenia z DBMS
2. Nawiązane połączenia z DBMS
3. Utworzenie zapytania SQL
4. Wysłanie zapytania SQL do DBMS
5. Odebranie odpowiedzi z serwera
6. Zamknięcie połączenia

Poniżej zawarte są najważniejsze informacje potrzebne do realizacji poszczególnych etapów połączenia.

### Konfiguracja połączenia z DBMS

Do obsługi połączenia z DBMS po stronie aplikacji w C++ służy klasa `System::Data::SqlClient::SqlConnection`. Należy utworzyć obiekt tej klasy. Jego konstruktor pozwala podać tzw. *connection string*, czyli tekstowy ciąg opisujący parametry połączenia z bazą danych. Ciąg ten musi zawierać przede wszystkim informacje o nazwie serwera SQL, nazwie bazy danych, z którą się łączymy (DBMS może zarządzać wieloma bazami równocześnie) oraz informacje o sposobie autoryzacji użytkownika. Zależnie od ustawień DBMS, może on sam zarządzać użytkownikami i ich prawami dostępu lub korzysta z mechanizmów kontroli dostępu systemu Windows. Drugi ze sposobów jest zalecany jako bezpieczniejszy. Ponieważ pierwszy ze sposobów jest jednak prostszy zarówno w konfiguracji jak i stosowaniu, w naszym przypadku wybrano zarządzanie użytkownikami z poziomu DBSM. Na serwerze zdefiniowany został użytkownik o nazwie „StudentDB” i hasło dostępu „sqlpass”. Będzie on wykorzystywany przez nasze aplikacje w celu autoryzacji wykonywanych operacji. Więcej informacji dotyczących możliwych elementów ciągu *connection string* można znaleźć na stronach:

<http://msdn.microsoft.com/>

[en-us/library/system.data.sqlclient.sqlconnection.connectionstring\(VS.90\).aspx](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.connectionstring(VS.90).aspx)

<http://www.connectionstrings.com/sql-server-2008>

**Przykładowy kod realizujący konfigurację połączenia z DBMS:**

---

```
System::Data::SqlClient::SqlConnection^ polaczenie;  
    //wskaźnik do obiektu polaczenie klasy SqlConnection  
polaczenie= gcnew System::Data::SqlClient::SqlConnection(→  
    →"Data Source=192.168.68.17; Database=Test; →  
    →User Id=StudentDB; Password=sqlpass;"); →  
    →//utworzenie obiektu
```

---

### Nawiązanie połączenia z DBMS

Po skonfigurowaniu połączenia z DBMS można je nawiązać. Służy do tego metoda `Open()` z klasy `SqlConnection`. Np.:

```
polaczenie->Open();
```

W przypadku, gdy połączenie nie może zostać nawiązane, metoda ta zgłasza wyjątek klasy `System::Data::SqlClient::SqlException`

Stan połączenia można sprawdzić korzystając z właściwości `State`.

### Utworzenie zapytania SQL

Do obsługi zapytania SQL po stronie aplikacji w C++ służy klasa `System::Data::SqlClient::SqlCommand`.

Najwygodniej jest najpierw utworzyć obiekt, a następnie zbudować odpowiedni tekst zapytania SQL i przypisać go do właściwości `CommandText` utworzonego obiektu. W przypadku bardziej złożonych zapytań (np. uwzględniających dane z kontroltek i zmiennych w programie) może to być dość rozbudowany ciąg kolejnych operacji na zmiennej typu `String`.

**Przykładowy kod realizujący utworzenie zapytania SQL:**

---

```
System::Data::SqlClient::SqlCommand^ komenda = →  
    → gcnew System::Data::SqlClient::SqlCommand();  
    //Utworzenie obiektu obsługującego zapytania SQL  
komenda->Connection=polaczenie;  
komenda->CommandText="SELECT * FROM Test.dbo.TabA"; //Treść zapytania
```

---

Składnia podstawowych zapytań SQL omówiona była na wykładzie. Szczegółowe informacje o poszczególnych poleceniach SQL i ich elementach można znaleźć na stronie:

[http://msdn.microsoft.com/en-us/library/aa299742\(SQL.80\).aspx](http://msdn.microsoft.com/en-us/library/aa299742(SQL.80).aspx)

### Wysłanie zapytania SQL

Do wysłania zapytania do DBSM w celu jego wykonania służy metoda `ExecuteReader()` z klasy `SqlCommand`. Metoda ta zwraca uchwyt do obiektu klasy

`System::Data::SqlClient::SqlDataReader`. Metody zawarte w tej klasie pozwolą w następnym kroku odczytać dane zwrócone przez serwer. Należy więc najpierw utworzyć (statyczny) uchwyt do obiektu klasy `SqlDataReader`, a następnie wywołać metodę `ExecuteReader()` dla określonego wcześniej zapytania.

Przykładowy kod realizujący utworzenie obiektu czytającego dane i wykonanie zapytania SQL:

---

```
System::Data::SqlClient::SqlDataReader^ czytacz = →  
                                                    →komenda->ExecuteReader();  
//Utworzenie obiektu odczytującego wyniki i wykonanie zapytania
```

---

### Odebranie odpowiedzi z serwera

Jeżeli oczekujemy, że w wyniku działania zapytania serwer zwróci dane, wówczas musimy je od niego odebrać. Służy do tego metoda `Read()` z klasy `SqlDataReader`. W przypadku ogólnym nie wiemy, jak dużo wierszy danych będzie zwróconych, w związku z czym, należy je odczytywać w pętli `while`. Aby odczytać dane z poszczególnych kolumn wyniku zapytania SQL, należy użyć odpowiedniej metody z klasy `SqlDataReader`. Metody z tej klasy pozwalają odczytać także dodatkowe informacje o strukturze danej relacji / tabeli oraz o samym zapytaniu (np. liczba zwróconych wierszy wynikowych). Pewnym utrudnieniem jest fakt, że trzeba dokładnie znać typy danych w poszczególnych kolumnach wyniku. Jest to konieczne by wybrać odpowiednią metodę pobierającą dane.

Listę wszystkich metod dostępnych w klasie `SqlDataReader` można znaleźć na stronie:

[http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldatareader\\_members\(VS.90\).aspx](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldatareader_members(VS.90).aspx)

Przykładowy kod realizujący odczytanie danych wynikowych z zapytania SQL:

---

```
while(czytacz->Read()) //Odczytanie rezultatów zapytania  
{  
    Zmienna1=czytacz->GetInt64(0);  
    //Pobranie wartosci typu bigint z zerowej kolumny wyniku  
    Zmienna2=czytacz->GetString(1);  
    //Pobranie wartosci typu nchar z pierwszej kolumny wyniku  
    //[...]<-- Tutaj, dalsze przetwarzanie zmiennych 1 i 2  
}
```

---

### Zamknięcie połączenia

Po zakończeniu operacji na bazie danych należy zamknąć połączenie z DBMS. Do tego celu służy metoda `Close()` z klasy `SqlConnection`. Np.:

```
polaczenie->Close();
```

### Uwagi dodatkowe

Jeśli program wykonuje wiele operacji na bazie danych, zamiast otwierać i zamykać połączenie przy każdym zapytaniu, wygodniej jest otworzyć połączenie raz, na początku programu (na przykład w konstruktorze formularza), i zamknąć je przy wychodzeniu z niego.

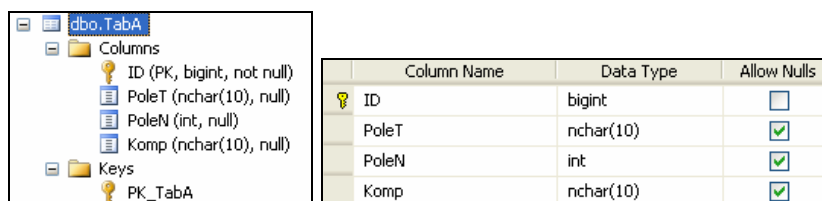
**Uwaga:** We wszystkich ćwiczeniach dotyczących baz danych wszyscy studenci pracują na bazach umieszczonych na serwerze ECO8. Oznacza to, że pracują na wspólnych bazach danych. Zawartość baz nie jest czyszczona pomiędzy zajęciami. **Pamiętaj, że dane, które umieścisz w bazie będą widziane nie tylko przez Ciebie i nie tylko na Twoim komputerze, ale przez wszystkich pozostałych studentów, także z innych grup.**

## 4.1 Ćwiczenie 32 – Obsługa prostej bazy danych

### 4.1.1 Ćwiczenie 32a – Struktura bazy

Na serwerze ECO8 działa baza o nazwie *Test*. Baza ta składa się z jednej tabeli o nazwie *TabA*. Tabela ta ma 4 kolumny (Rys. 19):

- ID – typ danych: *bigint* – liczby całkowite o zakresie wartości  $-/+2^{63}$ 
  - kolumna ma ustawiony parametr *Identity* i automatyczny przyrost wartości o 1 przy każdym dodaniu krotki, wpisanie wartości *NULL* jest niedozwolone
  - kolumna ta jest ustawiona jako klucz podstawowy (*PRIMARY\_KEY*)
  - kolumna wykorzystana będzie do przechowywania identyfikatorów poszczególnych krotek
- PoleT – typ danych: *nchar(10)* – ciągi tekstowe Unicode o długości 10 znaków; jeżeli ciąg jest krótszy, zostaje na końcu uzupełniony spacjami
  - kolumna będzie wykorzystana do przechowywania wartości tekstowych
- PoleN – typ danych: *int* – liczby całkowite o zakresie wartości  $-/+2^{31}$ 
  - kolumna będzie wykorzystana do przechowywania wartości tekstowych
- Komp – typ danych: *nchar(10)* – ciągi tekstowe Unicode o długości 10 znaków
  - kolumna będzie wykorzystana do przechowywania nazwy komputera, z którego została dodana dana krotka

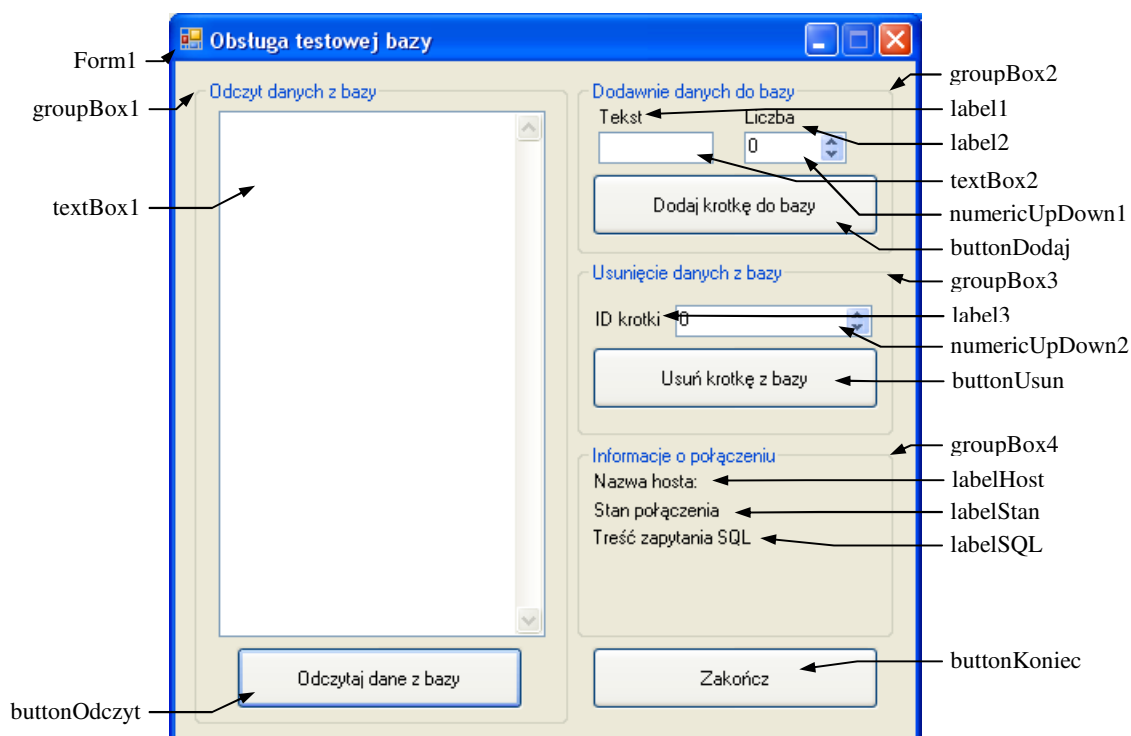


Column Name	Data Type	Allow Nulls
ID	bigint	<input type="checkbox"/>
PoleT	nchar(10)	<input checked="" type="checkbox"/>
PoleN	int	<input checked="" type="checkbox"/>
Komp	nchar(10)	<input checked="" type="checkbox"/>

Rys. 19 Widok struktury tabeli *TabA* – fragmenty widoków ekranu z aplikacji Microsoft SQL Server Management Studio

### 4.1.2 Ćwiczenie 32b – Program obsługujący prostą bazę danych

Utwórz nowy projekt dla GUI i napisz program umożliwiający wykonanie podstawowych operacji na bazie danych *Test* tj.: wyświetlenie zawartości tabeli, dodanie krotki i usunięcie wybranej krotki. Interfejs programu powinien wyglądać tak jak na Rys. 20. Na rysunku podane są także nazwy, jakie należy nadać poszczególnym kontrolkom poprzez ustawienie właściwości *Name*. Uwaga: ustaw nazwy dokładnie takie, jak pokazano. Jeśli kontrolki będą nazywać się inaczej, kod źródłowy podany w dalszej części ćwiczenia nie będzie działać.



Rys. 20

Ustawienia właściwości poszczególnych kontroltek oraz znaczenie niektórych z nich podane jest poniżej:

Kontrolka	Właściwość	Wartość do ustawienia	Uwagi dodatkowe
Form1	Text	Obsługa testowej bazy	Tytuł okna programu
	Size	470; 455	Rozmiar okna
	MaximizeBox	False	Wyłączenie wyświetlania przycisku maksymalizacji okna
	SizeGripStyle	Hide	Wyłączenie wyświetlania trójkątnego „uchwyty” zmiany rozmiaru w prawym, dolnym narożniku okna
groupBox1	Text	Odczyt danych z bazy	Tytuł panelu
	ScrollBars	Vertical	Włączenie wyświetlania paska przewijania w pionie
	Size	233; 400	Rozmiar panelu
textBox1	Multiline	True	Włączenie możliwości wyświetlania tekstu w wielu liniach
	Size	204; 328	Rozmiar kontrolki
buttonOdczyt	Text	Dodaj krotkę do bazy	
	Size	179; 39	
groupBox2	Text	Dodawanie danych do bazy	
	Size	197; 108	
label1	Text	Tekst	
label2	Text	Liczba	
textBox2	MaxLength	10	
	Multiline	False	
numericUpDown1	Maximum	65000	
	Minimum	0	

buttonDodaj	Size	179; 39	
groupBox3	Text	Usunięcie danych z bazy	
	Size	197; 108	
label3	Text	ID krotki	
numericUpDown2	Maximum	10000000	(10 mln.)
	Minimum	0	
buttonUsun	Size	179; 39	
groupBox4	Text	Informacje o połączeniu	
	Size	197; 120	
labelHost	Text	Nazwa hosta:	
labelStan	Text	Stan połączenia	
labelSQL	Text	Treść zapytania SQL	
	AutoSize	False	Wyłączenie automatycznego dopasowywania rozmiaru kontrolki do długości tekstu
	Size	182; 59	
buttonKoniec	Text	Zakończ	
	Size	179; 39	

Do kontroltek typu button przypisz obsługę zdarzeń *Click*:

- *buttonOdczyt\_Click*
- *buttonZapis\_Click*
- *buttonUsun\_Click*
- *buttonKoniec\_Click*

Kod źródłowy poszczególnych metod znajduje się poniżej. Opis kluczowych elementów kodu podany był we wstępie do ćwiczenia. Pod kodem każdej z metod zawarte są dodatkowe uwagi dotyczące jej działania.

```
private: System::Void buttonOdczyt_Click(System::Object^ sender, →
                                         →System::EventArgs^ e) {
    System::Data::SqlClient::SqlConnection^ polaczenie;
    polaczenie= gcnew System::Data::SqlClient::SqlConnection("Data →
        →Source=192.168.68.17;Database=Test;User Id=StudentDB; →
        →Password=sqlpass;"); →
        → //Utworzenie obiektu obsługi połączenia z serwerem

    labelHost->Text="Nazwa hosta: "+Environment::MachineName;

    try {
        polaczenie->Open(); //Otwarcie połączenia
    } catch (System::Data::SqlClient::SqlException ^wyjatek) {
        //Jeśli wystąpił wyjątek - brak połączenia
        labelStan->Text="Brak połączenia z serwerem";
    }

    if (polaczenie->State==System::Data::ConnectionState::Open){
        //Jeśli jest połączenie - pokaż zawartość bazy
        labelStan->Text="Połączono z serwerem "+polaczenie->DataSource->ToString();
        System::Data::SqlClient::SqlCommand^ komenda = →
            → gcnew System::Data::SqlClient::SqlCommand();
            //Utworzenie obiektu obsługującego zapytania SQL
        komenda->Connection=polaczenie;
        komenda->CommandText="SELECT * FROM Test.dbo.TabA"; //Treść zapytania
        labelSQL->Text=komenda->CommandText;
        textBox1->Clear();

        System::Data::SqlClient::SqlDataReader^ czytacz = komenda->ExecuteReader();
        //Utworzenie obiektu odczytującego wyniki i wykonanie zapytania
```



```

while(czytacz->Read()) { //Odczytanie rezultatów zapytania
    //Budowanie jednej linii tekstu z danymi z bazy do wyświetlenia w TextBoxie,
    //konieczne są odpowiednie konwersje z typów danych poszczególnych kolumn ...
    //...tabeli na typy z C++
    textBox1->AppendText(czytacz->GetInt64(0).ToString());
    textBox1->AppendText(" | ");
    textBox1->AppendText(czytacz->GetSqlString(1).ToString()->TrimEnd());
    //Pobranie tekstu z bazy i konwersja na String;
    //PoleT ma stałą długość - konieczne...
    //...obcięcie znaków za tekstem (TrimEnd())
    textBox1->AppendText(" | ");
    textBox1->AppendText(czytacz->GetInt32(2).ToString());
    textBox1->AppendText(" | ");
    textBox1->AppendText(czytacz->GetSqlString(3).ToString()->TrimEnd());
    textBox1->AppendText("\r\n");
}
czytacz->Close();
polaczenie->Close();
}
}

```

Uwagi dodatkowe i wyjaśnienia do kodu źródłowego metody *buttonOdczyt\_Click*

- *Environment::MachineName* - *MachineName* jest właściwość (stała) należąca do klasy *Environment*. Zawiera ona nazwę komputera (hosta lokalnego) NetBIOS, na którym uruchomiony jest program. W laboratorium będą to nazwy od ECO1 do ECO12.
- Zapytanie SQL w postaci „SELECT \* FROM Test.dbo.TabA” – W MSSQL Server zalecane jest odwoływanie się do poszczególnych tabel bazy poprzez ich pełną ścieżkę, podając ją w postaci *baza.schemat.tabela* lub *serwer.baza.schemat.tabela* jeśli w programie nawiązywana jest łączność z kilkoma serwerami SQL. W naszym przypadku chcemy pobrać zawartość z tabeli *TabA*, opisanej schematem *dbo* i znajdującej się w bazie *Test*. Schemat jest „kontenerem”, który może zawierać różne obiekty bazodanowe (w szczególności tabele). Jest on tworzony w celu zarządzania dostępem (np. nadawaniem praw użytkownikom) do tych obiektów. W bazie *Test* schematem domyślnym dla użytkownika StudentDB jest schemat *dbo*.
- Wyświetlanie danych – obsługa kontrolki *textBox*. Dla uproszczenia budowy programu przyjęto, że dane odczytane z bazy będą zamieniane na wartości tekstowe i wyświetlane w kolejnych wierszach pola tekstowego. Przyjęto, że zawartość poszczególnych kolumn będzie oddzielana znakiem „|”. Kontrolka *textBox* posiada metodę *AppendText()*, która pozwala dodać („dopisać”) ciąg tekstowy do tekstu już wyświetlanego w kontrolce. Procedura wyświetlania danych pobieranych z bazy polega na tym, że w pętli pobierane są dane kolejnych krotek / wierszy tabeli. Odczytane dane znajdują się w obiekcie *czytacz*. Wewnątrz pętli, należy pobrać wartości poszczególnych pól krotki. Musi to być wykonane za użyciem jednej z metod obiektu *czytacz*, dopasowanej do typu danych pobieranego pola / kolumny danych. Pobraną wartość należy następnie przekonwertować do postaci tekstowej i dodać do już wyświetlanego w kontrolce *textBox* tekstu. Pomiędzy wartościami trzeba dostawić znak „|”, a po pobraniu wszystkich wartości z krotki – dodać znak końca linii „\r\n”.

```

private: System::Void buttonZapis_Click(System::Object^ sender, →
    →System::EventArgs^ e) {
    System::Data::SqlClient::SqlConnection^ polaczenie;

    polaczenie= gcnew System::Data::SqlClient::SqlConnection("Data →
        →Source=192.168.68.17;Database=Test;User Id=StudentDB; →
        →Password=sqlpass;");
    //Utworzenie obiektu obsługi połączenia z serwerem

    labelHost->Text="Nazwa hosta: "+Environment::MachineName;

```

```
try {
    polaczenie->Open();           //Otwarcie połączenia
} catch (System::Data::SqlClient::SqlException ^wyjatek) {
    //Jeśli wystąpił wyjątek - brak połączenia
    labelStan->Text="Brak połączenia z serwerem";
}

if (polaczenie->State==System::Data::ConnectionState::Open){
    //Jeśli jest połączenie - dodaj krotkę do bazy
    labelStan->Text="Połączono z serwerem "+polaczenie->DataSource->ToString();

    System::Data::SqlClient::SqlCommand^ komenda= →
        → gcnew System::Data::SqlClient::SqlCommand();
        //Utworzenie obiektu obsługującego zapytania SQL
    komenda->Connection=polaczenie;
    komenda->CommandText="INSERT INTO Test.dbo.TabA ([PoleT],[PoleN],[Komp]) →
        → VALUES('";           //Budowa tekstu zapytania SQL
    komenda->CommandText+=textBox2->Text;
    komenda->CommandText+="',";
    komenda->CommandText+=numericUpDown1->Value;
    komenda->CommandText+="',";
    komenda->CommandText+=Environment::MachineName;
        //Pobranie nazwy komputera, z którego dodajemy dane...
        //... do bazy (komputera, na którym uruchomimy program)
    komenda->CommandText+="')";
    labelSQL->Text=komenda->CommandText;
    textBox1->Clear();

    System::Data::SqlClient::SqlDataReader^ czytacz = komenda->ExecuteReader();
        //Ponieważ zapytanie dotyczyło wstawienia danych do bazy,...
        //... nie ma nic do odczytania, czytacz nie jest potrzebny,
        //(choć powinniśmy sprawdzić, czy faktycznie polecenie zostało wykonane)
        //Utworzenie uchwytu do obiektu odczytującego wyniki i wykonanie zapytania
    czytacz->Close();
    polaczenie->Close();
}
}
```

---

#### Uwagi dodatkowe i wyjaśnienia do kodu źródłowego metody *buttonOdczyt\_Zapis*

- Zapytanie SQL w postaci „INSERT INTO Test.dbo.TabA ([PoleT],[PoleN],[Komp]) VALUES(wartości);” – W MSSQL Server możliwe jest umieszczanie nazw pól tabeli w nawiasach kwadratowych co ułatwia ich zauważenie i odczytanie przy czytaniu zapytania.
- Budowa zapytania SQL. Zapytanie, jakie ma zostać wykonane musi być ciągiem tekstowym, niezależnie od typu danych, jakie mają być wstawione do bazy danych. Początkowy fragment zapytania wstawiającego dane do bazy (INSERT) jest stały i określa nazwę tabeli oraz nazwy kolumn, do których mają być wstawione wartości. Same wartości są zmienne i zależą od ustawień poczynionych przez użytkownika programu. Dlatego dalszą część zapytania należy „budować” w sposób dynamiczny. Można w tym celu wykorzystać operator „+=”, który w przypadku zmiennych typu *String* pozwala dołączyć („dopisać”) ciąg znaków do ciągu już istniejącego wcześniej. W przypadku naszego zapytania należy pobrać dane z odpowiednich kontroltek i przekonwertować je, o ile jest taka konieczność, na postać tekstową. Ważnym jest, by pamiętać, że pomiędzy poszczególnymi wartościami należy w zapytaniu umieścić przecinki, oraz, dodatkowo, że ciągi znakowe muszą być zawarte w apostrofach pojedynczych.

---

```
private: System::Void buttonUsun_Click(System::Object^ sender, →
    →System::EventArgs^ e) {
    System::Data::SqlClient::SqlConnection^ polaczenie;

    polaczenie= gcnew System::Data::SqlClient::SqlConnection("Data →
        →Source=192.168.68.17;Database=Test;User Id=StudentDB;Password=sqlpass;");
```



```
//Utworzenie obiektu obsługi połączenia z serwerem
labelHost->Text="Nazwa hosta: "+Environment::MachineName;

try {
    polaczenie->Open();           //Otwarcie połączenia
} catch (System::Data::SqlClient::SqlException ^wyjatek) {
    //Jeśli wystąpił wyjątek - brak połączenia
    labelStan->Text="Brak połączenia z serwerem";
}

if (polaczenie->State==System::Data::ConnectionState::Open){
    //Jeśli jest połączenie - usuń krotkę z bazy
    labelStan->Text="Połączono z serwerem "+polaczenie->DataSource->ToString();
    System::Data::SqlClient::SqlCommand^ komenda = →
        → gcnew System::Data::SqlClient::SqlCommand();
    //Utworzenie obiektu obsługującego zapytania SQL
    komenda->Connection=polaczenie;
    komenda->CommandText="DELETE FROM Test.dbo.TabA WHERE ID="+→
        →numericUpDown2->Value.ToString(); //Treść zapytania
    labelSQL->Text=komenda->CommandText;
    textBox1->Clear();

    System::Data::SqlClient::SqlDataReader^ czytacz = komenda->ExecuteReader();
    //Utworzenie uchwytu do obiektu odczytującego wyniki i wykonanie zapytania
    czytacz->Close();
    polaczenie->Close();
}
}

private: System::Void buttonKoniec_Click(System::Object^ sender, →
    →System::EventArgs^ e) {
    Application::Exit();
}
```

---

Uruchom napisany program i sprawdź jego działanie. Dodaj kilka krotek do bazy, sprawdź zawartość bazy, usuń dodane przez siebie krotki z bazy.

## 4.2 Ćwiczenie 33 –Obsługa bazy danych 2, zadanie kompleksowe – „Rezerwacje sal wykładowych”

Utwórz nowy projekt dla GUI i napisz jeden z poniższych programów:

- program umożliwiający zarezerwowanie sali wykładowej dla konkretnej grupy, na wybrany termin,
- program umożliwiający sprawdzenie (przeglądanie) rezerwacji sal wykładowych.

Oba programy mają wykorzystywać bazę danych *RezSal* i wraz nią będą stanowić uproszczony system rezerwacji sal wykładowych.

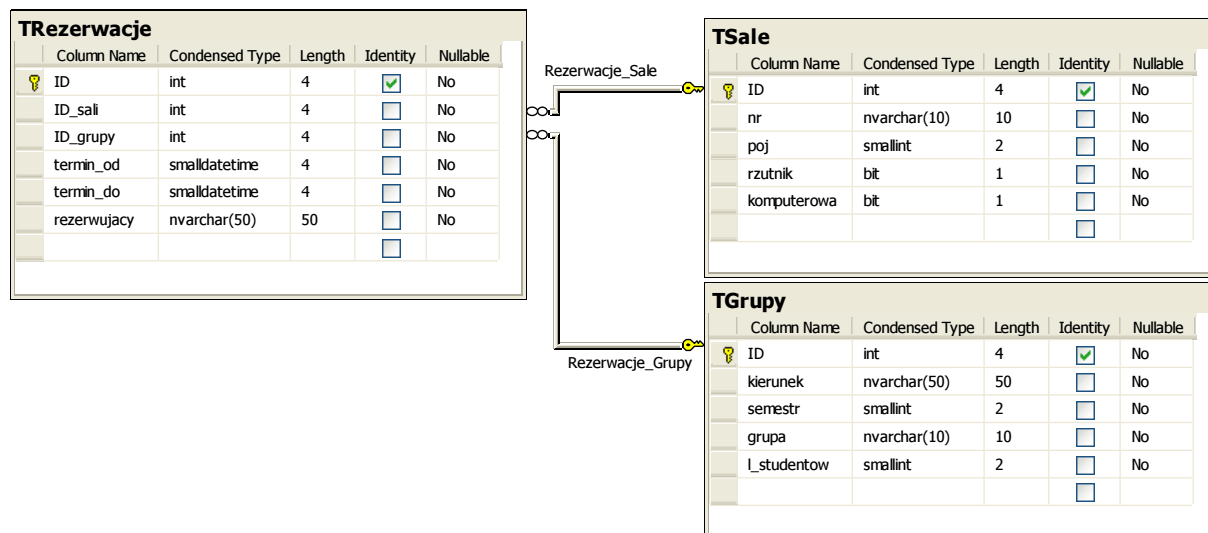
### Struktura bazy danych.

Baza *RezSal* uruchomiona na serwerze ECO8 składa się 3 tabel:

- *TGrupy* – przechowującej informacje o grupach studenckich
- *TSale* – przechowującej informacje o salach
- *TRezerwacje* – przechowującej informacje o rezerwacjach

Zawartość tabeli *TGrupy* i *TSale* nie będzie ulegać zmianie.

Nazwy, typy pól i powiązania pomiędzy tabelami pokazane są na Rys. 21.



Rys. 21

### Kolumny w poszczególnych tabelach

- TGrupy
  - o ID – identyfikator grupy, liczba całkowita 4-bajtowa, klucz główny tabeli
  - o kierunek – nazwa kierunku, tekst
  - o semestr – numer semestru, liczba całkowita 2-bajtowa
  - o grupa – nazwa grupy, tekst, przyjęto oznaczenia:
    - „cały rok” – wszyscy studenci danego roku
    - 1C, 2C, ... – grupy ćwiczeniowe, grupy dziekańskie
    - 1L, 2L, ... – grupy laboratoryjne
  - o l\_studentów – liczba studentów w danej grupie, liczba całkowita 2-bajtowa
- TSale – przechowującej informacje o salach
  - o ID – identyfikator sali, liczba całkowita 4-bajtowa, klucz główny tabeli
  - o nr – numer / nazwa sali, tekst (dopuszczalne są nie tylko nazwy ściśle numeryczne, ale także np.: „200D”, „Audytorium” itp.)
  - o poj – pojemność sali, liczba całkowita 2-bajtowa
  - o rzutnik – określa, czy sala wyposażona jest w rzutnik multimedialny, wartość bitowa, wartość 1 oznacza, że w sali jest rzutnik
  - o komputerowa – określa, czy sala jest salą komputerową
- TRezerwacje – przechowującej informacje o rezerwacjach
  - o ID – identyfikator rezerwacji, liczba całkowita 4-bajtowa, klucz główny tabeli
  - o ID\_sali – identyfikator sali, liczba całkowita 4-bajtowa, klucz obcy wiążący z tabelą *TSale*
  - o ID\_grupy – identyfikator grupy, liczba całkowita 4-bajtowa, klucz obcy wiążący z tabelą *TGrupy*
  - o termin\_od – termin początku rezerwacji, pole typu *smalldate* zapisujące datę i godzinę z dokładnością do minuty, np.: „2010-09-01 12:20:00”
  - o termin\_do – termin końca rezerwacji, pole typu *smalldate*
  - o rezerwujacy – nazwisko osoby rezerwującej, tekst

### Programy

Poniżej znajdują się uwagi dotyczące wymogów stawianych poszczególnym programom i sposobów realizacji zadania (w tym fragmenty kodu źródłowego). Niezależnie od wyboru zadania do realizacji, przeczytaj uwagi do obu programów, ponieważ omawiane

zagadnienia w równym stopniu ich dotyczą (np. sposób obsługi wyboru semestru czy grupy będzie taki sam w obu programach).

#### 4.2.1 Ćwiczenie 33a – Rezerwowanie sal wykładowych

Program powinien:

- Pobierać dane o grupach studenckich
- Umożliwiać filtrowanie grup poprzez podanie semestru studiów
- Pobierać dane o dostępnych salach
- Umożliwić filtrowanie listy sal pod względem wyposażenia w rzutnik oraz wyposażenia w komputery
- Umożliwić podanie daty rezerwacji o raz godziny początkowej i końcowej dla rezerwacji
- Umożliwić sprawdzenie, czy wybrana sala jest dostępna w podanym terminie
- Umożliwić wpisanie nazwiska osoby rezerwującej
- Zapisać rezerwację sali do bazy danych – po sprawdzeniu poprawności danych oraz dostępności sali w podanym terminie.

Wygląd interfejsu programu pokazany jest na Rys. 22

Rys. 22

#### Uwagi:

- Większość pól musi być wypełniona odpowiednimi wartościami pobranymi z bazy danych. Inicjalizację najlepiej wykonać poprzez oprogramowanie obsługi zdarzenia *Load* dla formularza *Form1*. Zdarzenie to zgłaszane jest przed pierwszym wyświetleniem formularza.
- Pole semestr powinno mieć nałożone ograniczenie w ten sposób, by można było wybrać wartości z zakresu od 1 do maksymalnego numeru semestru (należy go pobrać z bazy, np.: poleceniem „`SELECT max(semestr) FROM RezSal.dbo.TGrupy`”)
- Pole wyboru daty należy zrealizować z użyciem kontrolki *dateTimePicker*.

- Listy rozwijalne należy zrealizować z użyciem kontrolki *comboBox*. Teksty, które wyświetlane są jako kolejne pozycje na liście znajdują się we właściwości *Items*. Gdy użytkownik wybierze jedną z pozycji, we właściwości *SelectedIndex* można sprawdzić, która pozycja została wybrana. Wartość -1 oznacza brak wyboru.
  - W przypadku godzin początkowych i końcowych, należy kolejne pozycje (godziny) wpisać na etapie budowy programu. Dopuszczalne godziny początkowe to: 7:15, 8:15, ..., 18:15. Dopuszczalne godziny końcowe to: 8:00, 9:00, ..., 19:00
  - W polach wyboru grupy oraz sali pojawi się problem. Dane do list rozwijalnych powinny być pobrane z bazy i można je uznać za praktycznie stałe (zawartość tabeli *TSale* zmienia się bardzo rzadko, zaś *TGrupy* – najwyżej kilka razy w semestrze). Z drugiej strony zawartość tych pól nie jest stała, i zmienia się w zależności od wyboru semestru lub ustawień filtru rodzaju sali. Problem ten można rozwiązać na dwa sposoby:
    - Pobierać dane z bazy przy każdej zmianie filtru
      - Zaletą tego rozwiązania jest łatwość realizacji. Wystarczy jedynie sformułować odpowiednie zapytania do bazy danych
      - Wadą jest duża częstotliwość wysyłania zapytań gdyż muszą być wysyłane przy każdej zmianie w ustawieniach filtrów. Wykonanie zapytania (zwłaszcza przy połączeniu ze zdalnym serwerem) może być czasochłonne więc częste zapytania spowodują, że program będzie wolno reagować na polecenia użytkownika.
    - Pobrać dane z bazy raz, przy wywołaniu programu, a później wybierać dane do wyświetlenia w zależności od ustawień filtrów.
      - Zaletą tego rozwiązania jest odciążenie bazy danych i szybka reakcja programu na działania użytkownika.
      - Wadą jest bardziej złożona realizacja. Wymaga ona pobrania wszystkich potrzebnych danych z bazy do wewnętrznej, programowej tablicy. Filtrowanie danych do wyświetlenia oraz określenie, który z elementów został wybrany przez użytkownika trzeba będzie zrealizować programowo – nie będzie można skorzystać z automatycznej filtracji zapewnianej przez bazę danych.
    - Sposób rozwiązania problemu z wykorzystaniem drugiej metody będzie omówiony poniżej.

### **Pobranie danych dotyczących sal i ich filtracja przy wyświetlaniu:**

Problem pobierania danych i ich filtracji można rozwiązać w ten sposób, że utworzony zostanie obiekt klasy *KSale*. Obiekt ten składać się będzie z dwóch pól: *lSal* zawierającego liczbę sal oraz *lista* będącego dynamiczną tablicą nadzorowaną zawierającą informacje o poszczególnych salach. Informacje te będą przechowywane w strukturach opisanych przez typ strukturalny *SSala*. Składniki typu *SSala* odpowiadać będą kolejnym kolumnom z bazy danych. Dzięki temu będzie można w łatwy sposób przenieść zawartość tabli *TSale* do obiektu klasy *KSale*. Dodatkowo, w *SSala* dodane zostanie pole *wyswietlana*, które pozwoli zapamiętać, które z sal są wyświetlane w danym momencie w kontrolce *comboBox*.

Poniżej znajduje się kod źródłowy opisujący deklaracje omówionej klasy i struktury oraz deklarujący uchwyt do obiektu klasy *KSale*. Kod ten należy umieścić wewnątrz deklaracji klasy *Form1* dzięki czemu rozszerzymy klasę formularza o potrzebne nam elementy.

Ponieważ będziemy korzystać z nich tylko wewnątrz okna formularza *Form1* takie rozwiązanie jest uzasadnione. Inną możliwością jest umieszczenie własnych deklaracji „równolegle” do deklaracji klasy *Form1*.

Deklaracja klasy *Form1* rozpoczyna się od linii:

```
public ref class Form2 : public System::Windows::Forms::Form {  
    ref struct SSala{  
        int id;  
        String ^numer;  
        int pojemnosc;  
        bool rzutnik;  
        bool komputerowa;  
        bool wyswietlana;  
    };  
  
    ref class KSale{  
    public:  
        int lSal;  
        array<SSala^> ^lista;  
    };  
  
    private: KSale ^sale;
```

Jak wspomniano wcześniej, inicjalizację tablicy informacji o salach najlepiej wykonać przed załadowaniem formularza, tzn. w metodzie *Form1\_Load* przypisanej do zdarzenia *Load* dla formularza *Form1*. Sekwencja operacji jakie należy przeprowadzić jest następująca:

1. Połączenie z bazą danych
2. Wysłanie do bazy danych zapytania o liczbę sal
3. Utworzenie obiektu *sale* klasy *KSale*
4. Pobranie liczby sal
5. Utworzenie tablicy dynamicznej do przechowywania struktur opisujących sale
6. Wysłanie do bazy danych zapytania o informacje o kolejnych salach
7. Pobranie informacji o kolejnych salach
8. Wpisanie do kontrolki *comboBox* listy nazw wszystkich sal i ustawienie statusu *wyswietlana* na wartość *true*.

Warto zauważyć, że pobierając informacje o salach z bazy danych, można by dowiedzieć się, ile jest sal. Można by więc zrezygnować z kroku 2 i 4. Oznaczałoby to jednak, że później, w kroku 7 trzeba stopniowo zwiększać rozmiar tablicy dynamicznej. Jest to rozwiązanie nieefektywne gdyż stopniowe rozszerzanie tablicy wiąże się z wyszukiwaniem przez system nowych bloków pamięci, kopiowaniem do nich danych i zwalnianiem poprzednio użytego bloku – jest to powolne. Lepiej jest od razu utworzyć tablicę o odpowiednim rozmiarze.

Należy także zauważyć, że obiekt wskazywany przez uchwyt *sale* jest obiektem dynamicznym, musi być więc utworzony z użyciem wyrażenia *gcnew*. Następnie, składnikiem obiektu jest uchwyt do tablicy dynamicznej o nazwie *lista*. Ponownie należy użyć wyrażenia *gcnew*. Elementami listy są uchwyty do struktur typu *SSala*. Oznacza to, że poszczególne elementy listy są tworzone także z użyciem wyrażenia *gcnew*. Wewnątrz struktury *SSala* znajduje się uchwyt do ciągu tekstowego typu *String* o nazwie *numer*. Podczas pobierania danych z bazy, uchwyt ten otrzyma wartość wskazującą na tekst od funkcji *GetString()*. Reasumując – zmienna *sale* jest uchwytem do obiektu. Składnikiem obiektu jest uchwyt do tablicy. Elementami tablicy są uchwyty do struktur, zaś elementem każdej struktury jest uchwyt do ciągu tekstowego. Wszystkie uchwyty muszą zostać sukcesywnie określone podczas deklarowania zmiennych i obiektów dynamicznych.

```

private: System::Void Form2_Load(System::Object^ sender, System::EventArgs^ e)
{

    int i;
    System::Data::SqlClient::SqlConnection^ polaczenie;

    //krok 1 - Utworzenie obiektu obsługi połączenia z serwerem
    polaczenie= gcnew System::Data::SqlClient::SqlConnection("Data →
    →Source=192.168.68.17;Database=Test;User Id=StudentDB;Password=sqlpass;");
    try {
        polaczenie->Open(); //Otwarcie połączenia
    } catch (System::Data::SqlClient::SqlException ^wyjatek) {
        //Jeśli wystąpił wyjątek - brak połączenia
        //[...] tu obsługa braku połączenia
    }

    if (polaczenie->State==System::Data::ConnectionState::Open){
        //Jeśli jest połączenie

        System::Data::SqlClient::SqlCommand^ komenda = →
            → gcnew System::Data::SqlClient::SqlCommand();
            //Utworzenie obiektu obsługującego zapytania SQL
        komenda->Connection=polaczenie;
        //krok 2 - Wysłanie zapytania o liczbę sal
        komenda->CommandText="SELECT COUNT(*) FROM RezSal.dbo.TSale";
        System::Data::SqlClient::SqlDataReader^ czytacz1 =komenda->ExecuteReader();
        czytacz1->Read();

        //krok 3 - Utworzenie obiektu klasy KSale
        sale = gcnew KSale;
        //krok 4 - pobranie liczby sal
        sale->lSal=czytacz1->GetInt32(0);
        //krok 5 - utworzenie pustej tablicy na opisy sal
        sale->lista= gcnew array<SSala^>(sale->lSal);
        czytacz1->Close();

        //krok 6 - wysłanie zapytania o dane sal, sortowanie wg. numeru sali

        //nowe zapytanie wysyłamy w tym samym połączeniu SQL
        //wykorzystujemy znów zmienną „czytacz1” - poprzednie
        //zapytania zostało już wcześniej zamknięte
        komenda->CommandText="SELECT * FROM RezSal.dbo.TSale ORDER BY nr";
        czytacz1 =komenda->ExecuteReader();

        //krok 7 - Odczytanie rezultatów zapytania - informacji o salach
        i=0;
        while(czytacz1->Read()) {
            sale->lista[i] = gcnew SSala; //utworzenie struktury dla pojedynczej sali
            sale->lista[i]->id=czytacz1->GetInt32(0);
            sale->lista[i]->numer =czytacz1->GetString(1).ToString()->TrimEnd();
            sale->lista[i]->pojemnosc=czytacz1->GetInt16(2);
            sale->lista[i]->rzutnik=czytacz1->GetBoolean(3);
            sale->lista[i]->komputerowa=czytacz1->GetBoolean(4);
            i++;
        }

        //krok 8 - Wyświetlenie listy wszystkich sal (na początku filtry są
        // wyłączzone)
        czytacz1->Close();
        for (i=0; i<(sale->lSal); i++){
            comboBoxNrSali->Items->Add(sale->lista[i]->numer);
            sale->lista[i]->wyswietlana=true;
        }

        //[...]Pozostała część kodu inicjalizującego...]
        //[...]Inicjalizacji maksymalnego numeru semestru, listy grup, itp. ...]
    }
}

```

```
        polaczenie->Close();  
    }  
}
```

W bardzo podobny sposób należy zrealizować obsługę pobierania informacji o grupach studenckich oraz inicjalizację obiektów i struktur je przechowujących. W kroku 8 należy mieć jednak na uwadze, że wyświetlane nie będą wszystkie grupy, ale tylko grupy z pierwszego semestru. Należy więc wstawiać wartości do kontrolki *comboBox* i ustawiać wartość *True* dla pola *wyswietlana* tylko dla odpowiednich grup.

Drugim problemem związanym z wyświetlaniem listy jest uwzględnienie filtrowania ustawionego przez użytkownika z użyciem kontrolki typu *checkBox*. Filtrowanie powinno działać w ten sposób, że gdy nie jest wybrany żaden filtr – wyświetlane są wszystkie sale. Gdy zaznaczone jest pole „Sala z rzutnikiem” wyświetlane są tylko sale wyposażone w rzutnik. Gdy zaznaczone jest pole „Sala komputerowa” wyświetlane są tylko laboratoria komputerowe. Gdy zaznaczone są oba pola filtrów wyświetlane są tylko sale komputerowe wyposażone w rzutnik.

Obsługę filtrowania najlepiej zawrzeć w osobnej metodzie dodanej do klasy formularza *Form1*, która będzie wywoływana w metodzie obsługującej zdarzenie *CheckedChanged* dla obu pól wyboru sposobu filtrowania. Zauważ, że w każdej z pętli realizującej wpisywanie wartości do kontrolki *comboBox* ustawiane są wartości zmiennej *wyswietlana*. Kod metody *filtrujSale* podany jest poniżej. Najlepiej jest go umieścić zaraz za kodem budującym interfejs użytkownika i generowanym automatycznie przez Visual C++, tzn. za linią `#pragma endregion`.

```
private: void filtrujSale(void){  
    //filtrowanie sali wg. ustawiń checkBoxów.  
    //zapamiętanie, które elementy (sale) z tablicy sa wyswietlane, a które nie  
    //dzięki temu później będzie można na podstawie SelectedIndex stwierdzić,  
    //która sala faktycznie została wybrana  
  
    int i;  
    comboBoxNrSali->Text="";           //wyczyszczenie poprzedniego wyboru  
    comboBoxNrSali->Items->Clear();  
  
    //sale komputerowe z rzutnikiem  
    if ((checkBoxSRzutnik->Checked==true)&&(checkBoxSKomputerowa->Checked==false)){  
        for (i=0; i<(sale->lSal); i++){  
            if (sale->lista[i]->rzutnik==true){  
                comboBoxNrSali->Items->Add(sale->lista[i]->numer);  
                sale->lista[i]->wyswietlana=true;  
            } else {  
                sale->lista[i]->wyswietlana=false;  
            }  
        }  
    }  
  
    //sale komputerowe  
    if ((checkBoxSRzutnik->Checked==false)&&(checkBoxSKomputerowa->Checked==true)){  
        for (i=0; i<(sale->lSal); i++){  
            if (sale->lista[i]->komputerowa==true){  
                comboBoxNrSali->Items->Add(sale->lista[i]->numer);  
                sale->lista[i]->wyswietlana=true;  
            } else {  
                sale->lista[i]->wyswietlana=false;  
            }  
        }  
    }  
}
```

```
//sale z rzutnikiem
if ((checkBoxSRzutnik->Checked==true) && (checkBoxSKomputerowa->Checked==true)) {
    for (i=0; i<(sale->lSal); i++){
        if ((sale->lista[i]->rzutnik==true) && (sale->lista[i]->komputerowa==true)) {
            comboBoxNrSali->Items->Add(sale->lista[i]->numer);
            sale->lista[i]->wyswietlana=true;
        } else {
            sale->lista[i]->wyswietlana=false;
        }
    }
}

//wszystkie sale
if ((checkBoxSRzutnik->Checked==false) && (checkBoxSKomputerowa->Checked==false)) {
    for (i=0; i<(sale->lSal); i++){
        comboBoxNrSali->Items->Add(sale->lista[i]->numer);
        sale->lista[i]->wyswietlana=true;
    }
}
```

Ostatnim problemem związanym z filtrowaniem sal jest określenie, która sala została wybrana. W przypadku, gdy wyświetlane są wszystkie sale jest to łatwe do określenia gdyż indeks wybranego elementu (właściwość *SelectedIndex* z kontrolki *comboBox*) pokrywa się z indeksem w tablicy z listą sal. Jednakże, w przypadku, gdy filtr jest włączony *SelectedIndex* zawiera numer z wyświetlanej, przefiltrowanej listy sal, który nie musi pokrywać się z pozycją w tablicy *lista*. Aby określić, któremu faktycznie elementowi z listy odpowiada wartość *SelectedIndex* należy przejrzeć tablicę i zliczać, które elementy są wyświetlane. Kod realizujący to zadanie podany jest poniżej. Po jego zakończeniu, zmienna *iWyswietlanejSali* podaje indeks wybranego elementu z pełnej listy sal. Kod należy umieścić w metodzie obsługującej budowanie zapytania SQL dodającego rezerwację do bazy danych. W podobny sposób można zrealizować obsługę wyboru grupy studenckiej.

```
int i, iWyswietlanejSali;
bool Blad;

Blad=false;
i=0;
iWyswietlanejSali=0;
if (comboBoxNrSali->SelectedIndex>=0) {
    while (i<=comboBoxNrSali->SelectedIndex) {
        if (sale->lista[iWyswietlanejSali]->wyswietlana==true) {
            i++;
        }
        iWyswietlanejSali++;
    }
    iWyswietlanejSali--;
} else {
    MessageBox::Show("Nie wybrano numeru sali", "Błąd", →
        → MessageBoxButtons::OK, MessageBoxIcon::Error);
    Blad=true;
}
```

---

### Określenie daty i godziny początku i końca rezerwacji:

Do określenia daty najlepiej użyć kontrolki *dateTimePicker*. Właściwość *Value* pozwala odczytać wybraną przez użytkownika datę. Przed dalszymi operacjami należy ją jeszcze przekonwertować do postaci tekstowej – najlepiej z użyciem metody *ToShortDateString()*, której wynikiem jest data w postaci np.: „2010-09-20”. Przy budowaniu zapytania SQL datę trzeba będzie uzupełnić o godzinę pobraną z pól *comboBox* z godziną początkową lub końcową.



**Dodanie rezerwacji do bazy:**

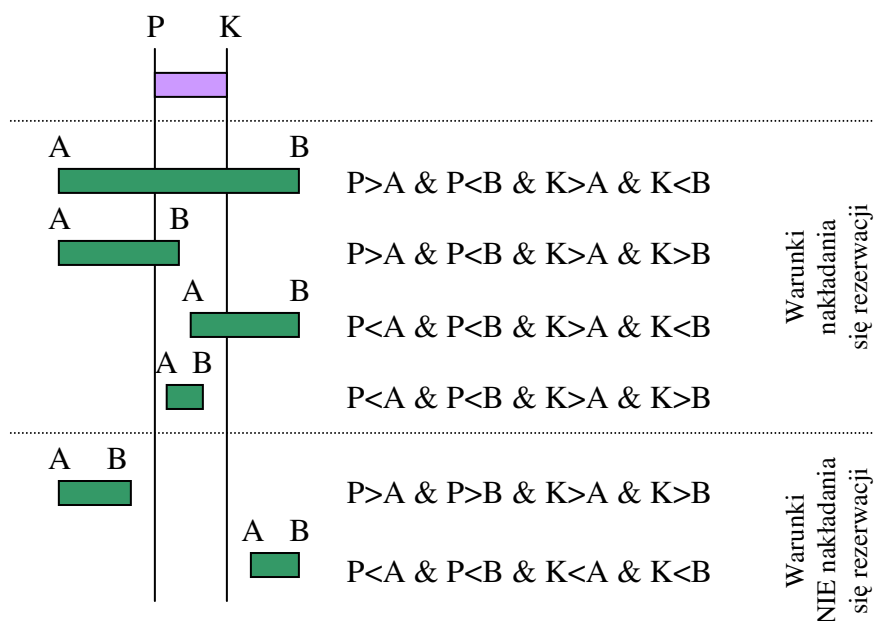
Dodanie nowej rezerwacji do bazy może być wykonane jedynie po sprawdzeniu czy użytkownik określił wartości we wszystkich polach (w tym także polu z nazwiskiem osoby rezerwującej sale).

W celu dodania rezerwacji do bazy należy zbudować odpowiednie zapytanie SQL. Przykładowe zapytanie podane jest poniżej:

```
INSERT INTO RezSal.dbo.TRezerwacje ([ID_sali], [ID_grupy], [termin_od],
[termin_do], [rezerwujacy]) VALUES(2, 6, '2010-09-20 8:15:00', '2010
09 20 10:00:00', 'Kowalski')
```

**Sprawdzenie istniejących rezerwacji sal:**

Program powinien umożliwiać sprawdzenie, czy wybrany termin jest wolny po wybraniu przez użytkownika przycisku „Sprawdź termin” oraz sprawdzać automatycznie, czy termin jest wolny przed dodaniem nowej rezerwacji do bazy. Sprawdzenie to można wykonać z użyciem odpowiednio zbudowanego zapytania SQL. Kluczowym elementem tego zapytania jest odpowiednie dobranie warunków, jakie muszą zostać sprawdzone by stwierdzić, czy termin jest wolny. Na Rys. 23 zobrazowano wszystkie możliwe przypadki położenia w czasie nowego terminu rozpoczynającego się w chwili A i kończącego w chwili B względem już istniejącej rezerwacji trwającej od chwili P do K. Warto zauważyć, że w bezpośrednie sprawdzenie czy termin AB koliduje z terminem PK wymaga skontrolowania aż 16 warunków. Można jednak to zagadnienie uprościć i zamiast warunków kolidowania terminów, sformułować warunek sprawdzający czy termin AB nie nakłada się na termin PK, a następnie warunek ten zanegować. Będzie to wymagało opisanie 8 warunków. Oprócz tego, należy uwzględnić w zapytaniu fakt, że sprawdzenie dat dotyczyć ma tylko jednej, wybranej sali. Ponadto, interesuje nas w praktyce sam fakt wystąpienia kolizji terminów, a nie, jakie są te terminy. W związku z tym, najlepiej jest sprawdzić, jaka jest liczba kolidujących terminów. Jeśli jest większa od 0 – oznacza to, że sala jest zarezerwowana.



Rys. 23 Warunki nakładania się rezerwacji:  
P, K – początek i koniec istniejącej rezerwacji;  
A, B – początek i koniec nowej rezerwacji

Poniżej podano przykład zapytania sprawdzającego czy sala o ID=2 jest zajęta 10 czerwca 2010 w godzinach od 9:15 do 10:00. Wynik większy od 0 oznacza, termin jest już zajęty.

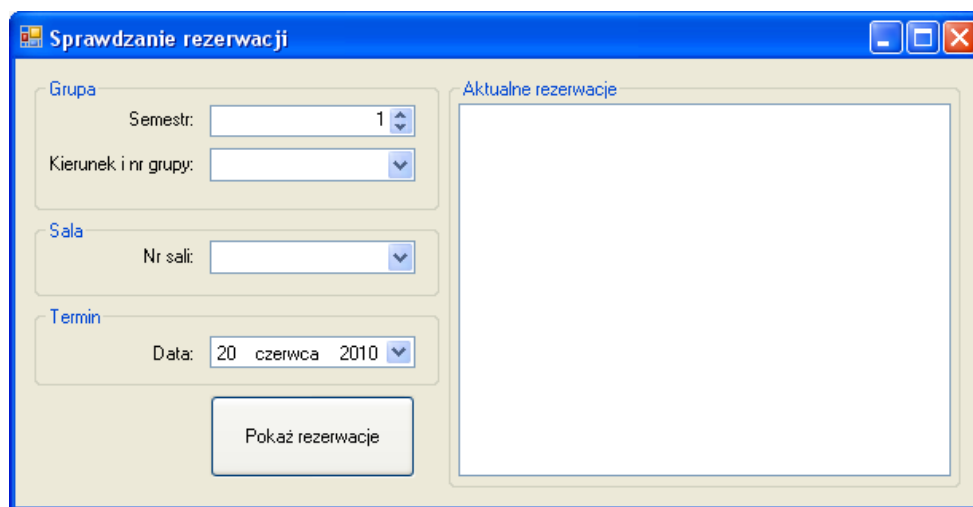
```
SELECT COUNT(*) From RezSal.dbo.TRezerwacje WHERE ID_sali=2 AND
(NOT ( (termin_od>='2010-06-10 9:15' AND termin_od>='2010-06-10 10:00'
AND termin_do>='2010-06-10 9:15' AND termin_do>='2010-06-10 10:00')
OR (termin_od<='2010-06-10 9:15' AND termin_od<='2010-06-10 10:00'
AND termin_do<='2010-06-10 9:15' AND termin_do<='2010-06-10 10:00'))))
```

#### 4.2.2 Ćwiczenie 33b – Przeglądanie rezerwacji sal wykładowych

Program powinien:

- Pobierać dane o grupach studenckich
- Umożliwiać filtrowanie grup poprzez podanie semestru studiów
- Pobierać dane o dostępnych salach
- Umożliwić podanie daty, dla której mają być wyświetlone wykonane rezerwacje
- Wyświetlić aktualne rezerwacje z uwzględnieniem podanych warunków
  - o Uwaga, nie jest konieczne podanie wszystkich warunków. Np. jeśli wybrany zostanie tylko semestr studiów – wyświetlone mają być wszystkie rezerwacje dla danego wszystkich grup danego semestru. Jeśli wybrana będzie tylko data – wszystkich rezerwacji dla wszystkich sal na dany dzień. Jeśli wybrana będzie sala i data – wszystkich rezerwacji danej sali na dany dzień itp. itd.

Wygląd interfejsu programu pokazany jest na Rys. 24



Rys. 24

Przy pobieraniu danych z bazy danych konieczne będzie łączenie tabel oraz odpowiednie określenie warunków filtrowania krotek. Przykład zapytania podającego informacje z połączonych tabel (ale bez filtrowania) podany jest poniżej:

```
SELECT * FROM RezSal.dbo.TRezerwacje
JOIN RezSal.dbo.TSale ON ID_sali=TSale.ID
JOIN RezSal.dbo.TGrupy ON ID_grupy=TGrupy.ID
```

Pozostała uwagi, dotyczące np. sposobu rozwiązania problemu pobierania informacji o grupach studenckich i ich filtracji podano w opisie programu do składania rezerwacji.