Wykład 7. Klasy i obiekty

1. Strumieniowe wyjście i wejście

1.1. Operatory strumieniowe

W części wykładu, która jest poświęcona programowaniu obiektowemu, zamiast funkcji *printf* i *scanf* będziemy stosować operatory strumieniowe:

wyjściowy: **cout <<**

wejściowy: **cin >>**

Stosując te operatory w środowisku Dev C++, należy włączyć zbiór nagłówkowy (bez rozszerzenia **.h**):

```
#include <iostream>
```

oraz dopisać na początku programu deklarację:

```
using namespace std;
```

Przykład stosowania operatorów strumieniowych

```
#include <stdio.h>
#include <conio.h>
int main()
{
int x,y;
printf(”Podaj x,y: ”);
scanf(”%d%d”,&x,&y);
printf(”%d+%d=%d”x,y,x+y);
getch();
return 0;
}
```

```
#include <iostream>>
#include <conio.h>
using namespace std;
int main()
{
 int x,y;
 cout<<”Podaj x,y: ”;
 cin>>x>>y;
 cout<<x<<’+’<<y<<’=’
     <<x+y;
 getch();
 return 0;
}
```

- Każdy element wchodzący lub wychodzący ze strumienia ma własny operator **<<** lub **>>**.
- Typy danych są rozpoznawane automatycznie, więc nie potrzeba kodów formatujących.

1.2. Domyślne formaty wyprowadzanych danych

- Liczby typów *float*, *double* wyprowadzane z precyzją 6 pozycji znaczących.
- Zbędne zera na końcu pomijane (np. **7.0** jest wyprowadzane jako **7**).
- Liczby **>999999** wyprowadzane w notacji wykładniczej.
- Wypisanie zmiennej typu **char*** jest rozumiane jako wyprowadzenie łańcucha, na który wskazuje ta zmienna.

1.3. Niektóre sposoby zmiany formatu danych

a/ precyzja (liczba cyfr znaczących)

```
cout.precision(8);
cout<<M_PI;//wypis 3.1415927
```

b/ szerokość pola (liczba pozycji zajętych przez wydruk danej)

```
cout.width(10);
cout<<M_PI; //3 spacje+7 znaków liczby
```

2. Cechy programowania obiektowego

Enkapsulacja

Łączenie w jednej wspólnej strukturze zwanej klasą:

- *zmiennych*, opisujących pewien obiekt
- *funkcji*, operujących na tych danych

Klasa jest opisem budowy *zmiennej obiektowej* (w skrócie: *obiektu*), tworzonej w pamięci komputera po jej zadeklarowaniu.

Zalety enkapsulacji:

- Ułatwia modelowanie rzeczywistych obiektów.
- Upraszcza zapis funkcji.
- Zmniejsza ryzyko popełnienia błędów logicznych.

•

Hermetyzacja

Składowe obiektu (funkcje i zmienne) mogą się różnić pod względem dostępności.

Składowe *prywatne* dostępne tylko dla funkcji składowych danej klasy.
Składowe *publiczne* są dostępne także dla funkcji zewnętrznych, zdefiniowanych poza daną klasą.

Zaleta hermetyzacji:

- Zmniejszenie ryzyka utraty istotnych danych na skutek błędów logicznych lub nieprzewidzianych zdarzeń przy wykonywaniu programu.

Dziedziczenie

Tworzona klasa może być klasą *pochodną* w stosunku do innej, wcześniej zdefiniowanej, klasy *podstawowej*.

Klasa pochodna dziedziczy, czyli przejmuje jako własne, wybrane elementy klasy podstawowej.

Zalety dziedziczenia:

- Łatwa modyfikacja klasy
- Odwzorowanie relacji pomiędzy rzeczywistymi obiektami (hierarchia, klasyfikacja)

\

Polimorfizm

W hierarchii klas dziedziczących definiuje się tzw. funkcje *wirtualne*.

Adres funkcji wirtualnej jest wybierany nie w czasie kompilacji, ale podczas wykonywania programu.

Zostaje wybrana ta z funkcji wirtualnych, która jest skladowa klasy obiektu, który tę funkcję wywołuje.

Zaleta polimorfizmu:

- Ułatwione sterowanie wyborem funkcji w złożonych programach obiektowych

3. Definicja klasy

Przykład: Definicja klasy o nazwie *circle*, która zawiera zmienne i funkcje, związane z obliczeniem obwodu i powierzchni koła:

```
class circle
  {
   private://składowe dost. w klasie
   double radius;
   double area, circumference;
   void calculate();
   void show_results();
   public://składowe dostępne ogólnie
   void get_radius();
   void process();
  };
```

Uwaga: Na końcu musi być średnik!

Komentarz:

a/ zmienne

Wszystkie zmienne są tutaj prywatne.
Zmienna *radius* przechowuje daną wejściową – promień koła.

Zmienne *area* i *circumference* służą do przechowania wyników obliczeń.

b/ Prototypy (deklaracje) funkcji:

Funkcje prywatne:

Funkcja *calculate*() oblicza pole oraz obwód koła.
Zadaniem funkcji *show_results*() jest wyprowadzenie na ekran wyników obliczeń.

Funkcje publiczne:

Funkcja *get_radius*() pozwala wprowadzić wartość promienia. Drugą funkcją publiczną jest funkcja *process*(). W jej bloku są kolejno wywoływane funkcje prywatne *calculate*() oraz *show_results*().

Wszystkie funkcje klasy *circle* są bezargumentowe. Argumenty najczęściej są zbyteczne – funkcje operują tylko na zmiennych lub funkcjach własnej klasy.

4. Definicje funkcji składowych

Wewnątrz pokazanej definicji klasy wpisano tylko prototypy funkcji. Jest dopuszczalne wpisywanie do definicji klasy pełnych definicji funkcji, jeśli nie zawierają one instrukcji powtarzających.

Definicje funkcji składowych klasy, które są umieszczane za definicją klasy, muszą zawierać, po nazwie typu funkcji, nazwę klasy, w której znajduje się prototyp funkcji. Po nazwie

klasy piszemy dwa dwukropki, a po nich – nazwę funkcji i pozostałą część nagłówka. Pokazano to poniżej.

```
class circle
      {
       private://składowe dostępne w klasie
       double radius;
       double area, circumference;
       void calculate();
       void show_results();
       public://składowe ogólnie dostępne
       void get_radius();
       void process();
      };

void circle::get_radius()
  {
   cout<<"Enter radius: ";
   cin>>radius;
  }

void circle::calculate()
   {
    area= M_PI*pow(radius,2.0);
    circumference=2*M_PI*radius;
   }

void circle::show_results()
  {
   cout<<"\nHere are the circle parameters: "
       <<endl;
   cout<<"  Radius="<<radius<<endl;
   cout<<"  Area="<<area<<endl;
   cout<<"  Circumference="<<circumference<<endl;
  }

void circle::process()
  {
   calculate();
   show_results();
  }
```

5. Deklarowanie obiektów danej klasy

Po nazwie klasy piszemy nazwę deklarowanej zmiennej obiektowej.

Można zadeklarować wiele obiektów danej klasy.
Każdy z nich zajmie określony rozmiar pamięci, w którym będą zapamiętane zmienne danego obiektu. Natomiast funkcje składowe klasy występują w pamięci tylko w *jednym* egzemplarzu. Każda z tych funkcji obsługuje wszystkie obiekty danej klasy.

Deklaracja dwóch obiektów klasy *circle*, nazwanych *c*1, *c*2:

```
class circle c1,c2;
```

Słowo kluczowe *class* można pominąć, pisząc:

```
circle c1,c2;
```

Obiekty mogą być elementami tablicy. Deklaracja tablicy *tab*, zawierającej sto obiektów klasy *circle*, wygląda następująco:

```
circle tab[100];
```

6. Odwoływanie się do składowych obiektu

Stosujemy jeden z dwóch *operatorów wyboru składowej*:

(a) Jeżeli obiekt jest określony przez *nazwę*, stosujemy operator w postaci kropki, wpisywanej pomiędzy nazwą obiektu a nazwą składowej, na przykład wywołanie funkcji *process* dla obiektu *c*1 ma postać:

```
c1.process();
```

(b) Jeżeli obiekt jest określony przez *zmienną wskaźnikową* (która przechowuje *adres* tego obiektu), to używamy operatora złożonego ze znaków **->**. Operator ten wpisujemy pomiędzy nazwą zmiennej wskaźnikowej a nazwą składowej, jak poniżej:

```
circle *wsk=&c1;
wsk->process();
```

Pełny tekst programu operującego na obiektach klasy *circle*

```
//circle.cpp
#include <iostream>
#include <conio.h>
#include <math.h>

using namespace std;

class circle
   {
    private:
      double radius;
      double area,circumference;
      void calculate();
      void show_results();
    public:
      void get_radius();
      void process();
   };

void circle::get_radius()
  {
    cout<<"\nEnter radius: ";
    cin>>radius;
  }

void circle::calculate()
  {
   area= M_PI*pow(radius,2.0);
   circumference=2*M_PI*radius;
  }

void circle::show_results()
  {
   cout<<"\nHere are the circle parameters: "
       <<endl;;
   cout<<"  Radius="<<radius<<endl;
   cout<<"  Area="<<area<<endl;
   cout<<"  Circumference="<<circumference<<endl;
```

```
  }

void circle::process()
  {
   calculate();
   show_results();
  }

int main()
  {
   circle c1,c2; //utworzenie 2 obiektów
   c1.get_radius();//operacje na obiekcie c1
   c1.process();
   c2.get_radius();//operacje na obiekcie c2
   c2.process();
   getch();
   return 0;
  }
```

Przykładowy wydruk wyników dzialania programu *circle.cpp*

```
Enter radius: 1.0

Here are the circle parameters:
  Radius=1
  Area=3.14159
  Circumference=6.28319

Enter radius: 7.5

Here are the circle parameters:
  Radius=7.5
  Area=176.715
  Circumference=47.1239
```

7. Konstruktory

7.1. Przeznaczenie konstruktora

Do utworzenia w pamięci zmiennej obiektowej jest wykorzystywana funkcję, zwaną konstruktorem obiektu.

Konstruktor domyślny włącza się automatycznie w wyniku zadeklarowania obiektu.

Można w klasie zdefiniować konstruktor własny. Wtedy ten konstruktor zostanie użyty do utworzenia obiektu, a konstruktor domyślny pozostanie nieaktywny.

Konstruktor własny pozwala na dokonanie potrzebnych operacji, związanych z uruchomieniem programu. Może na przykład wyczyścić ekran, ustawiając jednocześnie odpowiednie kolory tła i tuszu. Może dodatkowo wypisać na ekranie odpowiedni tekst.

Najczęstszym przeznaczeniem konstruktora własnego jest nadanie zmiennym obiektu odpowiednich wartości początkowych.

7.2. Definiowanie konstruktorów

Konstruktory posiadają specyficzne własności:

- W prototypie, a także w nagłówku definicji konstruktora pomija się nazwę typu (nawet słowo *void*).
- Nazwą konstruktora jest nazwa klasy.
- W przypadku zdefiniowania kilku konstruktorów, mają one tę samą nazwę, lecz różnią się argumentami.
- Konstruktora nie wolno wywołać tak, jak inne funkcje składowe. Jest on wywoływany samoczynnie po napoltkaniu odpowiednio napisanej deklaracji zmiennej obiektowej. Jeżeli jest kilka różnych konstruktorów, to wybór rodzaju konstruktora następuje na podstawie rozpoznania argumentów, użytych w deklaracji obiektu.

<u>Przykład</u>: Konstruktory dla klasy *circle*

Przeznaczeniem obu zdefiniowanych konstruktorów jest:

- nadanie określonej wartości zmiennej *radius* przy tworzeniu obiektu klasy *circle*
- wyprowadzenie na ekran komunikatu o utworzeniu obiektu.

Konstruktor **`circle(double)`** ma argument *r*, przez który przekazujemy żądaną wartość zmiennej *radius*.

Konstruktor bezargumentowy **`circle()`** umożliwia wprowadzenie wartości zmiennej *radius* z klawiatury.

Oczywiście, wszystkie konstruktory muszą być deklarowane jako *public*, aby były dostępne w *main*.

```
class circle
   {
    private:
    double radius;
    double area,circumference;
    void calculate();
    void show_results();
    public:
    void process();
    circle(double);//konstr. z argumentem
    circle();      //konstr. bez argumentu
    ~circle();     //destruktor
   };
```

7.3. Dwie postacie konstruktora z argumentami

postać zwyczajna:

```
circle::circle(double r)
{
 radius=r;
 cout<<"The object is ready with radius="
     <<radius<<endl;
 }
```

postać z listą inicjującą:

```
circle::circle(double r):radius(r)
{
 cout<<"The object is ready with radius="
     <<radius<<endl;
}
```

7.4. Wywołania konstruktorów własnych

a/ konstruktor bezargumentowy

```
//sposób 1:
   circle c1=circle();
//sposób 2:
   circle c1;
```

b/ konstruktor z argumentami

```
//sposób 1:
   circle c2=circle(3.0);
//sposób 2:
   circle c2(3.0);
```

8. Destruktory

8.1. Przeznaczenie destruktora

Destruktor jest funkcją, która służy do usunięcia obiektu z pamięci po zakończeniu programu.

W momencie zakończenia programu jest samoczynnie wywoływany destruktor domyślny, którego nie deklaruje się w definicji klasy.

Można zdefiniować destruktor własny, by przed usunięciem obiektu wykonać operacje kończące (czyszczenie ekranu, komunikat kończący itp.) Destruktor może na przykład oczyścić ekran, przywrócić domyślne atrybuty tekstu, wypisać na ekranie komunikat o zakończeniu programu itp.

Destruktor własny przesłania destruktor domyślny.

8.2. Definiowanie destruktorów

Destruktor własny nie ma typu.

Nazwą destruktora jest nazwa klasy, w której został zadeklarowany, poprzedzona znakiem tyldy ~, na przykład:

```
~circle(); //prototyp destruktora klasy circle
```

Definicja tego destruktora, którego dodatkowym zadaniem jest wydruk komunikatu, może wyglądać, jak następuje:

```
circle::~circle()
{
 cout<<"\nTu destruktor. Usuwam obiekt.";
 cout<<" Nacisnij klawisz.";
 getch();
}
```

Przykład Program obiektowy obliczający parametry koła – wersja z konstruktorami własnymi i destruktorem

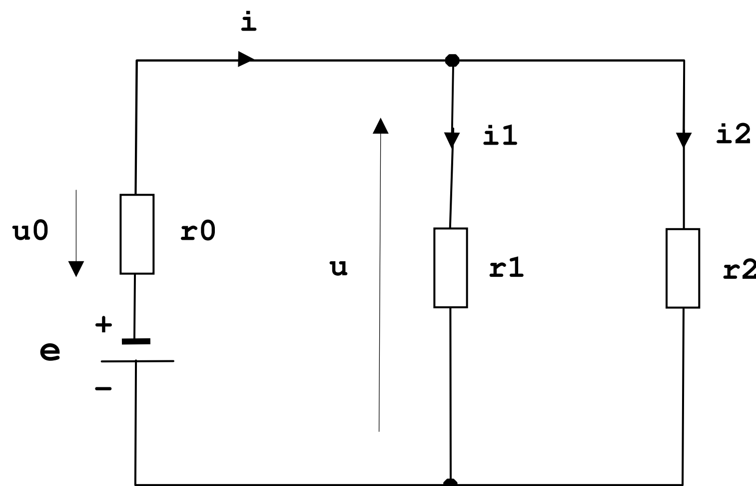
```
//circle_2k.cpp
#include <iostream>
#include <conio.h>
#include <math.h>

using namespace std;

class circle
   {
    private:
      double radius;
      double area,circumference;
      void calculate();
      void show_results();
    public:
      void process();
      circle(double);//konstruktor
      circle();      //konstruktor
      ~circle();     //destruktor
   };
```

```
circle::circle() //konstruktor
 {
  cout<<"Enter radius for the object: ";
  cin>>radius;
  cout<<"I've made the object with radius="
      <<radius<<endl;
 }

circle::circle(double r):radius(r) //konstruktor
 {
  cout<<"I've made the object with radius="
      <<radius<<endl;
 }

circle::~circle()  //destruktor
 {
  cout<<"I am removing the object with radius="
      <<radius<<". Good bye! Hit a key."<<endl;
  getch();
 }

void circle::calculate()
   {
    area= M_PI*pow(radius,2.0);
    circumference=2*M_PI*radius;
   }

void circle::show_results()
  {
    cout<<"\nHere are the circle parameters: ";
    cout<<"\n  Radius="<<radius;
    cout<<"\n  Area="<<area;
    cout<<"\n  Circumference="<<circumference
        <<endl<<endl;;
  }

void circle::process()
  {
   calculate();
   show_results();
  }
```

```
int main()
  {
   circle c1;      //konstrukcja obiektu c1
   c1.process();   //operacja na obiekcie c1

   circle c2(3.0); //konstrukcja obiektu c2
   c2.process();   //operacja na obiekcie c2

   return 0;
  }
```

9. Przykłady programów

9.1. Pokazany niżej program obiektowy służy do obliczania napięć, prądów i mocy w obwodzie prądu stałego, widocznym na rysunku. Klasa *obwód* zawiera zmienne i funkcje potrzebne do obliczeń, a także konstruktor i destruktor własny. Konstruktor ma dwa argumenty, przez które przy tworzeniu obiektu są wprowadzane wartości rezystancji *r*1, *r*2, wczytane wcześniej z klawiatury za pomocą funkcji *dane*. Wartości siły elektromotorycznej *e* oraz rezystancji źródła *r*0, jako wspólne dla wszystkich obiektów, ustalane są za pomocą listy inicjacyjnej konstruktora. Konstruktor wyprowadza także komunikat o utworzeniu obiektu. Dodatkowym zadaniem destruktora jest natomiast wypisanie komunikatu o usunięciu obiektu po zakończeniu programu.



Obwód prądu stałego będący obiektem obliczeń w przykładzie 9.1

```
#include <stdio.h>
#include <conio.h>

class obwod
   {
    double e,r0,r1,r2;
    double u0,u,i,i1,i2,p,p0,p1,p2;
    void oblicz();
    public:
    void wyniki();
    obwod(double,double);  //konstruktor
    ~obwod();              //destruktor
   };


void  obwod::oblicz()
   {
    i=e/(r0+r1*r2/(r1+r2));
    u0=i*r0;
    u=e-u0;
    i1=u/r1;
    i2=u/r2;
    p0=u0*i;
    p1=u*i1;
    p2=u*i2;
    p=e*i;
   }


void obwod::wyniki()
{
 oblicz();
 printf("\nDane:\ne =%.3le  r0=%.3le  ",e,r0);
 printf("r1=%.3le  r2=%.3le",r1,r2);
 printf("\nWyniki obliczen:");
 printf("\ni =%.3le  i1=%.3le  i2=%.3le",
        i,i1,i2);
 printf("\nu0=%.3le  u =%.3le",u0,u);
 printf("\np0=%.3le  p1=%.3le  ",p0,p1);
 printf("p2=%.3le  p =%.3le\n",p2,p);
}
```

```
obwod::obwod(double _r1,double _r2):
                  e(12),r0(1),r1(_r1),r2(_r2)
    {
     printf("Tu konstruktor. Obiekt utworzony:");
    }
obwod::~obwod()
    {
     printf("\nTu destruktor. Usuwam obiekt.”);
     printf(„Nacisnij klawisz.");
     getch();
    }

void dane(double &,double &);
int main()
   {
    double r1,r2;
    dane(r1,r2);
    obwod obw1(r1,r2);
    obw1.wyniki();
    dane(r1,r2);
    obwod obw2(r1,r2);
    obw2.wyniki();
    return 0;
   }

void dane(double &r1, double &r2)
  {
    printf("\nPodaj r1, r2: ");
    scanf("%lf%lf",&r1,&r2);
  }
```

Klasa *obwod* zawiera funkcję publiczną *wyniki*. Funkcja ta po utworzeniu obiektu zostaje wywołana w *main*. Funkcja *wyniki* w swoim ciele wywołuje funkcję prywatną *oblicz*, by za jej pomocą określić wartości parametrów obwodu. Następnie wartości tych parametrów wyprowadza na ekran. W *main* przeprowadzono obliczenia dla dwóch obiektów klasy *obwod*, któ-

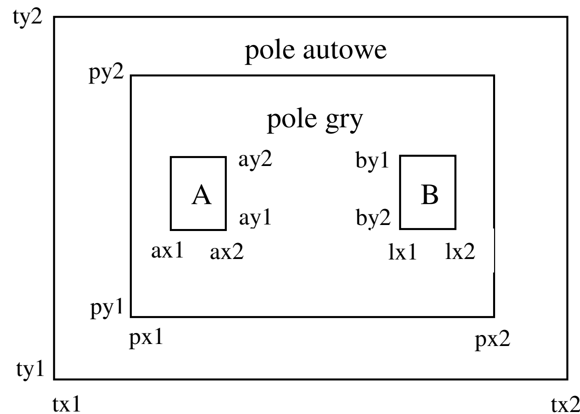
rymi są zmienne *obw*1, *obw*2. Poniżej pokazano przykładową postać wydruku wyników działania programu.

```
Podaj r1, r2: 10.0      20.0
Tu konstruktor. Obiekt utworzony.
Dane:
e =1.200e+01  r0=1.000e+00  r1=1.000e+01  r2=2.000e+01
Wyniki obliczen:
i =1.565e+00  i1=1.043e+00  i2=5.217e-01
u0=1.565e+00  u =1.043e+01
p0=2.450e+00  p1=1.089e+01  p2=5.444e+00  p =1.878e+01

Podaj r1, r2: 12.0      32.5
Tu konstruktor. Obiekt utworzony.
Dane:
e =1.200e+01  r0=1.000e+00  r1=1.200e+01  r2=3.250e+01
Wyniki obliczen:
i =1.229e+00  i1=8.976e-01  i2=3.314e-01
u0=1.229e+00  u =1.077e+01
p0=1.510e+00  p1=9.668e+00  p2=3.570e+00  p =1.475e+01

Tu destruktor. Usuwam obiekt. Nacisnij klawisz.
Tu destruktor. Usuwam obiekt. Nacisnij klawisz.
```

Wydruk otrzymany po wykonaniu programu z przykładu 9.1

9.2. Następny przykład to program obiektowy, który symuluje strzały do dwóch bramek. Rysunek poniżej pokazuje szkic terenu gry, podzielonego na cztery strefy: pole autowe, pole gry oraz dwie bramki, oznaczone symbolami *A*, *B*. Wszystkie te elementy terenu mają kształt prostokątów położonych na płaszczyźnie *x*, *y*. Współrzędne brzegowe krawędzi elementów oznaczone są na rysunku odpowiednimi symbolami. W definicji klasy *gra* występują zmienne o identycznych nazwach, przechowujące wartości współrzędnych, ustalane przez konstruktor zgodnie z jego listą inicjacyjną. Konstruktor inicjuje także wartość stałej *N*, określającej liczbę zagrań, po której mecz ulega zakończeniu. Zmienne *x*, *y* służą do przechowania wylosowanych w czasie kolejnego zagrania współrzędnych punktu trafienia, natomiast zmienna *ile_zagr* jest licznikiem

zagrań, zerowanym wstępnie przez konstruktor. Zmienne *b*1, *b*2, *aut*, *pudlo* służą jako liczniki, pamiętające wyniki gry: liczbę trafień do bramki *A*, liczbę trafień do bramki *B*, liczbę trafień do strefy autowej oraz liczbę strzałów, które trafiły w pole gry, ale nie do bramki. Wszystkie zmienne klasy są typu *int*, ponieważ przyjęto, że współrzędne terenu gry oraz losowane położenia piłki będą całkowite.



Strefy pola gry w przykładzie 9.2 *i symbole ich współrzędnych*

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
class gra
    {
     const int N;
     int tx1,tx2,ty1,ty2,px1,px2,py1,py2;
     int ax1,ax2,ay1,ay2,bx1,bx2,by1,by2;
     int x,y,ile_zagr,b1,b2,aut,pudlo;
     void kop();
     int arbiter();
     void graj();
     public:
     void wyniki();
     gra();  //konstruktor
    };
```

```
int gra::arbiter()
{
 if(x>=ax1&&x<=ax2&&y>=ay1&&y<=ay2) return 1;
 else if(x>=bx1&&x<=bx2&&y>=by1&&y<=by2)
      return 2;
      else if(x>=px1&&x<=px2&&y>=py1&&y<=py2)
           return -1;
                 else return 0;
    }
void gra::kop()
   {
    x=tx1+rand()%(tx2-tx1+1);
    y=ty1+rand()%(ty2-ty1+1);
    ile_zagr++;
   }
void gra::graj()
   {
    int w;
    b1=b2=aut=pudlo=0;
    srand(time(NULL));
    while (ile_zagr<N)
      {
       kop();
       w=arbiter();
       if (w==1) b1++;
       else if (w==2) b2++;
            else if (w==-1) pudlo++;
                 else aut++;
      }
   }
void gra::wyniki()
   {
    graj();
    printf("\nLiczba goli w bramce A: %d",b1);
    printf("\nLiczba goli w bramce B: %d",b2);
    printf("\nLiczba autow: %d",aut);
    printf("\nLiczba nietrafionych strzalow: %d",
            pudlo);
    printf("\nLiczba zagran: %d",ile_zagr);
   }
gra::gra(): tx1(0),tx2(70),ty1(0),ty2(50),
            px1(10),px2(60),py1(10),py2(40),
```

```
             ax1(15),ax2(20),ay1(20),ay2(30),
             bx1(50),bx2(55),by1(20),by2(30),
             ile_zagr(0),N(100)  {}
int main()
   {
    gra mecz=gra();
    mecz.wyniki();
    getch();
    return 0;
   }
```

Symulacja jest wykonywana z udziałem trzech funkcji składowych. Funkcja *kop* losuje parę współrzędnych punktu trafienia. Funkcja *arbiter* bada, w której ze stref terenu gry znalazła się piłka i w zależności od tego zwraca jedną z czterech możliwych wartości. Obie te pomocnicze funkcje wykorzystano w ciele funkcji *graj*, która przeprowadza właściwą symulację gry. Funkcja *graj* zawiera pętlę *while,* wykonującą *N* kroków – w każdym z nich wywołuje się funkcję *kop* oraz funkcję *arbiter*. Zależnie od wartości przekazanej przez funkcję *arbiter*, zostaje zwiększony o 1 stan jednego z czterech liczników przechowujących wyniki gry: *b*1, *b*2, *aut* lub *pudlo*. Funkcja publiczna *wyniki* uruchamia w swoim ciele funkcję *graj*, po czym wyprowadza na ekran rezultaty symulacji w postaci przedstawionej na rysunku:

```
Liczba goli w bramce A: 2
Liczba goli w bramce B: 1
Liczba autow: 55
Liczba nietrafionych strzalow: 42
Liczba zagran: 100
```

Wydruk uzyskany po wykonaniu programu z przykładu 9.2