



# Podstawy Programowania

dr inż. Tomasz Marciniak

# Plan wykładu

- Metody i dane statyczne,
- Funkcje wirtualne,

# Cechy programowania obiektowego

- Polimorfizm,
- Abstrakcja danych,
- Ukrywanie danych,
- Dziedziczenie.

# Metody i dane statyczne

- Możemy mieć kilka obiektów danej klasy,
- Każdy obiekt ma swoją własną kopie danych,
- Co zrobić, jeśli chcemy aby dla każdego obiektu istniała tylko jedna kopia zmiennej?

# Metody i dane statyczne

- Możemy mieć kilka obiektów danej klasy,
- Każdy obiekt ma swoją własną kopie danych,
- Co zrobić, jeśli chcemy aby dla każdego obiektu istniała tylko jedna kopia zmiennej?

**Zmienne globalne ☹**

**Zmienne statyczne ☺**

# Dane statyczne

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6      int y;
7      static int z;
8  };
9  int test::z;
10 int main(){
11     cout << test::z << endl;
12     test::z=10;
13     cout << test::z << endl;
14     getche();
15     return 0;
16 }
```

# Dane statyczne

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{ ← Deklaracja
5      public:      klasy test
6      int y;
7      static int z;
8  };
9  int test::z;
10 int main(){
11     cout << test::z << endl;
12     test::z=10;
13     cout << test::z << endl;
14     getche();
15     return 0;
16 }
```

# Dane statyczne

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6          int y;
7          static int z;
8  };
9  int test::z;
10 int main(){
11     cout << test::z << endl;
12     test::z=10;
13     cout << test::z << endl;
14     getch();
15     return 0;
16 }
```

Deklaracja zmiennej y

Deklaracja zmiennej statycznej z



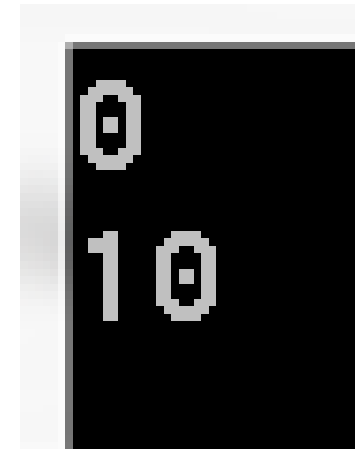
# Dane statyczne

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6          int y;
7          static int z;
8  };
9  int test::z;
10 int main(){
11     cout << test::z << endl;
12     test::z=10;
13     cout << test::z << endl;
14     getche();
15     return 0;
16 }
```

Deklaracja zmiennej **z**  
poza klasą, **BEZ**  
**SŁOWKA STATIC**

# Dane statyczne

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6      int y;
7      static int z;
8  };
9  int test::z;
10 int main(){
11     cout << test::z << endl;
12     test::z=10;
13     cout << test::z << endl;
14     getch();
15     return 0;
16 }
```



Jakie wnioski ?

# Dane statyczne - wnioski

- Składowe statyczne istnieją nawet jak nie ma żadnego obiektu,
- Składowa statyczna jest inicjowana przez domniemanie wartością „0”,
- Jeśli chcemy mieć dostęp do składowej statycznej poza klasą musimy ją zadeklarować bez użycia słowa *static* i poprzedzając ją nazwą klasy i „::”.

# Dane statyczne – przykład I

Jak to zadziała ?

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6      int y;
7      static int z;
8  };
9  int test::z;
10 int main(){
11     cout << test::z << " " << test::y << endl;
12     test::z=10;
13     cout << test::z << endl;
14     getch();
15     return 0;
16 }
```

# Dane statyczne – przykład I

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6      int y;
7      static int z;
8  };
9  int test::z;
10 int main(){
11     cout << test::z << " " << test::y << endl;
12     test::z=10;
13     cout << test::z << endl;
14     getch();
15     return 0;
16 }
```

[Error] invalid use of non-static data member 'test::y'

# Dane statyczne – przykład 2

Jak zadziała ?

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      private:
6          int y;
7          static int z;
8  };
9  int test::z;
10 int main(){
11     cout << test::z << endl;
12     test::z=10;
13     cout << test::z << endl;
14     getch();
15     return 0;
16 }
```

# Dane statyczne – przykład 2

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      private:
6          int y;
7          static i
8  };
9  int test::z;
10 int main(){
11     cout << test::z << endl;
12     test::z=10;
13     cout << test::z << endl;
14     getch();
15     return 0;
16 }
```

[Error] 'int test::z' is private

# Przykład 3

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6          int y;
7          static int z;
8          test(){z+=1;}
9  };
10 int test::z;
11 int main(){
12     test t1,t2,t3;
13     cout << test::z << endl;
14     getche();
15     return 0;
16 }
```

Co robi ten kod ?



# Przykład 3

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6          int y;
7          static int z;
8          test(){z+=1;}
9  };
10 int test::z;
11 int main(){
12     test t1,t2,t3;
13     cout << test::z << endl;
14     getche();
15     return 0;
16 }
```

3

# Przykład 4

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6          static int z;
7          test(){z+=1;}
8          ~test(){z-=1;}
9  };
10 int test::z;
11 int main(){
12     test* t1= new test();
13     test* t2= new test();
14     test* t3= new test();
15     cout << t1->z << endl;
16     delete t3;
17     cout << t1->z << endl;
18     getch();
19     return 0;
20 }
```

**z = ?**

# Przykład 4

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test{
5      public:
6          static int z;
7          test(){z+=1;}
8          ~test(){z-=1;}
9  };
10 int test::z;
11 int main(){
12     test* t1= new test();
13     test* t2= new test();
14     test* t3= new test();
15     cout << t1->z << endl;
16     delete t3;
17     cout << t1->z << endl;
18     getch();
19     return 0;
20 }
```

3  
2

# Metody statyczne - ograniczenia

- Statyczne funkcje składowe mogą odwoływać się bezpośrednio jedynie do **danych statycznych klasy i do danych globalnych**,
- Nie można tworzyć statycznych funkcji wirtualnych.

# Przykład I

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  int x=5;
5  class test{
6      public:
7          int y;
8          static int z;
9          static int oblicz(){return (z+1)*x;}
10 };
11 int test::z;
12 int main(){
13     cout << test::oblicz() << endl;
14     getche();
15     return 0;
16 }
```

Deklaracja zmiennej globalnej **x**

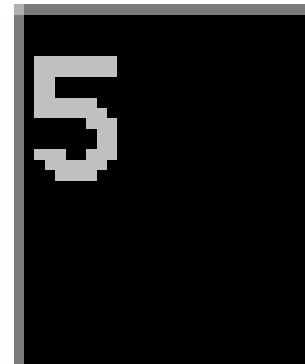
Deklaracja zmiennej statycznej **z**

Deklaracja metody statycznej **oblicz()**

Wywołanie metody statycznej **oblicz()** bez tworzenia obiektu

# Przykład

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  int x=5;
5  class test{
6      public:
7          int y;
8          static int z;
9          static int oblicz(){return (z+1)*x;}
10 };
11 int test::z;
12 int main(){
13     cout << test::oblicz() << endl;
14     getch();
15     return 0;
16 }
```



# Funkcje wirtualne

- Zapewniają osiągnięcie polimorfizmu w trakcie wykonywania programu,
- Jeśli funkcja zadeklarowana jest w klasie bazowej jako wirtualna to na etapie kompilacji tworzony jest wskaźnik do tej funkcji lecz nie przypisywana jest mu wartość,
- Wartość ta ustalana jest w trakcie wykonywania programu w zależności który obiekt wywołuje metodę.

# Funkcje wirtualne

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test1{
5      public:
6      int z;
7      test1(){z=5;}
8      virtual int wartosc(){return z;}
9  };
10 class test2:test1{
11     public:
12     int z1;
13     test2(int x){z1=x;}
14     virtual int wartosc(){return z1*test1::wartosc();}
15 };
16 int main(){
17     test2* t2=new test2(10);
18     cout << t2->wartosc() << endl;
19     delete t2;
20     getch();
21     return 0;
22 }
```



# Funkcje wirtualne

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class test1{
5      public:
6      int z;
7      test1(){z=5;}
8      virtual int wartosc(){return z;}
9  };
10 class test2:test1{
11     public:
12     int z1;
13     test2(int x){z1=x;}
14     virtual int wartosc(){return z1*test1::wartosc();}
15 };
16 int main(){
17     test2* t2=new test2(10);
18     cout << t2->wartosc() << endl;
19     delete t2;
20     getch();
21     return 0;
22 }
```



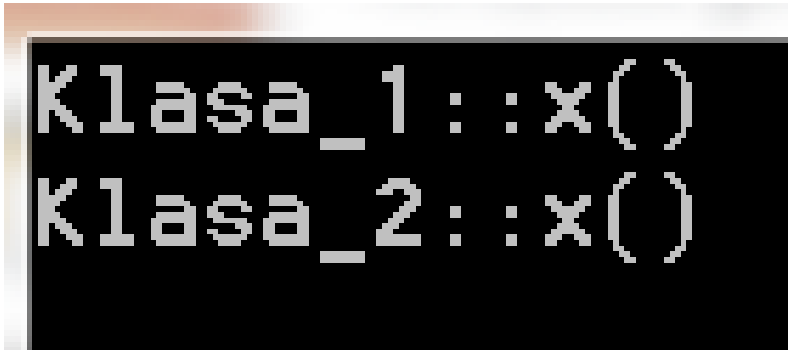
50

# Funkcje wirtualne

```
1  #include <stdio>
2  class Klasa_1{
3      public:
4          Klasa_1(){x();}
5          virtual void x(){
6              printf( "Klasa_1::x()\n" );
7          }
8      };
9  class Klasa_2: public Klasa_1{
10     public:
11         Klasa_2(){}
12         virtual void x(){
13             printf( "Klasa_2::x()\n" );
14         }
15     };
16  int main(){
17     Klasa_2 K2;
18     K2.x();
19     return 0;
20 }
```

# Funkcje wirtualne

```
1  #include <stdio>
2  class Klasa_1{
3  public:
4      Klasa_1(){x();}
5      virtual void x(){
6          printf( "Klasa_1::x()\n" );
7      }
8  };
9  class Klasa_2: public Klasa_1{
10 public:
11     Klasa_2(){}
12     virtual void x(){
13         printf( "Klasa_2::x()\n" );
14     }
15 };
16 int main(){
17     Klasa_2 K2;
18     K2.x();
19     return 0;
20 }
```



```
Klasa_1::x()
Klasa_2::x()
```

# Funkcje wirtualne

- Jeżeli wywołujemy w konstruktorze klasy pochodnej funkcję wirtualną, to będzie wywołana funkcja wirtualna klasy bazowej,
- Działają trochę wolniej niż zwykła metody,
- Zaleca się aby wszystkie destruktory w klasach bazowych były wirtualne !!!.

# Destruktory wirtualne – przykład. I

```
1  #include <iostream>
2  using namespace std;
3  class Klasa_1{
4  public:
5      Klasa_1(){
6          printf( "konstruktor : Klasa_1\n" );
7      }
8      ~Klasa_1(){
9          printf( "destruktor : Klasa_1\n" );
10     }
11 };
12 class Klasa_2: public Klasa_1{
13 public:
14     Klasa_2(){
15         printf( "konstruktor : Klasa_2\n" );
16     }
17     ~Klasa_2(){
18         printf( "destruktor : Klasa_2\n" );
19     }
20 };
21 int main(){
22     Klasa_1* K2 = new Klasa_2;
23     if (K2) delete K2;
24     return 0;
25 }
```

# Destruktory wirtualne – przykład. I

```
1  #include <iostream>
2  using namespace std;
3  class Klasa_1{
4  public:
5      Klasa_1(){
6          printf( "konstruktor : Klasa_1\n" );
7      }
8      ~Klasa_1(){
9          printf( "destruktor : Klasa_1\n" );
10     }
11 };
12 class Klasa_2: public Klasa_1{
13 public:
14     Klasa_2(){
15         printf( "konstruktor : Klasa_2\n" );
16     }
17     ~Klasa_2(){
18         printf( "destruktor : Klasa_2\n" );
19     }
20 };
21 int main(){
22     Klasa_1* K2 = new Klasa_2;
23     if (K2) delete K2;
24     return 0;
25 }
```

```
konstruktor : Klasa_1
konstruktor : Klasa_2
destruktor : Klasa_1
```

# Destruktory wirtualne – przykład.2

```
1  #include <iostream>
2  using namespace std;
3  class Klasa_1{
4  public:
5      Klasa_1(){
6          printf( "konstruktor : Klasa_1\n" );
7      }
8      virtual ~Klasa_1(){
9          printf( "destruktor : Klasa_1\n" );
10     }
11 };
12 class Klasa_2: public Klasa_1{
13 public:
14     Klasa_2(){
15         printf( "konstruktor : Klasa_2\n" );
16     }
17     ~Klasa_2(){
18         printf( "destruktor : Klasa_2\n" );
19     }
20 };
21 int main(){
22     Klasa_1* K2 = new Klasa_2;
23     if (K2) delete K2;
24     return 0;
25 }
```



# Destruktory wirtualne – przykład.2

```
1  #include <iostream>
2  using namespace std;
3  class Klasa_1{
4  public:
5      Klasa_1(){
6          printf( "konstruktor : Klasa_1\n" );
7      }
8      virtual ~Klasa_1(){
9          printf( "destruktor : Klasa_1\n" );
10     }
11 };
12 class Klasa_2: public Klasa_1{
13 public:
14     Klasa_2(){
15         printf( "konstruktor : Klasa_2\n" );
16     }
17     ~Klasa_2(){
18         printf( "destruktor : Klasa_2\n" );
19     }
20 };
21 int main(){
22     Klasa_1* K2 = new Klasa_2;
23     if (K2) delete K2;
24     return 0;
25 }
```

```
konstruktor : Klasa_1
konstruktor : Klasa_2
destruktor : Klasa_2
destruktor : Klasa_1
```



# Funkcje abstrakcyjne

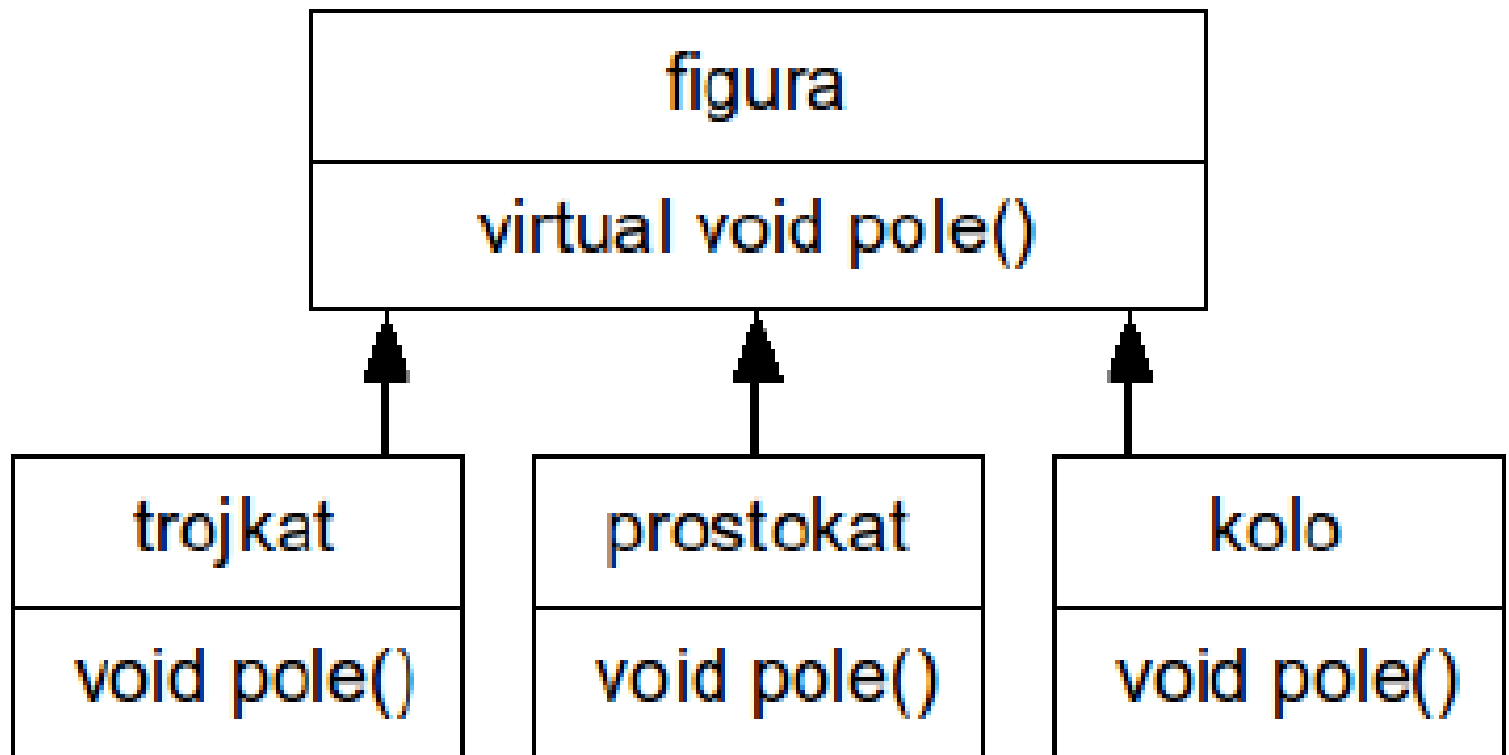
- Jeżeli nie wiemy jak zdefiniować funkcję w klasie bazowej, a wiedzę tę uzyskujemy w klasie pochodnej to nie podajemy definicji funkcji wirtualnej w klasie bazowej,
- Mechanizm ten nosi nazwę funkcji abstrakcyjnych.

***virtual typ\_zwracany nazwa\_fun()=0;***

# Funkcje abstrakcyjne

- Klasa zawierająca funkcję abstrakcyjną nosi nazwę klasy abstrakcyjnej,
- Nie można utworzyć obiektu na podstawie klasy abstrakcyjnej,
- W klasie pochodnej musi znaleźć się definicja funkcji abstrakcyjnej.

# Przykład



# Przykład

```
pole trojkata wynosi    3.000
pole prostokata wynosi  20.000
pole kola wynosi       56.520
```

```
1  #include <stdio>
2  using namespace std;
3  class figura{
4      protected:
5          float a,b;
6      public:
7          void SetAB(float x, float y){
8              a=x;b=y;
9          }
10         virtual void pole()=0;
11     };
12     class trojkat: public figura{
13     public:
14         void pole(){
15             printf( "pole trojkata wynosi %8.3f\n", a*b/2 );
16         }
17     };
18     class prostokat: public figura{
19     public:
20         void pole(){
21             printf( "pole prostokata wynosi %8.3f\n", a*b );
22         }
23     };
24     class kolo: public figura{
25     public:
26         void pole(){
27             printf( "pole kola wynosi %8.3f\n", 3.14*a*a/2 );
28         }
29     };
30     int main(){
31         trojkat t; prostokat p; kolo k;
32         t.SetAB(2,3); p.SetAB(4,5); k.SetAB(6,7);
33         t.pole(); p.pole(); k.pole();
34         return 0;
35     }
```

# Wskaźniki na funkcje

- W przeciwieństwie do normalnych wskaźników muszą też wskazywać na typ wartości zwracanej, ilość parametrów i ich typ,
- Dla wskaźników funkcyjnych nie obowiązują normalne zasady arytmetyki wskaźników.

# Wskaźniki na funkcje

```
1  #include <iostream>
2  using namespace std;
3
4  int f1(int a, int b, char* t);
5  int f2(int a, int b, char* t);
6
7  int main(){
8      int (*f)(int, int, char*);
9      f=f1;
10     (*f)(4,5,"Funkcja 1:");
11     f=&f2;
12     f(2,3,"Funkcja 2:");
13 }
14
15 int f1(int a, int b, char* t){
16     cout << t << " dodawanie = " << a+b << endl;
17 }
18 int f2(int a, int b, char* t){
19     cout << t << " mnozenie = " << a*b << endl;
20 }
```

# Wskaźniki na funkcje

```
1  #include <iostream>
2  using namespace std;
3
4  int f1(int a, int b, char* t);
5  int f2(int a, int b, char* t);
6
7  int main(){
8      int (*f)(int, int, char*);
9      f=f1;
10     (*f)(4,5,"Funkcja 1:");
11     f=&f2;
12     f(2,3,"Funkcja 2:");
13 }
14
15 int f1(int a, int b, char* t){
16     cout << t << " dodawanie = " << a+b << endl;
17 }
18 int f2(int a, int b, char* t){
19     cout << t << " mnozenie = " << a*b << endl;
20 }
```

Wskaźnik na funkcję  
trzyparametrową





# Wskaźniki na funkcje

```
1  #include <iostream>
2  using namespace std;
3
4  int f1(int a, int b, char* t);
5  int f2(int a, int b, char* t);
6
7  int main(){
8      int (*f)(int, int, char*);
9      f=f1;
10     (*f)(4,5,"Funkcja 1:");
11     f=&f2;
12     f(2,3,"Funkcja 2:");
13 }
14
15 int f1(int a, int b, char* t){
16     cout << t << " dodawanie = " << a+b << endl;
17 }
18 int f2(int a, int b, char* t){
19     cout << t << " mnozenie = " << a*b << endl;
20 }
```

Podstawienie adresu  
funkcji pod wskaźnik

Nie potrzeba operatora  
adresu



# Wskaźniki na funkcje

```
1  #include <iostream>
2  using namespace std;
3
4  int f1(int a, int b, char* t);
5  int f2(int a, int b, char* t);
6
7  int main(){
8      int (*f)(int, int, char*);
9      f=f1;
10     (*f)(4,5,"Funkcja 1:");
11     f=&f2;
12     f(2,3,"Funkcja 2:");
13 }
14
15 int f1(int a, int b, char* t){
16     cout << t << " dodawanie = " << a+b << endl;
17 }
18 int f2(int a, int b, char* t){
19     cout << t << " mnozenie = " << a*b << endl;
20 }
```

Wywołanie funkcji

Nie potrzeba dereferencji

# Wskaźniki na funkcje

```
1  #include <iostream>
2  using namespace std;
3
4  int f1(int a, int b, char* t);
5  int f2(int a, int b, char* t);
6
7  int main(){
8      int (*f)(int, int, char*);
9      f=f1;
10     (*f)(4,5,"Funkcja 1:");
11     f=&f2;
12     f(2,3,"Funkcja 2:");
13 }
14
15 int f1(int a, int b, char* t){
16     cout << t << " dodawanie = " << a+b << endl;
17 }
18 int f2(int a, int b, char* t){
19     cout << t << " mnozenie = " << a*b << endl;
20 }
```

Funkcja 1: dodawanie = 9  
Funkcja 2: mnozenie = 6



# KONIEC