

WYDZIAŁ FIZYKI
UNIWERSYTET WARSZAWSKI

Programowanie zaawansowane FM i NI

Wykład 11

Dziedziczenie klas + polimorfizm

Krzysztof Piasecki

Semestr letni roku akad. 2024-25

krzysztof.piasecki@fuw.edu.pl

- **Dziedziczenie** – to mechanizm umożliwiający zbudowanie nowej klasy *na bazie* wcześniejszej.
  Tę pierwotną nazywamy **klasą bazową** [**base class**],
  a nową – **klasą dziedziczącą** po bazowej (lub **pochodną**) [**derived class**].



⊙ Można w ten sposób utworzyć **hierarchię obiektów**, podobnie jak systematyka gatunków w biologii: klasa bazowa posiada cechy ogólne (wspólne), a kolejne klasy dziedziczące – dodatkowo cechy uszczegóławiające.

Określenie „*na bazie*” oznacza, że w klasie dziedziczącej *mogą być* dostępne pola i metody klasy bazowej, ale niekoniecznie wszystkie. Kwestia dostępności obszarów jest częściowo elastyczna dla projektanta klas.

⊙ Zaznaczmy na wstępie, że mamy prawo zażądać, aby klasa/struktura nie mogła być dziedziczona. (Od standardu C++11) piszemy wówczas:

```
class MojaKlasa final { …
```

- Prosty przykład.

  `Figura` jest strukturą bazową.
  `Okrag` i `Prostokat` to struktury dziedziczące po `Figura`.
  Fakt ten zapisujemy w nagłówku:

  `struct Okrag : Figura {...`

  Hierarchia odzwierciedla fakt, że każda figura ma pole, ale okrąg ma promień, a prostokąt – boki.

- ⊙ Tutaj wszystkie pola i metody struktury `Figura` są dostępne dla metod klas pochodnych i na zewnątrz.

  Pełną dostępność mamy dzięki wyborowi `struct` i jej domyślnej prywatności `public`, ale też temu, że w tej sytuacji domyślny **sposób dziedziczenia jest jak w `struct`**, czyli `public`.

[Link]

```
5  struct Figura {
6    float Pole;
7    void UstawPole (float _P) { Pole = _P ; }
8  };
9
10 struct Okrag : Figura {
11   float R;
12   void UstawR  (float _R)   { R = _R ;
13                               UstawPole ( M_PI * R * R ); }
14 };
15
16 struct Prostokat : Figura {
17   float A, B;
18   void UstawAB (float _A, float _B) { A = _A ; B = _B;
19                                       Pole = A * B;    }
20 };
21
22 int main () {
23   Figura F; F.UstawPole (4.) ; cout << F.Pole << endl;
24   Okrag  O; O.UstawR    (1.) ; cout << O.Pole << ' '
25                                   << O.R    << endl;
26 }
```

- Niech `Figura` stanie się klasą i ma prywatne `float Pole`.

[Link]

Ważna **zasada**:
obszar prywatny klasy bazowej nigdy nie jest dostępny wprost klasie pochodnej.
Wprawdzie klasa pochodna posiada prywatne obszary klasy bazowej, ale dostęp do nich jest nie-wprost, a np. przez metodę publiczną w klasie bazowej.

- Do nagłówka `struct Okrag` wpisalismy `public`.
To **specyfikator dostępu** (**access specifier**). Określa on, jaka w klasie pochodnej będzie **prywatność obszarów wziętych z klasy bazowej**.

Specyfikator **`public`** oznacza, że dla klasy dziedziczącej:
- ▻ obszar prywatny z klasy bazowej jest niedostępny wprost,
- ▻ inne obszary mają taką prywatność, co w klasie bazowej.

```
5  class Figura {
6    float Pole;
7   public:
8    void UstawPole (float _P) { Pole = _P ;  }
9    float PodajPole ()        { return Pole; }
10 };
11
12 struct Okrag : public Figura {
13   float R;
14   void UstawR  (float _R)   { R = _R ;
15                               UstawPole ( M_PI * R * R ); }
16 };
17
18 struct Prostokat : public Figura {
19   float A, B;
20   void UstawAB (float _A, float _B) { A = _A ; B = _B;
21                                       UstawPole ( A * B ); }
22 };
23
24 int main () {
25   Figura F; F.UstawPole (4.) ; cout << F.PodajPole() << endl;
26   Okrag  O; O.UstawR    (1.) ; cout << O.PodajPole() << ' '
27                                    << O.R    << endl;
28 }
```

- Reguła *„obszar prywatny klasy bazowej jest niedostępny wprost klasie pochodnej”* tworzy problem:
  Czasem chcemy, aby w klasie bazowej obszar był niedostępny na zewnątrz,
  ale był dostępny dla klas pochodnych.

  C++ wprowadza trzeci tryb prywatności: **`protected`** .
  Specyfikator ten, podany przed jakimiś polami/metodami, określa taką, jak powyżej formę ich prywatności.

  Ale **`protected`** można też wpisać do nagłówka klasy dziedziczącej.
  Wówczas: wzięte z klasy bazowej obszary `public` i `protected`,
  w klasie dziedziczącej będą mieć prywatność `protected`.

- **Reguły prywatności dziedziczenia obszarów**:

  ① Obszar prywatny klasy bazowej jest niedostępny wprost w klasie pochodnej

  ② Specyfikatory w nagłówku klasy pochodnej :

  - ▻ Specyfikator **`public`**: obszary k. bazowej `public` i `protected` przenoszą się bez zmian
  - ▻ Specyfikator **`private`**: obszary k. bazowej `public` i `protected` przenoszą się jako `private`
  - ▻ Specyfikator **`protected`**: obszary k. bazowej `public` i `protected` przenoszą się jako `protected`

  ③ Jeśli w nagłówku klasy pochodnej **brak specyfikatora**, to wówczas:

  - ▻ jeśli bazowa jest **`class`** : domyślny tryb przenoszenia prywatności to: `private`
  - ▻ jeśli bazowa jest **`struct`** : domyślny tryb przenoszenia prywatności to: `public`

- **Konstruktory i destruktor**: domyślnie nie dziedziczą się.

  Oznacza to, że gdy w klasie bazowej jest jakiś konstruktor, np. 1-argumentowy `(int x)`, to w klasie pochodnej znika ten sposób konstrukcji i musimy napisać taki od nowa.

⊙ Ale podczas tworzenia obiektu klasy pochodnej, domyślnie wywołany jest wpierw konstruktor `()` klasy bazowej.
Zaś podczas kasowania tego obiektu, wywoła się najpierw destruktor klasy pochodnej, a następnie - bazowej.

[Link]

```
4  struct A  {
5    A ()       { cout << "A() "; }
6   ~A ()       { cout << "~A " ; } };
7
8  struct B : A  {
9    B ()       { cout << "B() "; }
10  ~B ()       { cout << "~B " ; } };
11
12 int main ()   {
13   B b;        }
```

Efekt: `A() B() ~B ~A`

[Link]

```
4  struct A  {
5    float a;
6    A ()       { cout << "A() ";  }
7    A (int x) { cout << "A(" <<x<< ") "; }
8  };
9
10 struct B : A  {
11   B () : A( 1 ) { a = 1. ; cout << "B() "; }
12 };
13
14 int main ()                  {
15   B b1;
16 //B b2 ( 1 );   // error     }
```

Efekt: `A(1) B()`

- Możemy jednak wymusić wywołanie wybranego konstruktora klasy bazowej, jeżeli wskażemy go w liście inicjalizacyjnej przy konstruktorze klasy pochodnej.

⊙ Lista inicjalizacyjna w klasie pochodnej ma jednak mankament:

  nie można w niej wprost przypisać wartości polu klasy bazowej, nawet gdy jest publiczne.
  Można to zrobić w ciele konstruktora.

- **Konstruktory i destruktor**:
  można jednak wymusić dziedziczenie konstruktorów klasy bazowej.

⊙ W tym przykładzie
`using A::A;`
zaimportuje *wszystkie* konstruktory klasy bazowej do klasy pochodnej.

Nie można tą drogą odziedziczyć części konstruktorów, a części nie.
Można za to napisać ponownie dany konstruktor, który przesłoni ten odziedziczony. Choć shadowing – to ryzyko nieczytelności kodu...

W klasie pochodnej wciąż można dodawać inne konstruktory.
Jednak wówczas wywoła się domyślnie konstruktor `()` klasy bazowej.

[Link]

```
4  struct A {
5    A ()      { cout << "A() "           ; }
6    A (int x) { cout << "A(" << x << ") "; }
7  };
8
9  struct B : A {
10   using A::A ;
11   B (int y, int z) { cout << "B(" << y <<
12                                 "," << z << ") "; }
13 };
14
15 int main () {
16   B b0       ;
17   B b1 (1)   ;
18   B b2 (2, 3);
19 }
```

Efekt: `A() A(1) A() B(2,3)`

- Shadowing pola klasy bazowej.
  Niezalecane, ale można.
  Przesłonięte pole klasy bazowej adresujemy tak:
  `ObiektPochodny.KlasaBazowa::Pole;`

[Link]

```
4  struct A     { float x;  };
5  struct B : A { float x;  };
6
7  int main () {
8    B b ;   b.x = 1. ;    b.A::x = 2. ;
9    cout << b.x << ' ' << b.A::x << endl;  }
```

- **Polimorfizm i metody wirtualne**.

  Polimorfizm (wielopostaciowość) – to możliwość, aby referencja na klasę bazową przyjęła obiekt dowolnej z klas pochodnych, ale w ten sposób, aby wywołanie metody na tej referencji wykonało wariant metody adekwatny dla klasy przypisanego obiektu.

  Aby mechanizm zadziałał, w klasie bazowej nagłówek metody musi mieć przedrostek **`virtual`**. Mówimy, że metoda ta jest **wirtualna**.

⊙ W tym przykładzie, w klasie `Figura` jest wirtualna metoda `Print`. Jest ona przeciążona w klasach pochodnych.

  Funkcja `MyPrint` wystawia referencję o typie bazowym. W `main` funkcja ta jest wywołana za każdym razem z podaniem obiektu innego typu. Komputer wybiera ten wariant metody `Print` , który realnie dotyczy użytego obiektu.

[Link]

```
5  struct Figura {
6    float Pole;
7    virtual void Print () { cout << "F:Pole " << Pole << endl;  }
8  };
9
10 struct Okrag : public Figura {
11   float R;
12   Okrag (float _R) : R (_R) { Pole = M_PI * R * R; }
13   void Print () { cout << "O: P,R " << Pole <<' '<< R << endl;}
14 };
15
16 struct Kwadrat : public Figura {
17   float A;
18   Kwadrat (float _A) : A (_A) { Pole = A * A; }
19   void Print () { cout << "K: P,A " << Pole <<' '<< A << endl;}
20 };
21
22 void MyPrint ( Figura& fig ) {
23   fig.Print () ;
24 }
25
26 int main () {
27   Figura  F ; F.Pole = 1. ;   MyPrint ( F ) ;
28   Okrag   O ( 1. ) ;          MyPrint ( O ) ;
29   Kwadrat K ( 1. ) ;          MyPrint ( K ) ;
30 }
```

- *Uwaga*: to samo działa dla wskaźnika typu bazowego i przypisania mu adresu obiektu klasy pochodnej.

- **Czysta metoda wirtualna, klasa abstrakcyjna**.

  Czasem metoda w klasie bazowej
  nie ma sensu, ale wiemy, że zaistnieje ona
  w klasach pochodnych. W przykładzie,
  nie ma sensu liczyć pola figurze „ogólnej”,
  ale ma sens – każdej figurze konkretnej.

  Możemy w klasie bazowej „zapowiedzieć”
  metodę, którą skonkretyzują klasy pochodne:

  ```
  virtual typ MojaMetoda (typ) = 0;
  ```

  Nazywamy ją metodą „**czystą wirtualną**”.
  Klasę z taką metodą nazywamy **abstrakcyjną**.
  Nie można utworzyć obiektu takiej klasy.

  W funkcji `main` wywołana będzie zawsze
  metoda `Pole` właściwa typowi obiektu.

- ⊙ Czy jednak nie wystarcza po prostu nie pisać
  zapowiedzi `virtual … = 0` w klasie bazowej?

  → Jeśli chcemy skorzystać z dobrodziejstw
  polimorfizmu (w tym przykładzie nie widać) ,
  to zapowiedź jest konieczna.

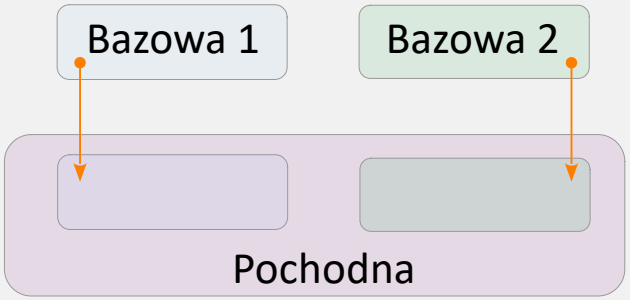
[Link]

```
5  struct Figura {
6    float Pole ;
7    virtual void ObliczPole () = 0 ;
8  };
9
10 struct Okrag : public Figura {
11   float R;
12   Okrag   (float _R) : R (_R) { }
13   void ObliczPole()  { Pole = M_PI * R * R; }
14 };
15
16 struct Kwadrat : public Figura {
17   float A;
18   Kwadrat (float _A) : A (_A) { }
19   void ObliczPole ()  { Pole = A * A;  }
20 };
21
22 int main () {
23   Okrag   O ( 1. ) ;
24   Kwadrat K ( 1. ) ;
25   Figura& f1 = O; f1.ObliczPole ();
26                   cout << f1.Pole << endl;
27   Figura& f2 = K; f2.ObliczPole ();
28                   cout << f2.Pole << endl;
29 }
```

Dziedziczenie wielokrotne

[Link]

Klasa może dziedziczyć cechy kilku klas na raz.



Wymieniamy je w nagłówku po przecinku.
Reguły dziedziczenia są jak wcześniej.

- W przykładzie klasa `Osoba` dziedziczy po `Miejsce` i `Ozywione`.
  W tym przypadku, publicznie dziedziczy wszystkie pola tych klas.

  Można wyobrazić sobie klasę `Rower`, która też dziedziczyłaby po `Miejsce` i np. po klasie `Pojazd`, ale nie po `Ozywione`.

```
5  struct Miejsce  {
6   float X, Y ;
7   Miejsce (float x, float y) : X(x) , Y(y) {}
8  };
9
10 struct Ozywione {
11  float tetno;
12  Ozywione (float t) : tetno (t) {}
13 };
14
15 struct Osoba : Miejsce, Ozywione {
16  string Nazwa;
17  Osoba (float _x, float _y, float t, string n)
18        : Miejsce (_x, _y), Ozywione (t), Nazwa (n) {}
19 };
20
21 int main () {
22  Osoba Adas (12.3 , 45.6, 70. , "Adas Miauczynski");
23  cout << Adas.Nazwa << " jest w miejscu: ["
24             << Adas.X << " , " << Adas.Y << "] i ma tetno "
25             << Adas.tetno << ".\n";
26 }
```