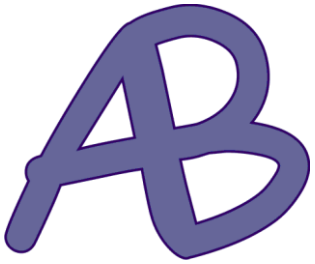
Wykład: Dziedziczenie



Dziedziczenie

Dziedziczenie

Dziedziczenie - jest jednym z fundamentalnych mechanizmów programowania obiektowego w C++. Pozwala na tworzenie nowych klas na podstawie istniejących.

- Klasę dziedziczącą nazywamy klasą pochodną (subklasa)
- klasę, po której klasa pochodna dziedziczy, nazywamy klasą bazową (superklasa).

```
class Bazowa
{ };

class Pochodna : typ_dziedziczenia Bazowa
{ };
```

Dziedziczenie

W klasie pochodnej możemy:

- zdefiniować dodatkowe dane składowe,
- zdefiniować dodatkowe funkcje składowe, zdefiniować składnik (najczęściej funkcję składową), który istnieje już w klasie podstawowej.

Definiowanie nowych funkcji składowych - bez definiowania dodatkowych danych składowych także ma sens. jest to jakby wyposażenie klasy w nowe zachowania;

Dziedziczenie

Klasa nie dziedziczy:

- ✓ Konstruktorów,
- ✓ Destruktora,
- ✓ Operatora przypisania

Dziedziczenie

Typy dziedziczenia (modyfiktory dostępu):

1. Dziedziczenie publiczne - najczęściej stosowane. Zachowuje dostępność składowych (publiczne pozostają publiczne, chronione pozostają chronione).

2. Dziedziczenie chronione - publiczne i chronione elementy klasy bazowej stają się chronione w klasie pochodnej.

3. Dziedziczenie prywatne - jest domyślne (gdy nie jest podany typ dziedziczenia). Publiczne i chronione elementy klasy bazowej stają się prywatne w klasie pochodnej.

Dziedziczenie

Typy dziedziczenia:

```
/*
                                +-----------------------------------+
                                |        sposób dziedziczenia       |
                                +-----------+-----------+-----------+
                                |  public   | protected |  private  |
+-------------+-----------+=====+===========+===========+===========+
|             |  public   #  public   | protected |  private  |
| widoczność  +-----------+-----------+-----------+-----------+
|  w klasie   | protected # protected | protected |  private  |
|  bazowej    +-----------+-----------+-----------+-----------+
|             |  private  #     -*     |     -*    |     -*    |
+-------------+-----------+-----------+-----------+-----------+

* - niedostępne, jeśli nie ma przyjaźni
```

Dziedziczenie a zasięg widoczności nazw

Przesłanianie dziedziczonych metod

Dziedziczenie

Dziedziczenie a zasięg widoczności nazw (przesłanianie składowych)

- ✓ Jeśli składowa klasy pochodnej ma taką samą nazwę jak składowa klasy bazowej - składowa klasy bazowej zostanie przesłonięta.

- ✓ W klasie bazowej można użyć przesłoniętej składowej klasy bazowej - za pomocą operatora zasięgu **::**

```
Klasa_bazowa::pole;
```

- ✓ Przesłaniać możemy zarówno pola, jak i metody.

Dziedziczenie

Uwaga: Przeciążanie metod działa tylko w obrębie klasy.

- ✓ Jeśli metody wewnątrz jednej klasy mają taką samą nazwę, lecz różną listę parametrów mamy do czynienia z przeciążaniem metod klasy.

- ✓ Jednak mechanizm nie działa w przypadku dziedziczenia – metoda klasy bazowej jest przesłaniana przez metodę klasy pochodnej o takiej samej nazwie.

Dziedziczenie

Dziedziczenie, a zasięg widoczności nazw (przesłanianie składowych)

```
class bazowa
{
public:
    int x=10;
};

class pochodna : public bazowa
{
    int x=1000;
public:
    void wypisz()
    {
        cout <<"x z klasy pochodnej = "<<x<<endl;;
        cout <<"x z klasy bazowej = "<<bazowa::x<<endl;
    }
};

int main()
{
    pochodna k1;
    k1.wypisz();
    return 0;
}
```

```
x z klasy pochodnej = 1000
x z klasy bazowej = 10
```

Inicjalizacja odziedziczonych składowych

Konstruktor klasy pochodnej

Dziedziczenie

Inicjalizacja odziedziczonych składowych

Aby zainicjalizować odziedziczoną część obiektu, wywołujemy w konstruktorze klasy pochodnej konstruktor klasy bazowej

```
class bazowa
{
private:
  int x;
public:
  bazowa(int parametr) : x(parametr) {} //konstruktor z parametrem
};

class pochodna : public bazowa
{
public:
  pochodna(int parametr) : bazowa(parametr) {}
  // konstruktor klasy pochodnej
  // uruchamia konstruktor klasy bazowej (przekazując do niego parametr),
  // a ten przekazuje parametr dalej, do prywatnego pola klasy bazowej
};
```

Dziedziczenie

Konstruktory klasy pochodnej

- jeżeli istnieje więcej niż jeden konstruktor

```
class bazowa
{
public:
  bazowa() {}                 //konstruktor domyślny
  bazowa(int parametr) {} //konstruktor z parametrem
};

class pochodna : public bazowa
{
public:
  pochodna() : bazowa() {}  //konstruktor klasy pochodnej
                            // uruchamia konstruktor klasy bazowej

  pochodna (int parametr) :bazowa(parametr) {}
};
```

Dziedziczenie

Jeśli na liście inicjalizacyjnej nie umieściliśmy wywołania konstruktora klasy podstawowej, a klasa podstawowa w zestawie swoich konstruktorów nie ma konstruktora domniemanego - to wówczas kompilator uzna to za błąd.

Na liście inicjalizacyjnej można pominąć wywołanie konstruktora bezpośredniej klasy podstawowej tylko wtedy, gdy:

- klasa podstawowa nie ma żadnego konstruktora.
- ma konstruktory, a wśród nich jest konstruktor domniemany.

Dziedziczenie - przykład

```
class punkt
{
private:
     int x;
     int y;
public:
     punkt(int px=0, int py=0)
     {
          x=px;
          y=py;
     }
     void wypisz ()
     {
          cout <<"x = "<<x<<endl;
          cout <<"y = "<<y<<endl;
     }
};
```

Dziedziczenie - przykład

```
class okrag : public punkt
{
private:
    int r;
public:
    okrag(int px=0, int py=0,int pr=0):punkt(px,py)
    {
        r=pr;
    }
    void wypisz() //funkcja przesłania funkję wypisz()
                  //z klasy bazowej
    {
        punkt::wypisz(); //siegamy po przesłoniętą funkcje
                         //klasy bazowej, gdyż inaczej nie dostaniemy się
                         //do jej pól prywatnych
        cout<<"r = "<<r<<endl;
    }
};
```

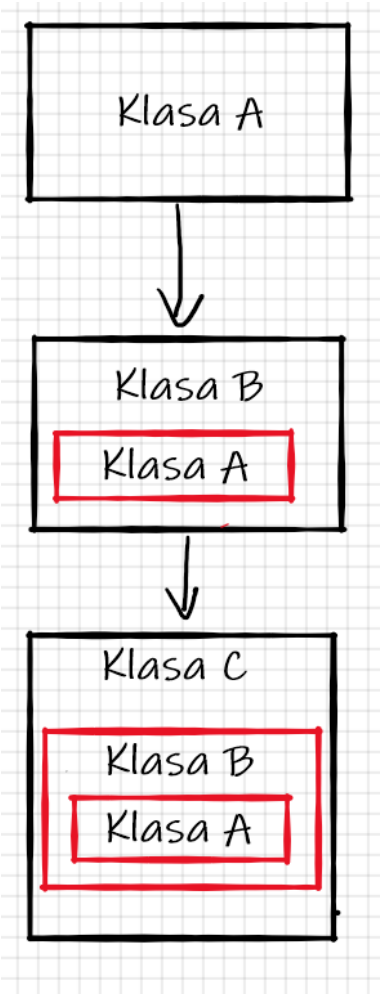
Dziedziczenie

Dziedziczenie wielopokoleniowe

```
class A
{
  // – ciało klasy A
};

class B : public A
{
  // – ciało klasy B
};

class C : public B
{
  // – ciało klasy C
};
```



Dziedziczenie

Dziedziczenie wielopokoleniowe - PRZYKŁAD

Klasa „punkt” jest bazą do dziedziczenia. Przechowuje współrzędne punktu na płaszczyźnie i posiada podstawowe metody.

```
#include <iostream>
#include <sstream>

using namespace std;

class punkt
{
protected:
    int x;
    int y;

public:
    punkt(int X, int Y) : x(X), y(Y)
    {
        cout << endl
             << "LOG: konstruktor klasy punkt, z parametrami " << endl;
    }
    void setX(int x) { this->x = x; }
    void setY(int y) { this->y = y; }
    int getX() { return x; }
    int getY() { return y; }
    string toString()
    {
        stringstream temp;
        temp << "(" << x << ";" << y << ")";
        return temp.str();
    }
};
```

Dziedziczenie

```
class punkt_3D : public punkt
{
protected:
    int z;

public:
    punkt_3D(int X, int Y, int Z) : punkt(X, Y), z(Z)
    {
        cout << endl
             << "LOG: konstruktor klasy punkt_3D, z parametrami " << endl;
    }
    void setZ(int z) { this->z = z; }
    int getZ() { return z; }
    string toString()
    {
        stringstream temp;
        temp << "(" << x << ";" << y << ";" << z << ")";
        return temp.str();
    }
};
```

Klasa „punkt_3D” dziedziczy publicznie po klasie „punkt”.

Posiada wszystkie składowe klasy „punkt”.

Dodatkowo uzupełniona jest o:

- trzecią współrzędną „z” (punkt w przestrzeni 3D),
- metody pozwalające na odczyt/modyfikację dodanej współrzędnej,
- nową wersję metody „toString()”

Dziedziczenie

```
class pixel_3D : public punkt_3D
{
protected:
    string kolor;

public:
    pixel_3D(int X, int Y, int Z, string K) : punkt_3D(X, Y, X), kolor(K)
    {
        cout << endl
             << "LOG: konstruktor klasy pixel_3D, z parametrami " << endl;
    }
    void setKolor(string K) { kolor = K; }
    string getKolor() { return kolor; }
    string toString()
    {
        stringstream temp;
        temp << punkt_3D::toString() << "-" << kolor << " ";
        return temp.str();
    }
};

int main()
{
    pixel_3D p1(10, 20, 30, "red");
    cout << p1.toString();
    return 0;
}
```

Klasa „pixel_3D” dziedziczy publicznie po klasie „punkt_3D”, a pośrednio także po klasie „punkt”

Posiada wszystkie składowe klas „punkt” i „punkt_3D”

Dodatkowo uzupełniona jest o:

- Kolor (dla czytelności w postaci nazwy koloru),
- metody pozwalające na odczyt/modyfikację koloru
- nową wersję metody „toString()” Zauważmy, że wywołuje ona jawnie metodę klasy rodzica `punkt_3D::toString()`

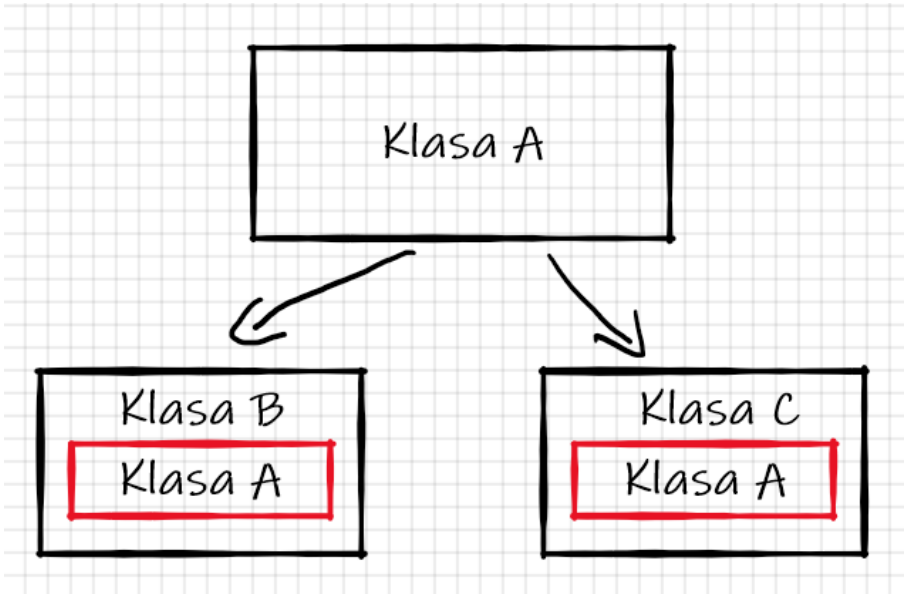
Dziedziczenie

Dziedziczenie równorzędne

```
class A
{
  // – ciało klasy A
};

class B : public A
{
  // – ciało klasy B
};

class C : public A
{
  // – ciało klasy C
};
```



Dziedziczenie

Dziedziczenie równorzędne - PRZYKŁAD

```
class Figura
{
protected:
    int x;
    int y;

public:
    Figura(int X, int Y) : x(X), y(Y) {}
    string toString()
    {
        stringstream s;
        s << "(" << x << ";" << y << ")";
        return s.str();
    }
};
```

Klasa „Figura” jest bazą do dziedziczenia przez trzy klasy „Prostokat” „Kolo” i „Trojkat”.

Współrzędne x, y oznaczają położenie figury – jej lewego górnego rogu (punktu wiodąceg) na układzie współrzędnych (np. na ekranie)

Dziedziczenie

```
class Prostokat : public Figura
{
protected:
    int w;
    int h;

public:
    Prostokat(int X, int Y, int W, int H) : Figura(X, Y), w(W), h(H) {}
    int getPole()
    {
        return w * h;
    }
    int getObwod()
    {
        return 2 * w + 2 * h;
    }
    string toString()
    {
        stringstream s;
        s << "Pozycja:" << x << ";" << y
          << " PP=" << getPole()
          << " Obw=" << getObwod();
        return s.str();
    }
};
```

Klasa „Prostokat” dziedziczy publicznie po klasie „Figura”.

Dziedziczenie

```
class Kolo : public Figura
{
protected:
    int d;

public:
    Kolo(int X, int Y, int D) : Figura(X, Y), d(D) {}
    double getPole()
    {
        return 3.14 * (0.5 * d) * (0.5 * d);
    }
    double getObwod()
    {
        return 3.14 * d;
    }
    string toString()
    {
        stringstream s;
        s << "Pozycja:" << x << ";" << y
          << " PP=" << getPole()
          << " Obw=" << getObwod();
        return s.str();
    }
};
```

Klasa „Kolo” dziedziczy publicznie po klasie „Figura”.

Dziedziczenie

```
class Trojkat : public Figura
{
protected:
    int w;
    int h;

public:
    Trojkat(int X, int Y, int W, int H) : Figura(X, Y), w(W), h(H) {}
    double getPole()
    {
        return 0.5 * w * h;
    }
    double getObwod()
    {
        return sqrt((0.5 * w) * (0.5 * w) + h * h);
    }
    string toString()
    {
        stringstream s;
        s << "Pozycja:" << x << ";" << y
          << " PP=" << getPole()
          << " Obw=" << getObwod();
        return s.str();
    }
};
```

Klasa „Trojkat” dziedziczy publicznie po klasie „Figura”.

Literatura:

W prezentacji wykorzystano przykłady i fragmenty:

- Gręboszl J. : ***Symfonia C++, Programowanie w języku C++ orientowane obiektowo***, Wydawnictwo Edition 2000.
- Jakubczyk K.: *Turbo Pascal i Borland C++ Przykłady*, Helion.

Warto zajrzeć także do:

- Sokół R. : ***Microsoft Visual Studio 2012 Programowanie w Ci C++,*** Helion.
- Kerninghan B. W., Ritchie D. M.: ***język ANSI C***, Wydawnictwo Naukowo Techniczne.

Dla bardziej zaawansowanych:

- Gręboszl J. : ***Pasja C++,*** Wydawnictwo Edition 2000.
- Meyers S.: ***język C++ bardziej efektywnie***, Wydawnictwo Naukowo Techniczne