Kurs języka C++

5. Dziedziczenie

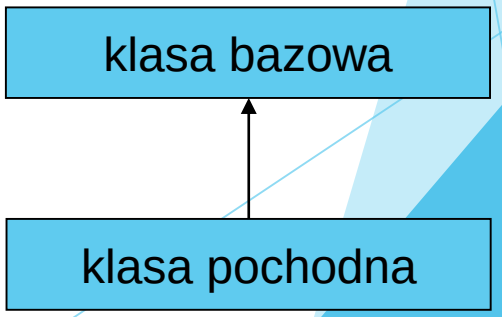
Spis treści

Istota dziedziczenia

- Kompozycja
- Dziedziczenie pozwala stworzyć nową klasę przy wykorzystaniu już istniejącej klasy.
- Dziedziczenie to modyfikacja typu polegająca na jego przystosowaniu do określonych warunków – jest to więc rodzaj specjalizacji.
- W klasie pochodnej można:
  - zdefiniować dodatkowe pola składowe,
  - zdefiniować dodatkowe funkcje składowe,
  - nadpisać funkcję składową (można też zasłonić pole składowe).

Nazewnictwo i oznaczenia

- Nomenklatura:
  - klasa bazowa (podstawowa albo nadklasa) to klasa, z której dziedziczą inne klasy;
  - klasa pochodna (podklasa) to nowa klasa, która dziedziczy strukturę informacyjną i funkcjonalność z innej klasy.
- Rysunek schematyczny:



Lista pochodzenia

- Przykład:

```
class punkt2D
{
protected:
    double x, y;
public:
    string opis () const;
// …
};
class punkt3D : public punkt2D
{
protected:
    double z;
public:
    double odleglosc (const punkt3D &p) const;
    string opis () const;
// …
};
```

- Lista pochodzenia jest umieszczona w nagłówku klasy po dwukropku.

Dostęp do składników

- Składniki z klasy bazowej stają się składnikami w klasie pochodnej (wszystko jest dziedziczone).
- Jeśli w klasie pochodnej jest składnik o takiej samej nazwie jak składnik w klasie bazowej, to w zakresie klasy pochodnej składnik z tej klasy zasłania składnik odziedziczony.
- Do zasłoniętych składników z klasy bazowej można się odwoływać kwalifikując ich nazwy nazwą klasy bazowej. Przykład:

```
punkt3D p(5.7,2.3,-0.1);
// …
cout << p.punkt2D::opis() << endl;
```

Dostęp do składników

- W klasie pochodnej nie ma dostępu do odziedziczonych składników prywatnych (czyli `private`).
- W klasie pochodnej jest dostęp do odziedziczonych składników nieprywatnych (czyli `protected` i `public`).
- Składniki chronione (czyli `protected`) są dostępne tylko w klasie bieżącej i w klasach pochodnych ale nie na zewnątrz.

Dostęp do składników

- Klasa pochodna też decyduje o zakresie widoczności odziedziczonych składników nieprywatnych poprzez sposób dziedziczenia (`public`, `protected`, `private`):
  - przy dziedziczeniu publicznym odziedziczone składniki nieprywatne zachowują swój zakres widoczności;
  - przy dziedziczeniu chronionym odziedziczone składniki nieprywatne stają się chronione;
  - przy dziedziczeniu prywatnym odziedziczone składniki nieprywatne stają się prywatne.
- Domyślny sposób dziedziczenia to `private`.

Dostęp do składników

- Za pomocą deklaracji dostępu `using` można wybiórczo przywrócić pierwotny zakres widoczności składnika przy dziedziczeniu niepublicznym.
- Przykład:

```
class potomek: private przodek
{
    // …
protected:
    using przodek::polechr;
    using przodek::funchr;
public:
    using przodek::polepub;
    using przodek::funpub;
};
```

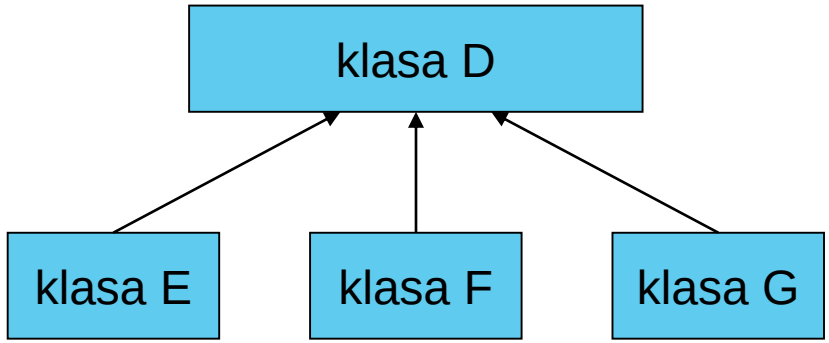
- W deklaracji dostępu używamy tylko nazw składników z klasy bazowej poprzedzonej kwalifikatorem zakresu dla tej klasy.

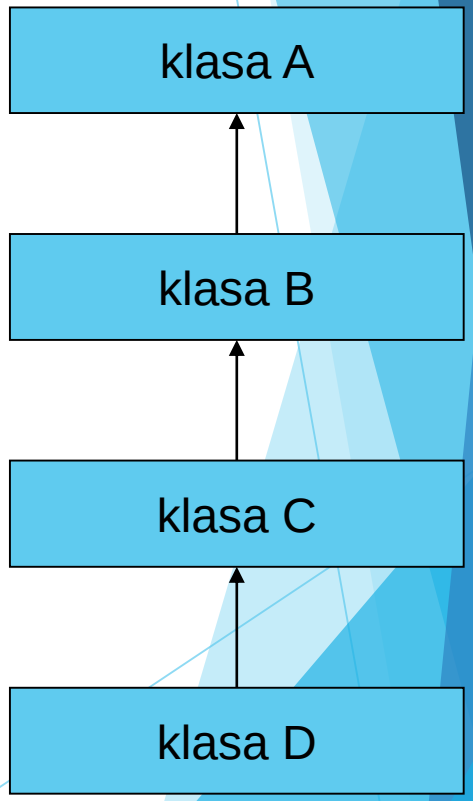
Czego się nie dziedziczy?

- Dziedziczy się wszystko, ale pewne składniki nie będą dostępne w klasie pochodnej:
  - „nie dziedziczy się” konstruktorów (konstruktor w klasie bazowej nie staje się konstruktorem w klasie pochodnej);
  - „nie dziedziczy się” operatora przypisania kopiującego (jeśli nie zdefiniujemy operatora przypisania kopiującego w klasie pochodnej to kompilator sam go wygeneruje jeśli będzie to możliwe);
  - „nie dziedziczy się” destruktora.

Hierarchia klas

- Dziedziczenie może mieć wiele poziomów.
- Jedna klasa może być klasą bazową dla wielu innych klas.





Konstrukcja i destrukcja obiektu w warunkach dziedziczenia

- Gdy tworzymy nowy obiekt klasy pochodnej to:
  - najpierw zostanie wywołany konstruktor klasy bazowej,
  - potem konstruktory obiektów składowych (w kolejności ich deklaracji w klasie),
  - a na końcu ruszy do pracy konstruktor klasy pochodnej.
- Jawne wywołanie konstruktora klasy bazowej może się pojawić tylko na liście inicjalizacyjnej konstruktora klasy pochodnej (inaczej zostanie wywołany konstruktor domyślny). Przykład:

```
potomek::potomek () : przodek()
   {/*…*/}
```

- Destruktory będą wywoływane w odwrotnej kolejności w stosunku do konstruktorów.

Konstrukcja i destrukcja obiektu w warunkach dziedziczenia

- Konstruktor delegatowy wywołuje inny konstruktor z tej samej klasy (zamiast konstruktora klasy bazowej). Przykład:

```
potomek::potomek () : potomek(param)
   {/*…*/}
```

- W konstruktorze delegatowym operujemy na zainicjalizowanym obiekcie.

Inicjalizacja przez kopiowanie w warunkach dziedziczenia

- Gdy klasa pochodna nie definiuje swojego konstruktora kopiującego, to konstruktor taki zostanie wygenerowany automatycznie przez kompilator (o ile klasy bazowe i obiekty składowe udostępniają swoje konstruktory kopiujące).
- Automatycznie wygenerowany konstruktor kopiujący działa na zasadzie konstruowania kopiującego części odziedziczonej i kopiowania kolejnych składników.

Przypisanie kopiujące w warunkach dziedziczenia

- Gdy klasa pochodna nie definiuje swojego operatora przypisania kopiującego, to przypisanie takie zostanie wygenerowane automatycznie przez kompilator (o ile klasa nie ma składowych stałych lub referencyjnych oraz wszystkie obiekty składowe można kopiować operatorem przypisania).
- Kopiowanie automatyczne działa na zasadzie kopiowania kolejnych składników.

Inicjalizacja i przypisanie według stałego obiektu wzorcowego w warunkach dziedziczenia

- Kompilator automatycznie wygeneruje dla klasy `K` konstruktor kopiujący
  `K::K (const K &x)`
  i przypisanie kopiujące
  `K& K::operator = (const K &x)`
  gdy wszystkie klasy bazowe oraz wszystkie klasy pól składowych posiadają analogiczne konstruktory kopiujące i przypisania kopiujące.

Co generuje kompilator C++

- Kompilator C++ dołącza do obiektów konstruktor domyślny, konstruktor kopiujący i przypisanie kopiujące, ~~konstruktor przenoszący i przypisanie przenoszące,~~ gdy użytkownik nie zdefiniuje swoich własnych wersji tych metod.
- Kompilator C++ definiuje także kilka globalnych operatorów (takich jak operator `new` czy operator `delete`), które pracują ze wszystkimi klasami i które użytkownik także może zastąpić swoimi wersjami.
- Stworzenie klasy nieinstancyjnej wymaga deklaracji tylko prywatnych konstruktorów lub niedefiniowania żadnego.
- Stworzenie klasy, po której nie można dziedziczyć wymaga dopisania w nagłówku klasy słowa `final`, na przykład:

```
class P final : public B { … };
```

Ustawianie metod jako `default`

- Deklaracja `default` wymusza na kompilatorze wygenerowanie domyślnej metody (konstruktora domyślnego).
- Aby kompilator C++ wygenerował konstruktor domyślny pomimo istnienia w klasie deklaracji innych konstruktorów należy użyć specyfikatora **`=default`**:

```
Klasa () = default;
```

- Przykład klasy z konstruktorem domyślnym wygenerowanym przez kompilator:

```
struct SomeType {
  // domyślny konstruktor
  // jest jawnie określony
  SomeType () = default;
  SomeType (OtherType value);
  // …
};
```

Ustawianie metod jako `delete`

- Deklaracja `delete` blokuje w kompilatorze mechanizm generowania domyślnych metod (konstruktora kopiującego, przypisania kopiującego, konstruktora domyślnego).
- Aby kompilator C++ nie wygenerował automatycznie konstruktora kopiującego czy przypisania kopiującego należy użyć specyfikatora **`=delete`** :

```
Klasa (const Klasa&) = delete;
Klasa& opeartor= (const Klasa&) = delete;
```

- Przykład klasy, której obiekty będą niekopiowalne:

```
struct NonCopyable {
  // konstruktor kopiujący i przypisanie kopiujące
  // nie zostaną wygenerowane
  NonCopyable& operator= (const NonCopyable&) = delete;
  NonCopyable (const NonCopyable&) = delete;
  NonCopyable () = default;
  // …
};
```

Ustawianie metod jako `delete`

- Przykład klasy, której obiektów nie będzie można utworzyć za pomocą operatora `new`:

```
struct NonNewable {
  void* operator new (std::size_t) = delete;
  // …
};
```

- Specyfikator `=delete` może być użyty do zablokowania wywołania dowolnej metody, co może być użyte do zablokowania wywołania metody z określonymi parametrami.
- Przykład zakazania wywołania metody `f()` z argumentem typu `int` (domyślnie kompilator dokonałby niejawnej konwersji do typu `double`):

```
struct NoDouble {
  void f (double d);
  void f (int) = delete;
  // …
};
```

- Uogólnienie powyższego przykładu (zakazanie wywołania metody `f()` z argumentem dowolnego typu różnym od `int`):

```
struct OnlyInt {
  void f (int i);
  template<typename T>
    void f (T) = delete;
  // …
};
```

Dziedziczenie wielokrotne (wielodziedziczenie)

- Wielodziedziczenie ma miejsce wtedy, gdy klasa ma kilka klas bazowych.
- Za pomocą wielodziedziczenia można ze sobą łączyć różne typy danych.
- Na liście pochodzenia znajdują się różne klasy i przy każdej z nich jest określony indywidualny sposób dziedziczenia (`public`, `protected`, `private`).
- Wszystkie klasy na liście pochodzenia muszą być znane kompilatorowi (nie wystarczy sama deklaracja zapowiadająca).

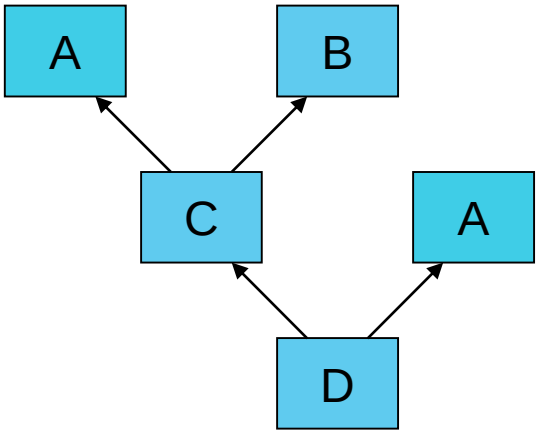
Dziedziczenie wielokrotne (wielodziedziczenie)

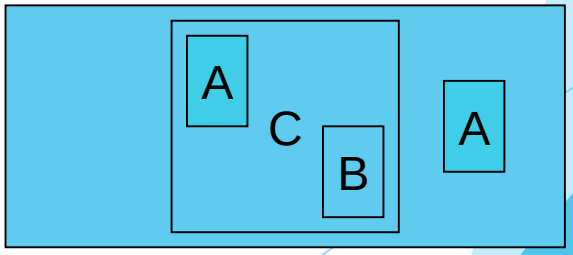
- Konstruktory klas bazowych będą wywoływane w kolejności ich występowania na liście pochodzenia.
- Destruktory klas bazowych będą wywoływane w kolejności odwrotnej niż konstruktory.

- Istnieje ryzyko wieloznaczności nazw przy dziedziczeniu wielokrotnym.
- Przy rozstrzyganiu wieloznaczności posługiwanie się operatorem zakresu jest możliwe ale ryzykowne w stosunku do funkcji wirtualnych.
- Bliższe pokrewieństwo nie usuwa wieloznaczności i poszlaki nie są uwzględniane.

Dziedziczenie wielokrotne (wielodziedziczenie)

- Wielodziedziczenie może prowadzić do wielu skomplikowanych sytuacji: w pojedynczym obiekcie pewna informacja może się wielokrotnie powtórzyć.





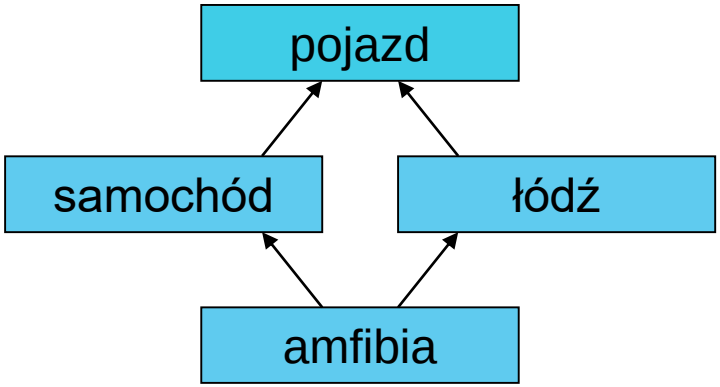
Dziedziczenie wirtualne

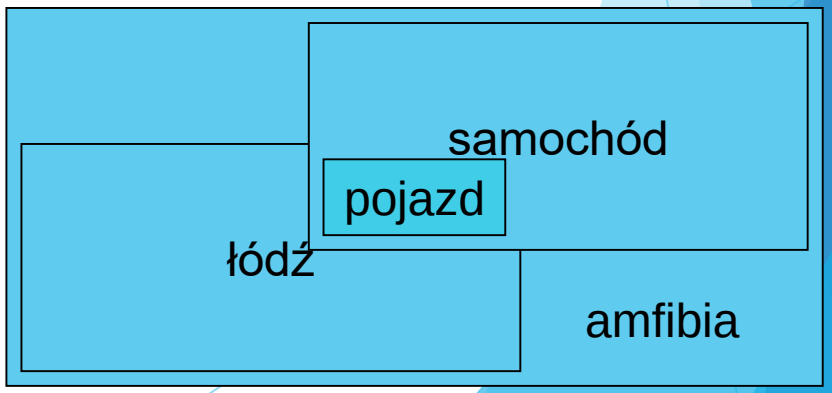
- Dziedziczenie wirtualne może rozwiązać część problemów z dziedziczeniem wielobazowym.
- Dziedziczenie wirtualne powoduje, że pewne informacje występujące wielokrotnie w obiekcie mogą stać się wspólne dla wielu części.
- Dziedziczenie wirtualne deklaruje się słowem `virtual` występującym na liście pochodzenia przed klasą bazową.
- Konstruktor wirtualnej klasy podstawowej jest wywoływany przed konstruktorami jej klas pochodnych.

Dziedziczenie wirtualne

- Przykład dziedziczenia wirtualnego:

```
class pojazd
    {/*…*/};
class samochód: public virtual pojazd
    {/*…*/};
class łódź: public virtual pojazd
    {/*…*/};
class amfibia: public samochód, public łódź
    {/*…*/};
```





Konwersje standardowe przy dziedziczeniu

- Wskaźnik do obiektu klasy pochodnej może być niejawnie przekształcony na wskaźnik dostępnej jednoznacznie klasy bazowej (czyli wskaźnikiem do klasy bazowej możemy pokazywać na obiekty klas pochodnych).
- Referencja do obiektu klasy pochodnej może być niejawnie przekształcona na referencję dostępnej jednoznacznie klasy bazowej (czyli referencja do klasy bazowej może się odnosić do obiektu klasy pochodnej).
- Sformułowanie „dostępny jednoznacznie” w kontekście hierarchii klas oznacza dziedziczenie publiczne tylko po jednej klasie.

Konwersje standardowe przy dziedziczeniu

- Konwersje standardowe wskaźnika (albo referencji) do obiektu klasy pochodnej na wskaźnik (albo referencję) do obiektu klasy bazowej mogą zajść:
  - przy przesyłaniu argumentów do funkcji,
  - przy zwracaniu przez funkcję rezultatu,
  - przy przeładowanych operatorach,
  - w wyrażeniach inicjalizujących.
- Konwersja standardowa wskaźnika w przypadku dziedziczenia sprawdza się dobrze z pojedynczymi obiektami – konwersji tej nie wolno stosować w przypadku tablic.

Przenoszenie konstruktorów z klasy bazowej

- Kompilator C++11 pozwala się na odziedziczenie konstruktorów po klasie bazowej.
- Kompilator C++11 wygeneruje kod przenoszący do klasy pochodnej wszystkie konstruktory z klasy bazowej – jest to operacja typu wszystko albo nic: albo wszystkie konstruktory klasy bazowej są przenoszone albo żaden.
- Istnieją ograniczenia na dziedziczenie konstruktorów:
  - przy wielokrotnym dziedziczeniu konstruktory klas nie mogą być dziedziczone z dwóch klas używających konstruktorów o tej samej sygnaturze,
  - oraz nie mogą istnieć konstruktory w klasie bazowej i pochodnej o tej samej sygnaturze.

Przenoszenie konstruktorów z klasy bazowej

- Przykład:

```
class BaseClass {
public:
  BaseClass(int iValue);
// …
};

class DerivedClass : public BaseClass
{
public:
  using BaseClass::BaseClass;
// …
};
```

Inicjalizacja pól w definicji klasy

- Kompilator C++11 dopuszcza inicjalizację pól na etapie definicji klasy – jeśli konstruktor nie nadpisze tego pola własną wartością, to pozostanie tam wartość użyta w inicjalizatorze.
- Przykład (konstruktor klasy zainicjuje pole `value` określoną wartością, jeśli konstruktor nie nadpisze tego pola własną wartością):

```
class SomeClass {
protected:
  OtherClass value = OtherClass(1,2.3,”cztery”);
public:
  SomeClass () {}
  explicit SomeClass (const OtherClass &newValue)
  : value(newValue) {}
  // …
};
```

Pospolite stare struktury danych

- Klasa/struktura musi spełniać kilka wymagań, by stać się pospolitą strukturą danych POD (ang. *Plain Old Data*) – typ jest kompatybilny z typami używanymi w języku ANSI C, to znaczy może być wymieniany bezpośrednio z bibliotekami ANSI C w postaci binarnej (kompatybilne z typami w ANSI C).
- Typy podstawowe w języku C++ zalicza się do POD.
- Klasa/struktura jest uważana za POD, jeśli:
  - konstruktor domyślny jest trywialna (nic nie robi);
  - destruktor o ile istnieje jest trywialny (nic nie robi);
  - obiekty takich klas/struktur nie wymagają specjalnych technik kopiowania/przenoszenia (są trywialne).

Pospolite stare struktury danych

- Klasa/struktura jest POD, gdy:
  - posiada ten sam poziom dostępu (`private`, `protected`, `public`) dla wszystkich niestatycznych składowych;
  - nie posiada wirtualnych metod;
  - nie posiada wirtualnych klas bazowych;
  - może być tylko jedna klasa z niestatycznymi składowymi w całej hierarchii klas.

Inicjalizacja listą wartości

- Struktura POD lub tablica mogą być inicjalizowane poprzez listę argumentów o kolejności zgodnej, odpowiednio, z kolejnością definicji składowych struktury lub kolejnymi elementami tablicy.
- Inicjujące listy wartości są rekursywne i mogą być zastosowane także do tablicy struktur albo struktury zawierającej inną strukturę.
- C++11 wiąże koncepcję inicjowania za pomocą list wartości z typem `std::initializer_list` - to pozwoli konstruktorowi lub metodom na podanie takich list jako argumentów.

Inicjalizacja listą wartości

- Przykład:

```
class JakasKlasa {
public:
  JakasKlasa (std::initializer_list<int> list);
  // …
};
// …
JakasKlasa jakasZmienna = {1, 4, 5, 6};
```

- Taki konstruktor to konstruktor list inicjujących - klasy z takim konstruktorem są traktowane priorytetowo podczas jednolitego inicjalizowania.
- Listy inicjalizujące w C++11 mogą być początkowo inicjalizowane tylko statycznie przez kompilator przy użyciu składni `{}` - lista może być kopiowana raz przy konstrukcji i jest to tylko kopia przez referencję.
- Lista inicjalizująca jest stałą - ani jej składowe ani też dane w tych składowych nie mogą być zmienione po jej utworzeniu.

Jednolita inicjalizacja obiektów

- C++11 posiada składnię w pełni ujednolicającą inicjalizowanie dowolnych typów, która jest rozszerzeniem składni z listą wartości inicjalizujących.
- Przykład:

```
struct PodstStrukt {
  int x;
  float y;
};
struct AlternatStrukt {
  AlternatStrukt(float _y, int _x)
    : x(_x), y(_y) {}
private:
  int x;
  float y;
};
// …
PodstStrukt zm1 {5, 3.2f}; // przypisanie do
pól
AlternatStrukt zm2 {4.3f, 2}; // konstruktor
```