C++ - dziedziczenie

Kompozycję i dziedziczenie można łączyć, ale należy uważać z destruktorami przy usuwaniu obiektu.

Destruktorów nie wywołujemy jawnie ponieważ destruktor jest zawsze tylko jeden i jest bezargumentowy.

Destruktory obiektów będących składowymi i destruktor dziedziczony zostaną wywołane w ściśle określonej kolejności.

Przykład:

Przyjmijmy, ze mamy zadeklarowanych 5 klas:

`Bazowa`, `Skladowa1`, `Skladowa2`, `Skladowa3` i `Skladowa4`.

Wszystkie klasy mają zadeklarowany konstruktor jednoargumentowy.

C++ - dziedziczenie

```
class Pochodna1: public Bazowa {
   Skladowa1 s1;
   Skladowa2 s2;
   public:
   Pochodna1(int x): s2(4), s1(5), Bazowa(6) { … }
   ~Pochodna1() { … }
};
class Pochodna2: public Pochodna1 {
   Skladowa3 s3;
   Skladowa4 s4;
   public:
   Pochodna2(int x): s3(1), Pochodna1(2), s4(3) { … }
   ~Pochodna2() {… }
};
int main() {
   Pochodna2 p2(0);
};
```

Visual Studio

Program #7

C++ - dziedziczenie

Może zdarzyć się, że w naszej klasie pochodnej umieścimy metodę o nazwie takiej samej jak jedna z metod klasy bazowej.

Następuje wtedy:

1. przedefiniowanie (*redefining*), jeżeli są to zwykłe metody składowe
2. przesłanianie (*overriding*), gdy metoda klasy bazowej jest wirtualna

Jeżeli lista argumentów jest taka sama, to na tym się kończy.

Jeżeli lista argumentów jest różna, a na dodatek metoda jest przeciążona..

C++ - dziedziczenie

```
class Bazowa {
   public:
   int fun() {  return 0; }
   int fun(char *a) {  return 0; }
   …
};
class Pochodna1: public Bazowa {
   public:
   int fun(int) {  return 0; }
   …
};
class Pochodna2: public Bazowa {
   public:
   void fun() {  }
   …
};
```

```
class Pochodna3: public Bazowa {
   public:
   int fun() {  return 0; }
};

int main(int argc, char *argv[])
{
   Pochodna1 p1;
   int x = p1.fun(123);
   Pochodna2 p2;
   p2.fun();
   // W p1 i p2 brak dostępu
   // do dziedziczonych metod fun!
   Pochodna3 p3;
   x = p3.fun();
   // brak dostępu do p3.fun("Aqq");

   …
```


C++ - dziedziczenie

- Przedefiniowanie metody, której nazwa jest przeciążona w klasie bazowej, powoduje, że *wszystkie* pozostałe wersje tej metody przestają być dostępne.
- Użycie słowa **`virtual`**, tj. wykorzystanie metod wirtualnych powoduje też dalsze konsekwencje (*o czym będzie w dalszej części wykładu*).

C++ - dziedziczenie

Kompozycja i dziedziczenie – porównanie

Podobieństwa:

1. powodują utworzenie w nowej klasie obiektów podrzędnych
2. do skonstruowania obiektów podrzędnych wykorzystywana jest lista inicjalizatorów konstruktora

Różnice:

Kompozycja wprowadza do nowej klasy właściwości klasy, która już istnieje, ale nie jej interfejs
(jeżeli chcemy udostępnić użytkownikowi pola i metody będące własnością istniejącej klasy, to w nowej klasie korzystamy tylko odpowiednio ze zwykłych reguł dostępu)

Jeżeli mamy klasy 'silnik' i 'AM_DB9' to 'silnik' powinien być raczej składową 'AM_DB9'. Natomiast 'AM_DB9' powinien dziedziczyć po klasie 'samochód', bo Aston Martin nie zawiera w sobie samochodu, ale jest samochodem

C++ - dziedziczenie

Jeżeli przed nazwą dziedziczonej klasy nie napiszemy słowa **`public`** to mamy *dziedziczenie prywatne*

```
class A: B {
   …
};
class A: private B {
   …
};
```

Tworzymy nową klasę, posiadającą wszystkie składowe klasy bazowej ALE pozostają one ukryte – stanowią element wewnętrznej implementacji

Obiekt takiej klasy nie może być traktowany jako egzemplarz klasy bazowej np. przy rzutowaniu adresu obiektu między wskaźnikami

Po co taka konstrukcja, skoro można użyć kompozycji i dodać składową prywatną?
Dla porządku ☺.

185

C++ - dziedziczenie

Aby składowe odziedziczone prywatnie były widoczne publicznie, należy je wymienić z nazwy w publicznej części klasy pochodnej:

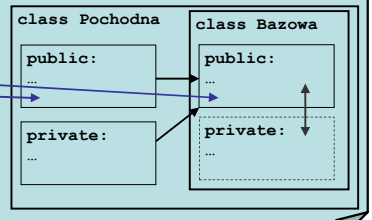
```
class Bazowa {
  public:
  int fun() { … }
  int fun(string) { … }
  …
};
class Pochodna: private Bazowa {
  public:
  Bazowa::fun;   // widoczne są obie przeciążone metody
  …
};
```


186

C++ - dziedziczenie

Dziedziczenie – prawa dostępu:

```
class Pochodna: public Bazowa;
…
int main() {
      Pochodna P;
      …
};
```

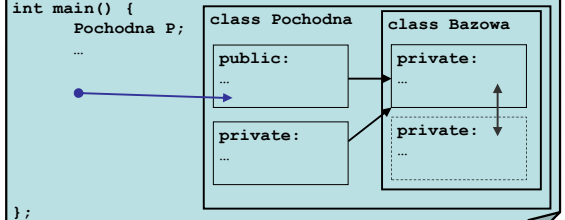


187

C++ - dziedziczenie

Dziedziczenie – prawa dostępu:

```
class Pochodna: private Bazowa;
…
int main() {
      Pochodna P;
      …
};
```



188

C++ - dziedziczenie

Jeżeli chcielibyśmy, żeby pewne składowe w klasie były chronione, czyli niedostępne zewnętrznym użytkownikom tej klasy, jednak jako dziedziczone składowe stały się dostępne metodom klasy dziedziczącej, pozostając jednak nadal niedostępne użytkownikom klasy dziedziczącej to…

w klasie bazowej deklarujemy je jako **`protected`**

Kiedy nie korzystamy z dziedziczenia, składowe zadeklarowane jako **`private`** i jako **`protected`** mają takie same prawa dostępu wewnątrz klasy i na zewnątrz. Różnica ujawnia się dopiero przy dziedziczeniu.

189

C++ - dziedziczenie

```
class Bazowa {
   int x;        // domyślnie private
protected:
   int y;
public:
   int z;
   …
};
class Pochodna: public Bazowa {
public:
   int fun(int a, int b) {
     // nie mam prawa modyfikować 'x'
        y = a;   // ale 'y' – tak (!)
        z = b;
   }
…
};
```

```
int main(int argc, char *argv[]) {
   Bazowa b;
   // nie mam prawa modyfikować x i y
   b.z = 0;

   Pochodna p;
   p.z = 0;
   // nie mam prawa modyfikować x i y
   ..
}
```


190

C++ - dziedziczenie

Sposoby dziedziczenia po klasie bazowej

public. Składniki typu `public` klasy bazowej stają się składnikami `public` klasy potomnej. Składniki typu `protected` klasy bazowej stają się składnikami `protected` klasy potomnej.

`public` → `public`

`protected` → `protected`

protected. Składniki typu `public` oraz `protected` klasy bazowej stają się składnikami `protected` klasy potomnej.

`public` → `protected`

`protected` → `protected`

private. Składniki typu `public` oraz `protected` klasy bazowej stają się składnikami `private` klasy potomnej.

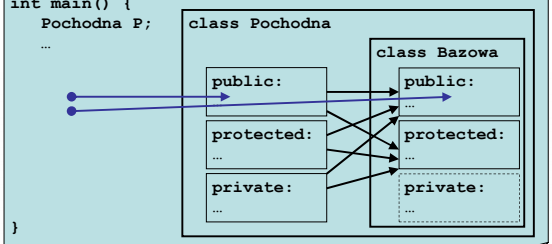
`public` → `private`

`protected` → `private`

Brak określenia sposobu dziedziczenia. Domyślnie wówczas przyjmowany jest typ `private`.

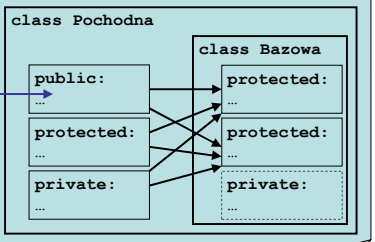
C++ - dziedziczenie

```
class Pochodna: public Bazowa;
…
int main() {
    Pochodna P;
    …
}
```



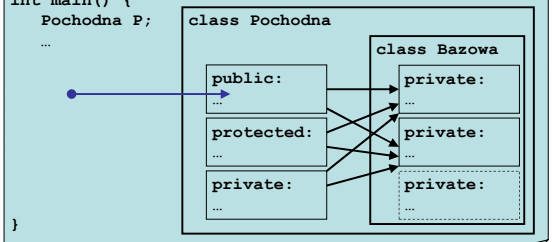
C++ - dziedziczenie

```
class Pochodna: protected Bazowa;
…
int main() {
    Pochodna P;
    …
}
```



C++ - dziedziczenie

```
class Pochodna: private Bazowa;
…
int main() {
    Pochodna P;
    …
}
```



C++ - dziedziczenie

Programowanie przyrostowe

Zaleta dziedziczenia i kompozycji jest programowanie przyrostowe: dodawanie nowego kodu bez edycji (*i ewentualnego wprowadzania błędów*) do kodu już istniejącego.

Dodając nową klasę dziedziczącą po innej, pozostawiamy istniejący kod w stanie nienaruszonym.

Przy założeniu poprawności działania klasy bazowej (*tj. realizującej prawidłowo swoje czynności i nie powodującej w trakcie działania efektów ubocznych*) ewentualny błąd – jeżeli się pojawi – może wystąpić tylko w nowym kodzie klasy dziedziczącej.

Projektowanie oprogramowanie jest procesem przyrostowym: zamiast pisać od razu cały program, lepiej jest pisać jego fragmenty i po ich wytestowaniu dopisywać następne – „hodować" program, tak aby wzrastał z czasem.

C++ - dziedziczenie

Rzutowanie w górę

Klasa dziedzicząca posiada wszystkie cechy klasy bazowej (plus swoje własne)

Tworzy się relacja między klasami:
nowa klasa jest typu tamtej, istniejącej już klasy

Jeżeli klasa bazowa ma jakąś metodę, to ma ją również klasa dziedzicząca, co oznacza, że każdy obiekt typu takiego, jak klasa dziedzicząca, jest również obiektem typu takiego jak klasa bazowa.

Kod utworzony dla klasy bazowej nigdy nie jest tracony.

Dlatego możliwa jest konwersja wskaźnika do obiektu takiego typu, jak klasa dziedzicząca, na wskaźnik takiego typu jak klasa bazowa.

Takie rzutowanie nazywane jest rzutowaniem w górę (*upcasting*).

Dlaczego w górę?

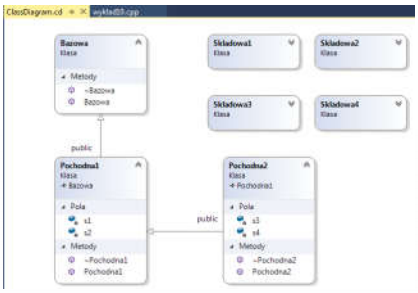
C++ - dziedziczenie

Tradycyjnie diagramy dziedziczenia były rysowane z klasą główną (najbardziej podstawową, bazową) znajdującą się na górze strony.

Diagram rozrastał się w dół poprzez dodawanie kolejnych dziedziczących klas

197

C++ - dziedziczenie



Struktura klas dla prezentowanego wcześniej przykładu (Visual Studio 2017)

198

C++ - dziedziczenie

Rzutowanie w górę jest bezpieczne, ponieważ od typu wyspecjalizowanego, z bogatszą listą metod i pól, przechodzimy do typu bardziej ogólnego, uboższego.

Jedyna zmiana w interfejsie klasy, wynikająca z takiego rzutowania, polega na tym, że może on utracić część metod i/lub pól (ponieważ typ bazowy ich nie ma), ale nie może w ten sposób uzyskać nowych metod i/lub pól.

Dlatego kompilator pozwala na rzutowanie w górę bez konieczności jawnych rzutowań ani żadnych innych szczególnych notacji

199

C++ - dziedziczenie

```
class Bazowa {
  public:
  int fun() {  return 0; }
  int fun(char *a) {  …; return 0; }
};
class Pochodna3: public Bazowa {
  public:
  int fun() {  return 0; }
};
int main(int argc, char *argv[]) {
  Pochodna3 p3;
  Bazowa *pb;
  pb = &p3;                    // rzutowanie w górę
  char napis[] = "Asta la vista";
  pb->fun(napis);              // (!)
  …
```


200

C++ - dziedziczenie

```
Bazowa *pb;
pb = &p3;                // rzutowanie w górę
char napis[] = "Asta la vista";
pb->fun(napis);
pb->fun();
```

Wywołanie obydwu metod za pomocą wskaźnika **pb** sprawi, że zostaną wywołane wersje zdefiniowane dla typu **Bazowa**

To może być problem: dla obiektów typu **Pochodna3** została przecież napisana inna wersja metody **int fun()**, która miała przedefiniować działanie tej należącej do klasy **Bazowa**

Aby tego uniknąć, należy wykorzystać polimorfizm obiektów

201

POLIMORFIZM

202

C++ - polimorfizm

Podstawowe pytanie, które nieustannie ma towarzyszyć autorowi programowi:

czy gdyby nagle okazało się, że jest więcej ... (danych na wejściu, typów danych, czynności, które program ma wykonać, etc.), to w wprowadzenie poprawek wiązałoby się z:

a) dodaniem kilku linijek kodu
b) przerobieniem kilku metod
c) przerobieniem kilku metod i klas
d) przerobieniem całego programu
e) kompletnym załamaniem się – tyle mojej roboty na marne..
O, nie! Zmuszę ich, żeby zrezygnowali ze swoich wymagań i używali mojego programu w tej cudownej postaci, jaką ma w tej chwili.

C++ - polimorfizm

Polimorfizm – jeden z trzech filarów obiektowego języka programowania (obok abstrakcji danych (hermetyzacji) i dziedziczenia)

„Dziedziczenie + polimorfizm" –
ułatwia tworzenie programów możliwych do rozszerzania;

1. Ten sam obiekt może być traktowany jakby był obiektem swojego typu, albo swojego typu bazowego.
2. To pozwala na traktowanie obiektów różnych typów tak, jakby były utworzone na podstawie jednego typu.
3. Dzięki temu pojedynczy fragment kodu może działać identycznie z różnymi typami danych.

C++ - polimorfizm

Metody wirtualne stanowią odpowiedź na problem towarzyszący rzutowaniu w górę:

```
Bazowa *pb;
pb = &p3;                  // rzutowanie w górę
char napis[] = "Asta la vista";
pb->fun(napis);            // odwołanie do kodu z Bazowej
pb->fun();                 // odwołanie do kodu z Bazowej
```

Połączenie wywołania metody z jej ciałem (kodem) nazywane jest wiązaniem. Jeżeli wiązanie wykonywane jest przed uruchomieniem programu, np. na etapie kompilacji, to mamy do czynienia z tzw. wczesnym wiązaniem (*early binding*).

Wczesne wiązanie występuje zawsze w programach w C.

Powyższy problem wynika właśnie z wczesnego wiązania: kompilator nie wie, jakiego typu naprawdę jest obiekt wskazywany przez 'pb', dlatego – żeby nie zgadywać – wiąże wywołania metod z kodem metod zadeklarowanym w klasie 'Bazowa'

C++ - polimorfizm

Rozwiązaniem jest późne wiązanie (*late binding*)

Inne nazwy: wiązanie dynamiczne (*dynamic binding*), wiązanie podczas wykonywania programu (*runtime binding*)

To wiązanie jest wykonywane w trakcie wykonania programu na podstawie informacji o rzeczywistym typie aktualnie wiązanego obiektu

W momencie kompilacji kompilator nie wiąże wywołania metody z konkretnym adresem metody, ale wstawia kod umożliwiający odnalezienie i wywołanie odpowiedniego ciała metody

Aby spowodować późne wiązanie jakiejś metody należy w jej deklaracji użyć słowa kluczowego **`virtual`**

C++ - polimorfizm

Metody wirtualne

Dwa obiekty dynamiczne różnych typów mających wspólną klasę bazową mogą być kontrolowane za pomocą tego samego wskaźnika do klasy bazowej a mimo to mogą wyrazić swoją odmienność.

Wywołanie metody wirtualnej uruchomi wykonanie metody w wersji właściwej dla typu obiektu – wersja metody zostanie ustalona dopiero w trakcie wykonania programu.

Deklaracja określonej metody jako wirtualnej musi mieć miejsce w klasie bazowej.

C++ - polimorfizm

Jeśli metoda została zadeklarowana w klasie bazowej jako wirtualna, to wersje przesłaniające tę metodę we wszystkich klasach pochodnych (nie tylko na pierwszym, ale również na wszystkich następnych poziomach dziedziczenia) są też wirtualne.

```
class A {
        virtual double fun(int, int);
        …
};
class B: public A {
        double fun(int, int);
        …
};
```

Powtarzanie deklaracji **`virtual`** w klasach pochodnych jest dopuszczalne, ale zbędne.

C++ - polimorfizm

```
class Bazowa {
   public:
   virtual int fun() { return 0; }
   int fun(char *a) { return 0; }
};

class Pochodna3: public Bazowa {
   public:
   int fun() {  return 0; }
};
```

```
int main(int argc, char *argv[])
   {
   Pochodna3 p3;
   Bazowa *pb;
   char napis[] = "Asta la vista";
   pb = &p3; // rzutowanie w górę
   pb->fun(napis);
   pb->fun(); // Tutaj!
   pb->Bazowa::fun();
   …
```


209

C++ - polimorfizm

Nie ma obowiązku definiowania w klasach pochodnych wszystkich metod zadeklarowanych w bazowych jako wirtualne

```
class Bazowa {
   public:
   virtual int fun() { return 0; }
};
class Pochodna3: public Bazowa {
   public:
   // int fun() { return 0; } - zakomentowaliśmy na chwilę,
   //                           zobaczymy co się stanie..
   …
};
int main(int argc, char *argv[]) {
   Pochodna3 p3;
   Bazowa *pb= &p3;   // rzutowanie w górę
   pb->fun();// zostanie wywołana wersja dla klasy 'Bazowa' –
             // bo nie ma innej
```


210

C++ - polimorfizm

Przesłaniając w klasie pochodnej metodę dziedziczoną z klasy bazowej możemy zawęzić jej dostępność, ale nie rozszerzyć.

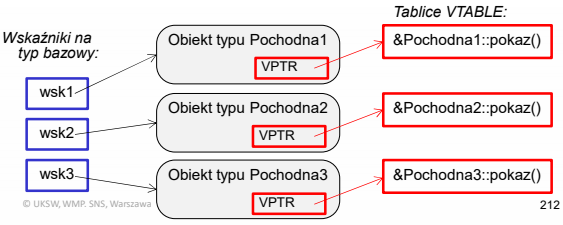
```
class Bazowa {
   public:
   virtual int fun() { return 0; }
};
class Pochodna3: public Bazowa {
   protected: //
   int fun() { return 0; }
   …
};
int main(int argc, char *argv[]) {
   Pochodna3 p3;
   Bazowa *pb= &p3;   // rzutowanie w górę
   pb->fun();// nie ma problemu z wywołaniem wersji z Pochodna3!
```


211

C++ - polimorfizm

Realizacja późnego wiązania

Typowy kompilator dla każdej klasy zawierającej metody wirtualne tworzy pojedynczą tablicę VTABLE na adresy jej wirtualnych metod. W obiektach tej klasy dodatkowo umieszczany jest wskaźnik VPTR wskazujący na VTABLE.



212

C++ - polimorfizm

Realizacja późnego wiązania

Wywołanie metody polimorficznej:

Gdy za pośrednictwem wskaźnika obiektu klasy bazowej wywołuje się metodę wirtualną, kompilator w tym miejscu wstawia kod, pobierający z aktualnie wskazywanego obiektu wskaźnik VPTR i odnajdujący we wskazanej tablicy adres żądanej metody wirtualnej.

Wszystkie te działania odbywają się automatycznie.

213

C++ - polimorfizm

Realizacja późnego wiązania

Korzystanie z polimorfizmu powoduje narzut w rozmiarze zajmowanej przez obiekt pamięci oraz w koszcie wykonania. Jaki?

1. W każdej klasie (bazowej i wszystkich pochodnych) oddzielna tablica wirtualna z adresami metod polimorficznych właściwymi dla obiektów danej klasy.
2. W każdym obiekcie wskaźnik na tablicę wirtualną jego klasy.
3. Dodatkowy kod w konstruktorze inicjalizujący ten wskaźnik.
4. Dodatkowy kod we wszystkich konstruktorach klas pochodnych reinicjalizujący wskaźnik w klasach bazowych po których klasa pochodna dziedziczy (obiekt typu pochodnego ma w sobie obiekt typu bazowego).
5. W miejscu każdego wywołania takiej metody kod ustalający na bieżąco adres właściwej metody polimorficznej, którą należy wywołać.

214

C++ - polimorfizm

Skoro polimorfizm jest takim ważnym elementem języka, to (mimo, że trochę kosztuje) dlaczego nie jest stosowany automatycznie we wszystkich wywołaniach metod?

Właśnie dlatego, że powoduje pewien nakład pamięciowy i obliczeniowy.

Język C++ jest spadkobiercą C, w którym efektywność ma podstawowe znaczenie. C powstał po to by zastąpić język assembler przy tworzeniu systemów operacyjnych. C++ miał sprawić, że programowanie miało być jeszcze bardziej efektywne.

Gdyby używanie C++ było podobnie wydajne jak C, ale przy każdym wywołaniu metody powodowało dodatkowy narzut obliczeniowy, większość użytkowników pozostałaby przy C. Dlatego ustalono, że metody wirtualne stanowią w C++ opcję.

C++ - polimorfizm

W trakcie projektowania nierzadko występuje potrzeba, by klasa podstawowa stanowiła wyłącznie interfejs dla swoich klas pochodnych – nie chcemy tworzenia obiektów klasy podstawowej, chcemy jedynie, aby doprowadziła ona do standaryzacji klas pochodnych.

- Takimi klasami będą klasy, w których pewne metody w ogóle nie są zdefiniowane, a tylko zadeklarowane.
- W dziedziczących klasach muszą zostać do tych metod dostarczone implementacje.
- Takie klasy to **klasy abstrakcyjne**

C++ - polimorfizm

Metodę wirtualna można zadeklarować jako **czysto wirtualną**, pisząc po nawiasie kończącym listę argumentów '=0', np. :

```
virtual void fun(int i) = 0;
```

Wystarczy, że wśród zadeklarowanych metod będzie tylko jedna taka wirtualna metoda, aby cała klasa stała się klasą abstrakcyjną.

C++ - polimorfizm

Przykład klasy abstrakcyjnej:

```
class Bazowa {                  // klasa abstrakcyjna
   public:
   virtual int fun() = 0 ;
};
class Pochodna3: public Bazowa {
   public:
   int fun() { return 0; }
   …
};
int main(int argc, char *argv[]) {
   Pochodna3 p3;
   Bazowa *pb = &p3;  // rzutowanie w górę
   pb->fun();// istnieje tylko wersja z klasy Pochodna3
   …
```


C++ - polimorfizm

Korzyści z klas abstrakcyjnych i metod czysto wirtualnych:

1. Pozwalają napisać dużą część kodu w terminach klas abstrakcyjnych, co upraszcza program i czyni łatwiejszym do modyfikacji. Klasy dziedziczące mogą zostać dospecyfikowane później.
2. Dzięki dziedziczeniu nie musimy dokładnie rozumieć jak metody z klasy bazowej mają działać, ważne, żeby były dobrze wyspecyfikowane warunki wywołania metody oraz skutki jej działania.
3. Deklarowanie metod czysto wirtualnych wymusza na wszystkich programistach piszących klasy dziedziczące definiowanie tych metod.

W funkcjach i metodach nie wolno przekazywać przez wartość argumentów typów abstrakcyjnych klas – można się natomiast odwoływać przez wskaźnik typu abstrakcyjnego