C++ - klasy

Metody stałe w klasie

Zadeklarowanie metody jako stałej (`const`) stanowi „obietnicę", że jej wykonanie nie zmieni stanu obiektu, na rzecz którego została wykonana

Umieszczamy `const` między nawiasem zamykającym listę argumentów, a średnikiem lub nawiasem klamrowym otwierającym:

```
class X {
  int i;
  public:
  X(int a=0);
  void funconst(X &ox) const;
  void fun(X &ox);
};
```

© UKSW, WMP. SNS, Warszawa

C++ - klasy

Jeżeli zadeklarujemy obiekt jako stały (`const`), to na jego rzecz możemy wywoływać tylko metody stałe:

```
X x1(0), x2(1);
const X x3(2);

x1.funconst(x2);      // OK.
x1.fun(x2);           // OK.
x1.funconst(x3);      // OK.
x1.fun(x3);           // Źle!
x3.fun(x1);           // Źle!
x3.funconst(x1);      // OK.
```

© UKSW, WMP. SNS, Warszawa

C++ - klasy

Argumenty wywołania metody/funkcji zadeklarowane jako stałe

Jeżeli wybrane argumenty zostaną zadeklarowane jako stałe, to nie wolno zmieniać ich wartości w kodzie funkcji. Jest to gwarancja bezpieczeństwa danych dla użytkownika tej metody/funkcji

Np. funkcja do kopiowania łańcuchów tekstowych:

```
char *strcpy (char* strTo, const char* strFrom);
```

Funkcja przyjmuje dwa argumenty: skąd kopiować i dokąd kopiować

W nagłówku jest zadeklarowane, że obszar pamięci wskazywany przez `strFrom` nie zostanie zmodyfikowany w wyniku działania tej funkcji

© UKSW, WMP. SNS, Warszawa

C++ - klasy

Konstruktor kopiujący (*dokończenie*)

Konstruktor kopiujący nie powinien modyfikować obiektu, który jest mu podawany w argumencie wywołania, dlatego można zadeklarować ten argument jako stały (nie podlegający zmianom):

```
class MojaKlasa {
  public:
  MojaKlasa();                          // domyślny
  MojaKlasa(const MojaKlasa& mk);       // kopiujący
  …
};
```

© UKSW, WMP. SNS, Warszawa

PRZECIĄŻANIE METOD

© UKSW, WMP. SNS, Warszawa

C++ - klasy

Przeciążanie metod:

Przykład: metoda, który wypisuje słownie liczbę:

- Jeżeli liczba całkowita **12**, napis: **dwanaście**
- Jeżeli liczba rzeczywista **12**, napis: **dwanaście przecinek zero**

Możemy w klasie zadeklarować dwie metody o tej samej nazwie `napisz_slownie`, różniące się argumentami wywołania:

```
class X {
public:
  void napisz_slownie(int);
  void napisz_slownie(double);
};
```

© UKSW, WMP. SNS, Warszawa

C++ - klasy

Przeciążanie konstruktorów:

Tak samo jak metody, przeciążamy konstruktory:

```
class X {
  int i;
  public:
  X();            // konstruktor domyślny
  X(int a);       // inny konstruktor
};
```

Możemy też redukować liczbę metod/konstruktorów, stosując argumenty domyślne, np.:

```
...
X(int a=0);
```

© UKSW, WMP. SNS, Warszawa

C++ - klasy

Argumenty domyślne:

```
class X {
  int i;
  public:
  X(int a=0);
  void fun(int, int, int a=0, double b = 0, double c = 0);
};

X x1[3] = {X(), X(1), X(2)};
```

Pierwszy z obiektów w tablicy będzie inicjalizowany konstruktorem, w którym przyjęto domyślną wartość argumentu – 0.

Zasada: domyślne mogą być tylko końcowe argumenty (może być ich kilka).

```
  x1.fun(12, 21, 1);      // pominięto argumenty b i c
                          // - przyjmą wartości 0
```

© UKSW, WMP. SNS, Warszawa

LISTY DYNAMICZNE

© UKSW, WMP. SNS, Warszawa

C++ - listy dynamiczne

Listy jednokierunkowe obiektów

Z dynamicznych obiektów możemy tworzyć listy dynamiczne w taki sam sposób, jak z dynamicznych zmiennych strukturalnych

```
struct film_t {

    char tytul[80];
    int rok;
    struct film_t *nast;
};
```

```
class film_t {
public:
    char tytul[80];
    int rok;
    film_t *nast;
};
```

© UKSW, WMP. SNS, Warszawa

C++ - listy dynamiczne

Wykorzystanie konstruktora może uprościć kod:

```
class film_t {
public:
  char tytul[80];
  int rok;
  film_t *nast;
  film_t() {tytul[0]=0;
            rok=0;
            nast=NULL; };
  film_t(char*s, int r) {
   strcpy(tytul, s);
   rok=r;
   nast=NULL;
  };
};
```

```
FILE*stream;
film_t *glowa, *wsk;
… // otwarcie pliku
char buffer[100];
int r;
fscanf(stream,"%s %i\n",buffer,r);
while (!feof( stream )) {
  if (glowa == NULL)
      glowa=wsk=new film_t(buffer,r);
  else {
      wsk->nast=new film_t(buffer,r);
      wsk=wsk->nast;
  }
  fscanf(stream,"%s %i\n",buffer,r);
}
… // dalszy kod programu
```

© UKSW, WMP. SNS, Warszawa

C++ - listy dynamiczne

Niektóre czynności można zamknąć w kodzie destruktora:

Usuwanie tradycyjne (tak jak dla struktur):

```
wsk = glowa;
while (glowa != NULL) {
      glowa = glowa->nast;
      delete wsk;
      wsk = glowa;
};
```

Usuwanie z wykorzystaniem destruktora:

```
film_t::~film_t() {
   delete nast;
};
…
delete glowa;  // rekurencja!
```

© UKSW, WMP. SNS, Warszawa

this

© UKSW, WMP. SNS, Warszawa

164

C++ - klasy

Wskaźnik `this`

Kiedy wywołujemy metodę na rzecz istniejącego obiektu, to możemy w tej metodzie wywołać publiczne metody innych obiektów, np.:

```
class Pierwsza {
   void fun1(Druga*);
};
void Pierwsza::fun1(Druga *d) {
   d->j = 0;
   …           // tutaj pozostałe instrukcje fun1
}
```

Przyjmijmy, że w klasie **`Druga`** jest metoda **`fun2`**, której argumentem jest wskaźnik do obiektu klasy **`Pierwsza`** i chcemy właśnie tę metodę wywołać w metodzie **`fun1`**:

© UKSW, WMP. SNS, Warszawa

165

C++ - klasy

Wskaźnik `this`

```
void Druga::fun2(Pierwsza *p) {
   …
}

void Pierwsza::fun1(Druga *d) {
   d->j = 0;
   d->fun2( co tu napisać? );
}
```

Problem: metoda **`fun1`** należąca do klasy **`Pierwsza`** chce wywołać metodę **`fun2`** należącą do klasy **`Druga`**. Argumentem wywołania **`fun2`** jest wskaźnik na obiekt typu **`Pierwsza`**. To wywołanie **`fun2`** odbywa się wewnątrz metody należącej do **`Pierwsza`**, więc w wywołaniu należy przekazać wskazanie na samego siebie. *Tylko skąd je wziąć?*

© UKSW, WMP. SNS, Warszawa

166

C++ - klasy

Wskaźnik `this`

..należy odwołać się do wskaźnika **`this`**, który zawiera adres pożądanego obiektu.

Każda z metod klasy, niezadeklarowanych jako **`static`**, w momencie wywołania otrzymuje, oprócz argumentów z jawnie zadeklarowanej listy parametrów wywołania, wskazanie na obiekt, na rzecz którego wystąpiło wywołanie. To wskaźnik o nazwie **`this`**.

Kompilator jest odpowiedzialny za incjalizację **`this`** prawidłowym adresem obiektu.

Wskaźnika **`this`** nie można modyfikować, np. przypisać mu innej wartości.

© UKSW, WMP. SNS, Warszawa

167

C++ - klasy

Teraz już wiadomo co wpisać:

```
void Druga::fun2(Pierwsza *p) {
   …
}

void Pierwsza::fun1(Druga *d) {
   d->j = 0;
   d->fun2( this );
}
```

W ten sposób w kodzie należącym do obiektu możemy manipulować na tymże obiekcie w taki sam sposób, jakbyśmy działali na dowolnym innym tego samego typu.

© UKSW, WMP. SNS, Warszawa

168

C++ - klasy

Kiedy jeszcze używać `this`:

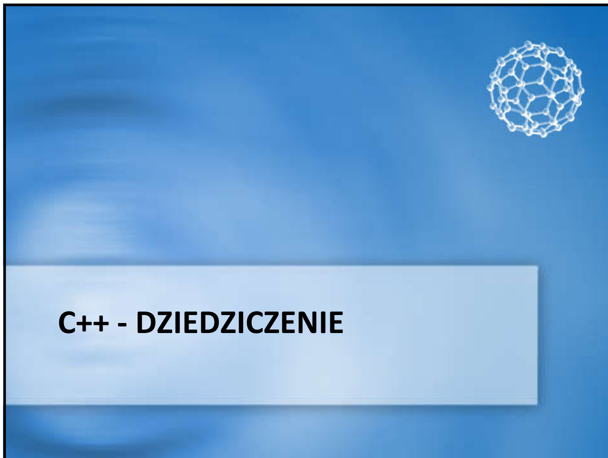
```
class MyClass
{
public:
  void f(int);
private:
  int x;
};

void MyClass::f(int x)
{
..
  this->x = x;
..
}
```

Ale można tego uniknąć.. Np. tak:

```
class MyClass
{
public:
  void f(int);
private:
  int x;
};

void MyClass::f(int xval)
{
..
  x = xval;
..
}
```

© UKSW, WMP. SNS, Warszawa

169



170

C++ - dziedziczenie

Do najważniejszych cech języka C++ należy możliwość wielokrotnego wykorzystywania kodu

Prymitywnym, ale skutecznym sposobem jest *kompozycja*: deklarowanie pól obiektowych wewnątrz innych klas, tak że nowe klasy są zestawiane z obiektów tych klas, które już istnieją:

```
class Pierwsza {
public:
      oblicz(double x);
      …
};
```

```
class Druga {
public:
          Pierwsza x;
          Druga(): x() { }
…
};
…
Druga y;
y.x.oblicz(123);
```


171

C++ - dziedziczenie

Sposobem dającym dużo więcej możliwości jest *dziedziczenie*

Stosując dziedziczenie oznajmia się: nowa klasa jest podobna do tamtej, istniejącej już klasy. Inaczej mówiąc: nowa klasa

1. dziedziczy po tamtej klasie jej atrybuty i usługi (tj. pola i metody), w takim zakresie, w jakim określają to w tamtej klasie prawa dostępu,
2. a ponadto ma kilka oryginalnych swoich własnych atrybutów i usług.

Deklaracja nowej klasy dziedziczącej po klasie tzw. bazowej:

```
class KlasaDziedziczaca: public KlasaBazowa {
    … };
```

Obiekt utworzony z klasy dziedziczącej jest szczególnym przypadkiem obiektu innego typu: ma wszystkie cechy tego obiektu być może uzupełnione lub zmodyfikowane nowymi właściwościami.

172

C++ - dziedziczenie

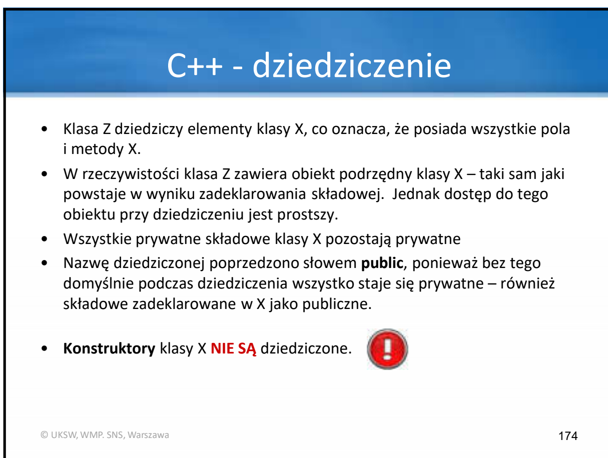
Kompozycja:

```
class X {
      public:
      oblicz(double x);
 };
class Y {
      public:
      X x;
      …
 };

Y y;
y.x.oblicz(123);
```

Dziedziczenie:

```
class Z: public X {
 …
};

Z z;
z.oblicz(123);
```


173



C++ - dziedziczenie

- Klasa Z dziedziczy elementy klasy X, co oznacza, że posiada wszystkie pola i metody X.
- W rzeczywistości klasa Z zawiera obiekt podrzędny klasy X – taki sam jaki powstaje w wyniku zadeklarowania składowej. Jednak dostęp do tego obiektu przy dziedziczeniu jest prostszy.
- Wszystkie prywatne składowe klasy X pozostają prywatne
- Nazwę dziedziczonej poprzedzono słowem **public**, ponieważ bez tego domyślnie podczas dziedziczenia wszystko staje się prywatne – również składowe zadeklarowane w X jako publiczne.
- **Konstruktory** klasy X **NIE SĄ** dziedziczone.

174

C++ - dziedziczenie

Terminologia dla

`class Dziedziczaca: public Bazowa`

`Dziedziczaca` jest klasą pochodną względem klasy `Bazowa`
`Dziedziczaca` jest szczególnym przypadkiem klasy `Bazowa`
`Dziedziczaca` jest specjalizacją klasy `Bazowa`
`Dziedziczaca` jest podklasą klasy `Bazowa`
`Bazowa` jest klasą bazową klasy `Dziedziczaca`
`Bazowa` jest nadklasą klasy `Dziedziczaca`

175

C++ - dziedziczenie

Tworzenie nowego obiektu

W przypadku kompozycji jedynym sposobem wywołania konstruktora dla składowej obiektowej klasy jest lista incjalizatorów konstruktora

Należy podać nazwę składowej (pola) oraz w nawiasach argumenty dla konstruktora, np. :

```
class Pierwsza {
        public:
        Pierwsza(double x, double y);
    };
class Druga {
        public:
        Pierwsza x;
        Druga(double a): x(a,0) { … }
    };
```

A co w przypadku dziedziczenia?

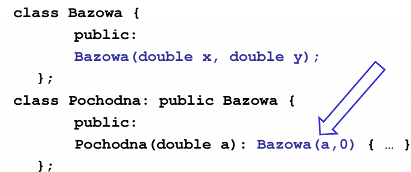
© UKSW, WMP. SNS, Warszawa

C++ - dziedziczenie

Tworzenie nowego obiektu

W przypadku dziedziczenia w liście inicjalizatorów należy podać nazwę klasy bazowej oraz w nawiasach argumenty dla wybranego konstruktora, np.:

```
class Bazowa {
        public:
        Bazowa(double x, double y);
    };
class Pochodna: public Bazowa {
        public:
        Pochodna(double a): Bazowa(a,0) { … }
    };
```



© UKSW, WMP. SNS, Warszawa

C++ - dziedziczenie

Kompozycję i dziedziczenie można łączyć:

```
class Pierwsza {
        public:
        Pierwsza(double x, double y);
    };
class Bazowa {
        public:
        Bazowa(double x);
    };
class Pochodna: public Bazowa {
        public:
        Pierwsza x;
        Pochodna(double a): x(a,0), Bazowa(a) { … }
    };
```

© UKSW, WMP. SNS, Warszawa

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.

© UKSW, WMP. SNS, Warszawa

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);
};
```

Program #7

© UKSW, WMP. SNS, Warszawa

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..

© UKSW, WMP. SNS, Warszawa

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");

    …
```

© UKSW, WMP. SNS, Warszawa

182

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 funkcji wirtualnych powoduje też dalsze konsekwencje (*o czym będzie w dalszej części wykładu*).

© UKSW, WMP. SNS, Warszawa

183

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

© UKSW, WMP. SNS, Warszawa

184

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 ☺.

© UKSW, WMP. SNS, Warszawa

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 funkcje
  …
};
```

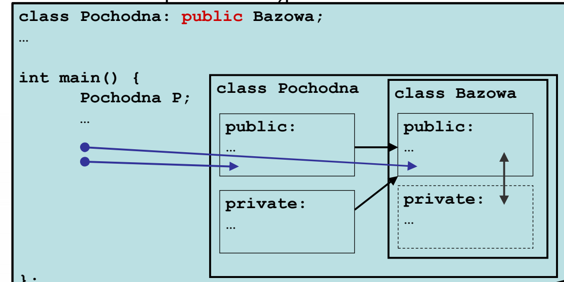
© UKSW, WMP. SNS, Warszawa

186

C++ - dziedziczenie

Dziedziczenie – prawa dostępu:

```
class Pochodna: public Bazowa;
…

int main() {
       Pochodna P;
       …
};
```



© UKSW, WMP. SNS, Warszawa

187

C++ - dziedziczenie

Dziedziczenie – prawa dostępu:

```
class Pochodna: private Bazowa;
…

int main() {
        Pochodna P;
        …
```

class Pochodna: public: … / private: …

class Bazowa: private: … / private: …

```
};
```

© UKSW, WMP. SNS, Warszawa

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.

© UKSW, WMP. SNS, Warszawa

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
    ..
}
```

© UKSW, WMP. SNS, Warszawa

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**.

© UKSW, WMP. SNS, Warszawa

C++ - dziedziczenie

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

class Pochodna: public: … / protected: … / private: …

class Bazowa: public: … / protected: … / private: …

© UKSW, WMP. SNS, Warszawa

C++ - dziedziczenie

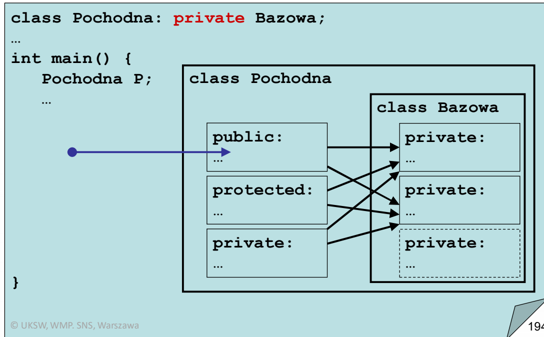
```
class Pochodna: protected Bazowa;
…
int main() {
    Pochodna P;
    …
}
```

class Pochodna: public: … / protected: … / private: …

class Bazowa: protected: … / protected: … / private: …

© UKSW, WMP. SNS, Warszawa

C++ - dziedziczenie



© UKSW, WMP. SNS, Warszawa 194

194

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 funkcje 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.

© UKSW, WMP. SNS, Warszawa 195

195

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?

© UKSW, WMP. SNS, Warszawa 196

196

C++ - dziedziczenie

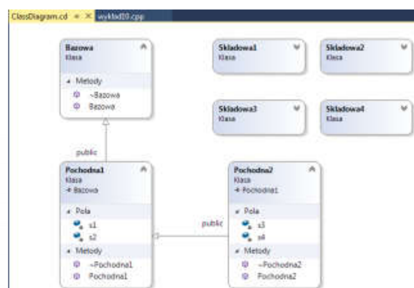
Tradycyjnie diagramy dziedziczenia były rysowane z klasą główną (najbardziej podstawową, bazową) znajdującą się na górze strony.

Diagram rozrastał się w dół poprzez dodawanie kolejnych dziedziczących klas

© UKSW, WMP. SNS, Warszawa 197

197

C++ - dziedziczenie



Struktura klas dla prezentowanego wcześniej przykładu (Visual Studio 2017)

© UKSW, WMP. SNS, Warszawa 198

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

© UKSW, WMP. SNS, Warszawa 200

200

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);              // (!)
  …
```

© UKSW, WMP. SNS, Warszawa

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

© UKSW, WMP. SNS, Warszawa