203

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 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.

204

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.

205

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 funkcji 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'

206

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`**

207

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.

208

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.

209

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();
   …
```

Visual Studio

210

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
```


211

C++ - polimorfizm

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

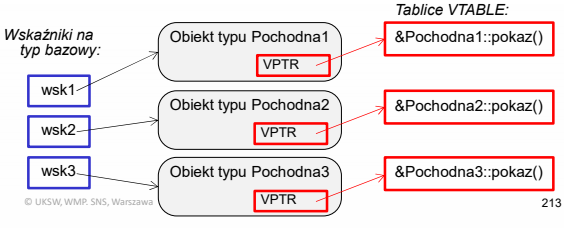
```
class Bazowa {
   public:
   virtual int fun() { return 0; }
};
class Pochodna3: public Bazowa {
   protected: //
   int fun() { return 0; }
   …
};
int main(int argc, char *argv[]) {
   Pochodna3 p3;
   Bazowa *pb= &p3;   // rzutowanie w górę
   pb->fun();// nie ma problemu z wywołaniem wersji z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


212

C++ - polimorfizm

Realizacja późnego wiązania

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



213

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 klasy aktualnie wskazywanego obiektu wskaźnik VPTR i odnajdujący we wskazanej tablicy adres żądanej metody wirtualnej.

Wszystkie te działania odbywają się automatycznie.

214

C++ - polimorfizm

Realizacja późnego wiązania

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

1. W klasie bazowej tablica wirtualna z adresami metod polimorficznych
2. W każdym obiekcie wskaźnik na tablicę wirtualną jego klasy
3. Dodatkowy kod w konstruktorze inicjalizujący ten wskaźnik
4. Tablica wirtualna w klasie pochodnej, ale wypełniona innymi adresami, niż w bazowej
5. 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)
6. W miejscu każdego wywołania takiej metody kod ustalający na bieżąco adres właściwej metody polimorficznej, którą należy wywołać.

215

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 funkcji powodowało dodatkowy narzut obliczeniowy, większość użytkowników pozostałaby przy C. Dlatego ustalono, że funkcje wirtualne stanowią w C++ opcję.

216

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**

217

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.

218

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
  …
```


219

C++ - polimorfizm

Korzyści z klas abstrakcyjnych i metod 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 (*budowa domu zaczynając od dachu*).
2. Dzięki dziedziczeniu nie musimy dokładnie rozumieć jak metody z klasy bazowej działają, ważne, żeby były dobrze wyspecyfikowane warunki wywołania metody oraz skutki jej działania.
3. Deklarowanie metod wirtualnych wymusza na wszystkich programistach piszących klasy dziedziczące definiowanie tych metod.

W funkcjach nie wolno przekazywać przez wartość argumentów typów abstrakcyjnych klas – do takich obiektów można się odwoływać tylko przez wskaźnik typu abstrakcyjnego

220

C++ - polimorfizm

Jeżeli w klasie dziedziczącej jest metoda, która nie została zadeklarowana w klasie bazowej, ale potrzebujemy się do niej odwołać, musimy zastosować rzutowanie wskaźnika:

```
class Bazowa {                  // klasa abstrakcyjna
   public:
   virtual int fun() = 0 ;
};
class Pochodna3: public Bazowa {
   public:
   int fun() { … ; return 0; }
   int fun2(char *s) { printf("%s\n", s); }
};
int main(int argc, char *argv[]) {
   Pochodna3 p3;
   Bazowa *pb = &p3;  // rzutowanie w górę
   ((Pochodna3*)pb)->fun2("Asta la vista");
```


C++ - polimorfizm

Okrajanie obiektów

Jeżeli do funkcji przekażemy obiekt przez wartość, a nie przez wskaźnik, przy czym oczekiwany typ argumentu to klasa bazowa, podczas gdy w argumencie podajemy obiekt typu klasa pochodna, to rzutowany w ten sposób obiekt zostanie okrojony.

To co pozostanie i będzie przekazane do wewnątrz funkcji, stanowi *podobiekt*, odpowiadający typowi, do którego dokonywane było rzutowanie.

C++ - polimorfizm

```
class Bazowa {
  public:
    virtual int fun() { printf("%s\n", "bazowa"); return 0; }
};
class Pochodna3: public Bazowa {
  public:
    int fun() {  printf("%s\n", "pochodna3"); return 0; }
};
void podpis(Bazowa b) {
     b.fun();
}
int main(int argc, char *argv[])
{
    Pochodna3 p3;
    podpis(p3);        // co pojawi się w oknie konsoli?
```


C++ - polimorfizm

Konstruktory nie mogą być wirtualne.

Konstruktory mogą być wywoływane tylko kolejno od konstruktora klasy bazowej poprzez kolejne konstruktory w klasach dziedziczących w kolejności hierarchii dziedziczenia.

Pytanie: jeżeli w którymkolwiek z konstruktorów zostanie wywołana metoda wirtualna, to która jej wersja zostanie uruchomiona?

Odpowiedź: **Zawsze wersja lokalna.**

Ta zadeklarowana w klasie do której należy konstruktor, który ją wywołał, lub odziedziczona.

C++ - polimorfizm

Destruktory mogą być wirtualne.

Po co?

- Destruktory są wywoływane w kolejności odwrotnej do kolejności konstruktorów – od najniższego.
- Co będzie, jeżeli będziemy chcieli wywołać operator **`delete`** dla obiektu, wskazywanego przez wskaźnik na typ bazowy tego obiektu?
  *Operator **`delete`** ocenia typ obiektu po typie wskaźnika. Który destruktor wywoła?*

C++ - polimorfizm

```
class Bazowa {
  public:
    ~Bazowa() { printf("%s\n","~Bazowa"); }
};
class Pochodna3: public Bazowa {
  public:
    ~Pochodna3() { printf("%s\n","~Pochodna3"); }
};

int main(int argc, char *argv[])
{
    Bazowa *bp = new Pochodna3;
    delete bp; // jaki napis pojawi się w oknie?
```

C++ - polimorfizm

```
class Bazowa {
  public:
    virtual ~Bazowa() { printf("%s\n","~Bazowa"); }
};
class Pochodna3: public Bazowa {
  public:
    ~Pochodna3() { printf("%s\n","~Pochodna3"); }
};

int main(int argc, char *argv[])
{
    Bazowa *bp = new Pochodna3;
    delete bp; // jaki napis pojawi się w oknie?
```

Visual Studio

227

C++ - polimorfizm

Wywoływanie metod wirtualnych w destruktorach – jest możliwe, ale która wersja metody wirtualnej zostanie uruchomiona? Powinna – i jest – wersja lokalna.

W destruktorze mechanizm późnego wiązania nie działa.

Jest to zabezpieczenie, które ma chronić przed próbą wywołania metody niższego poziomu dziedziczenia w destruktorze wyższego poziomu. Metoda niższego poziomu mogłaby próbować odwołać się do składników klasy, niższego poziomu, które już zostały zniszczone..

228

WSKAŹNIKI KLASOWE

229

C++ - klasy

Wskaźniki klasowe

- Każdy obiekt zajmuje fragment pamięci i wszystkie obiekty tego samego typu zajmują fragmenty pamięci tej samej długości
- początek miejsca w pamięci zajmowanego przez obiekt jest nazywany wskaźnikiem na ten obiekt
- wskaźnik przechowuje adres pierwszej komórki pamięci od której zaczyna się fragment pamięci zajęty przez ten obiekt
- miejsce w pamięci, w którym przechowywane są poszczególne składowe obiektu jest zawsze w tym samym położeniu względem początku obiektu
- wartość przesunięcia względem początku to adres względny składowej
- mając adres względny i adres bezwzględny obiektu możemy odwołać się do składowej obiektu za pomocą jej adresu bezwzględnego

230

C++ - klasy

Wskaźniki klasowe do pól

Żeby przechować wartość adresu względnego składowej w zmiennej, potrzebujemy mieć dla niej zdefiniowany typ.

W C++ kontrola typów jest bardzo restrykcyjna, więc nie może to być jakiś ogólny typ, ale konkretnie przeznaczony do tego jednego typu składowej. Składowa innego typu będzie miała zdefiniowany inny typ dla swojego adresu względnego.

Przyjmijmy, że w klasie **`MojaKlasa`** są zdefiniowane składowe typu **`int`**. Wtedy wskaźnik względny do składowych tego typu będzie miał typ:

```
int MojaKlasa::*wsk;
```

Uwaga: wskaźnikom klasowym nie można przypisać wskazań na składowe statyczne – w tym przypadku stosuje się zwykłe wskazania.

231

C++ - klasy

Wskaźniki klasowe do pól

```
class MojaKlasa {
    public:
    int a,b;
    MojaKlasa();
    int fun(int x);
};
int MojaKlasa::*wska = &MojaKlasa::a;
int MojaKlasa::*wskb = &MojaKlasa::b;
```

Symbol '&' nie oznacza tu pobrania bezwzględnego adresu żadnego obiektu, ale oznacza, że wartościami wskaźników klasowych **`wska`** i **`wskb`** będzie przesunięcie składowej względem dowolnego obiektu klasy **`MojaKlasa`**.

232

C++ - klasy

Wskaźniki klasowe do pól

```
class MojaKlasa {
    public:
    int a,b;
    MojaKlasa();
    int fun(int x);
};
int MojaKlasa::*wska = &MojaKlasa::a;
MojaKlasa mk;
MojaKlasa *mkwsk = NULL;
…   // tutaj tworzymy obiekt typu 'MojaKlasa'
mk.*wska = 3;
mkwsk->*wska = 3;
```


C++ - klasy

Wskaźniki klasowe do metod

```
class MojaKlasa {
    public:
    int a,b;
    MojaKlasa();
    int fun(double x);
};
int (MojaKlasa::*funwsk)(double);     // tworzymy wskaźnik
funwsk = &MojaKlasa::fun;      // inicjalizujemy wskaźnik
MojaKlasa mk;
MojaKlasa *mkwsk = NULL;
…   // tutaj tworzymy obiekt typu 'MojaKlasa'
(mk.*funwsk)(3.14);
(mkwsk->*funwsk) (3.14);
```