Paradygmaty programowania obiektowego

Rzutowanie typów w programowaniu obiektowym

dr inż. Radosław Idzikowski

Rzutowanie podstawowe

Rzutowanie stosuje się w celu zmiany typu obiektu. Podstawowym rzutowaniem jest „*rzutowanie w stylu C*”:

```
typ obiekt = (typ) obiekt_innego_typu;

int obiekt = (int) obiekt_innego_typu;
```

oraz „*rzutowanie w stylu funkcyjnym*”:

```
typ obiekt = typ(obiekt_innego_typu);

int obiekt = int(obiekt_innego_typu);
```

Oba rzutowania działają dokładnie tak samo.

Wady podstawowych rzutowań

- mało czytelne (trudno wypatrzeć),
- potencjalne miejsce wystąpienia błędów,
- trudne do wyszukania.

Rzutowanie w C++

Rodzaje rzutowań:

- niejawne,
- jawne:
  - `static_cast` – rzutowanie statyczne,
  - `const_cast` – rzutowanie typu const,
  - `reinterpret_cast` – rzutowanie reinterpretacyjne,
  - `dynamic_cast` – rzutowanie dynamiczne.

Wprowadzono różne typy rzutowań w celu poprawy bezpieczeństwa. Użycie nieodpowiedniego rzutowania spowoduje błąd kompilacji.

Rzutowanie niejawne

Nazywane często automatycznym lub domyślnym, to proces automatycznego konwertowania jednego typu danych na inny przez kompilator bez jawnego uzycia operatora.

```
int i = 10;
double d = i;

float f = 2.5;
int j = f;
```

W przypadku rzutowania niejawnego, istnieje ryzyko utraty danych, ponieważ nie ma kontroli nad dokładnością konwersji.

Rzutowanie statyczne

Zapewnia wysoki poziom bezpieczeństwa – rzutowanie gwarantuje możliwie sensowny rezultat, zmieniając nawet w razie potrzeby reprezentacje bitową obiektu poddanego rzutowaniu. Tzn. przy rzutowaniu zmiennej typu `int` na `double` zostaną zmienione bity reprezentacji wewnętrznej według formatu dla zmiennej typu `double`.

```
int numberI = 9;
double numberD = static_cast<double>(numberI);
```

Rzutowanie statyczne

Rzutowanie statyczne do poprawnego działania wymaga odpowiedniej konwersji.

```
std::string str = "Hello World!";
std::cout<<static cast<char*>(str)<<"\n";
```

Błąd! Brak konwersji.

Operator `static_cast` służy w szczególności do:

- konwersji typów podstawowych,
- konwersji zdefiniowanych przez użytkownika.

Konwersja zdefiniowana

```
class Vector;

class Point {
public:
    double x, y;
    Point(double _x, double _y) : x(_x), y(_y){}
    Point(Vector& v);
};

std::ostream& operator<<(std::ostream& os, Point point) {
    os << "(" << point.x << "," << point.y <<")";
    return os;
}
```

Konwersja zdefiniowana

```
class Vector {
public:
    Point first;
    Point second;
    Vector(Point f, Point s) : first(f), second(s) {}
    Vector(Point& p) : first(p), second(Point(0, 0)) {}
};

Point::Point(Vector& v) : x(v.first.x), y(v.second.y) {}

int main()
{
    Vector vec(Point(-1, 1), Point(2, -2));
    Point point = static_cast<Point>(vec);
    std::cout << point << "\n";
    Vector vec2 = static_cast<Vector>(point);
    std::cout << vec2.first << "\n";
    std::cout << vec2.second << "\n";
}
```

Konwersja wskaźnika obiektów

```
class Animal {
public:
    void virtual voice() const {};
};

class Dog : public Animal {
public:
    void voice() const {
        std::cout << "woof!\n";
    }
};

int main()
{
    Dog* dog = new Dog();
    Animal *animal = static_cast<Animal*>(dog);
    Dog* dogy = static_cast<Dog*>(animal);
}
```

Konwersja na wskaźnikach nie wymaga definiowania konstruktora.

Rzutowanie statyczne – podsumowanie

Rzutowanie statyczne nie nadaje się do konwersji wskaźników różnych typów, jeśli nie ma specjalnie zdefiniowanej konwersji miedzy tymi wskaźnikami.

```
unsigned int ui;
int i = static_cast<int>(ui);
unsigned int *wui;
int *wi = static cast <int*>(wui);
```

Błąd!
Ponadto wynik działania `static_cast` jest ustalany w czasie kompilacji. Rzutowanie statyczne służące do konfesji referencji działa tak samo jak na wskaźnikach.

```
Dog dog = Dog();
Dog& rdog = dog;
Animal& ranimal = static_cast<Animal&>(rdog);
Dog& rdogy = static_cast<Dog&>(ranimal);
```

Rzutowanie typu const

Służy do nadawania bądź zdejmowania kwalifikatorów `const` oraz `volatile`.

```
void fun(const float* ptr) {
    float* ptr2 = const_cast<float*>(ptr);
    *ptr2 = 2.71f;
}
int main() {
    const float number = 3.14f;
    fun(&number);
    std::cout << number << std::endl;
    return 0;
}
```

Typ musi być wyrażony przez wskaźnik lub referencje.

Kwalifikacja wskaźników – przypomnienie

wskaźnik	schemat	modyfikowanie	przesuwanie
zwykły	`typ* wsk`	tak	tak
do obiektu stałego	`const typ* wsk`	nie	tak
stały	`typ* const wsk`	tak	nie
stały do obiektu stałego	`const typ* const wsk`	nie	nie

Rzutowanie typu `const` – podsumowanie

- umożliwia usunięcie lub dodanie kwalifikatora `const` do wskaźnika lub referencji,
- `const` wskazuje na to, że wartość zmiennej nie powinna być zmieniana w trakcie działania programu,
- użycie kwalifikatora `const` jest często stosowane do ochrony danych przed przypadkowymi modyfikacjami,
- kompilator może optymalizować kod, zakładając, że zmienne oznaczone jako `const` nie ulegną zmianie, co może wpłynąć na wydajność programu,
- kwalifikator `volatile` informuje kompilator, że wartość zmiennej może być modyfikowana poza programem i zmusza go do odczytywania jej wartości z pamięci za każdym razem, gdy jest ona używana, co jest szczególnie ważne w programowaniu mikrokontrolerów.

Rzutowanie reinterpretacyjne

Pozwala na interpretację wartości jednego typu jako wartości innego typu bez zmiany samej wartości.

```
float* f = new float;
*f = 3.14;
int* i = reinterpret_cast<int*>(f);
std::cout << "float: " << *f << ", int: " << *i << "\n";
```

oraz do konwersji między wskaźnikami na różne typy obiektów, które dzielą tę samą przestrzeń pamięci"

```
long int adres = 0x01FF643C6330;
long long pi = 0x0000000B;
long int* wsk = reinterpret_cast<long int*>(adres);
std::cout << "Wskaznik wsk wskazuje na: " << wsk << "\n";
wsk = reinterpret_cast<long int*>(&pi);
std::cout << "Teraz wsk wskazuje na: " << wsk << "\n";
std::cout << *wsk << "\n";
```

Rzutowanie reinterpretacyjne – podsumowanie

- rzutowanie reinterpretacyjne umożliwia interpretowanie wartości pamięci w różny sposób,
- za pomocą rzutowania reinterpretacyjnego możemy zmieniać typy wskaźników,
- przy użyciu rzutowania reinterpretacyjnego konieczne jest uważne projektowanie kodu i testowanie jego działania, ponieważ nieprawidłowe użycie może prowadzić do nieoczekiwanych błędów i problemów z wydajnością programu.

Rzutowanie dynamiczne

Konwersja wskaźników oraz referencji klas bazowych na ich odpowiedniki dla klas pochodnych oraz odwrotnie podczas wykonywania programu.

```
class Animal {
public:
    void virtual voice() const {};
};

class Dog : public Animal {
public:
    void voice() const {
        std::cout << "woof!\n";
    }
};

int main()
{
    Dog* dog = new Dog();
    Animal* animal = dynamic_cast<Animal*>(dog);
    Dog* dogy = dynamic_cast<Dog*>(animal);
}
```

Rzutowanie dynamiczne

Używane do rzutowania wskaźników lub referencji do obiektów na obiekty klas pochodnych. Aby użyć rzutowania dynamicznego w dół, należy spełnić następujące wymagania:

- klasa bazowa musi zawierać co najmniej jedną metodę wirtualną,
- klasa pochodna musi być wywoływana za pomocą wskaźnika lub referencji do klasy bazowej,
- rzutowanie musi być zgodne z hierarchią dziedziczenia klas.

W razie niekompatybilności zwróci:

- wartość `NULL` dla rzutowania wskaźników,
- wyjątek `bad_cast` dla rzutowania referencji.

```
class Animal {
public:
    void virtual voice() const {};
};

class Dog : public Animal {
public:
    void voice() const {
        std::cout << "woof!\n";
    }
};

Animal* fun(bool base) {
    if (base) return new Animal;
    else return new Dog;
}

int main()
{
    Animal* wsk = fun(true);
    Dog* dogy = dynamic_cast<Dog*>(wsk);
    if(dogy) dogy->voice();
}
```

Rzutowanie dynamiczne – podsumowanie

- jest używane do rzutowania wskaźników i referencji pomiędzy obiektami klas w hierarchii dziedziczenia,
- pozwala to na bezpieczne korzystanie z polimorfizmu w C++,
- wymagane jest, aby obiekty miały informację o swoim typie (zazwyczaj poprzez funkcje wirtualne),
- używane w przypadkach, gdy potrzebujemy przeprowadzić konwersję obiektów klasy bazowej na obiekty klas pochodnych, a jednocześnie potrzebujemy sprawdzić, czy rzutowanie jest możliwe i bezpieczne,
- jest mniej elastyczne i mniej wygodne niż rzutowanie statyczne lub reinterpretacyjne.

Zasady poprawnego stosowania rzutowania typów

- Staraj się unikać rzutowania, gdy to możliwe.
- Stosuj rzutowanie jawne tylko wtedy, gdy nie ma innej opcji lub gdy konieczne jest zachowanie zgodności typów.
- Używaj rzutowania statycznego, gdy jesteś pewien, że rzutowanie jest bezpieczne i nie ma ryzyka utraty danych.
- Używaj rzutowania dynamicznego tylko wtedy, gdy masz pewność, że rzutowanie jest bezpieczne i gdy potrzebujesz dynamicznie określić typ obiektu.
- Unikaj rzutowania reinterpretacyjnego, chyba że wiesz, co robisz i masz dobre powody, aby go użyć.
- **Pamiętaj, że nadużywanie rzutowania typów może prowadzić do nieprzewidywalnego zachowania i błędów, dlatego zawsze staraj się unikać nadużywania rzutowania.**

Rzutowanie w C#

- Rzutowanie w C# działa podobnie jak w C++.
- C# również obsługuje rzutowanie niejawne i jawne.
- Rzutowanie niejawne zachodzi wtedy, gdy konwersja jest bezpieczna i nie powoduje utraty danych.
- Rzutowanie jawne zachodzi wtedy, gdy konwersja jest potencjalnie niebezpieczna i może powodować utratę danych.
- C# istnieje również operator `as`, który wykonuje rzutowanie w dół z zachowaniem bezpieczeństwa typów. Jeśli rzutowanie się nie powiedzie, zwracany jest null.
- Operator `is` pozwala na sprawdzenie, czy dany obiekt należy do określonego typu.
- Operator `dynamic` pozwala na rzutowanie dynamiczne podobnie jak w C++.

Rzutowanie w C# - konwersja jawna

```
public class Dog
{
    public string Name { get; set; }
    public override string ToString()
    {
        return Name;
    }
    public static explicit operator Dog(string name)
    { return new Dog { Name = name }; }
}
static void Main(string[] args)
{
    Dog dog = new Dog();
    dog = (Dog)"Rex";
    Console.WriteLine(dog);
    Console.ReadLine();
}
```

Rzutowanie w C# - konwersja niejawna

```
public class Dog
{
    public string Name { get; set; }
    public override string ToString()
    {
        return Name;
    }
    public static implicit operator Dog(string name)
    { return new Dog { Name = name }; }
}
static void Main(string[] args)
{
    Dog dog = new Dog();
    dog = "Rex";
    Console.WriteLine(dog);
    Console.ReadLine();
}
```

Podsumowanie

W programowaniu obiektowym C++ rzutowanie typów jest bardzo ważnym elementem, który pozwala na przekształcanie jednego typu na inny w sposób bezpieczny i kontrolowany. Podsumowując, warto zapamiętać kilka ważnych punktów:

- rzutowanie niejawne i jawne służą do przekształcania typów prostych i klasowych,
- rzutowanie statyczne pozwala na bezpieczne przekształcanie typów z hierarchii dziedziczenia,
- rzutowanie reinterpretacyjne pozwala na rzutowanie wskaźników między typami niezwiązanymi,
- rzutowanie dynamiczne pozwala na przekształcanie wskaźników w hierarchii dziedziczenia w czasie wykonania,
- rzutowanie typów powinno być stosowane z umiarem i zgodnie z zasadami poprawnego programowania,
- w C++ istnieje wiele zaawansowanych zastosowań rzutowania typów, które mogą być przydatne w specyficznych sytuacjach.