Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 3

22 października 2020

treść dzisiejszego wykładu

- definiowanie własnych typów
  - metody
  - konstruktry
  - destrutkor
  - atrybuty

Definiowanie własnego typu danych

```
class TemperatureCelsius
{
public:
    TemperatureCelsius(float value);
    TemperatureCelsius(const TemperatureKelvin&);
    TemperatureCelsius(const TemperatureFahrenheit&);
    float value() const;
    static char symbol();

private:
    float value_;
};
```

Każdą klasę wyróżniają trzy cechy:

- nazwa, identyfikująca jednoznacznie dany typ;
- metody, określające w jaki sposób zachowują się (działają) obiekty danego typu;
- pola (atrybuty), odpowiadające za przechowanie stanu.

Definiowanie własnego typu danych

klasy i struktur

W powyższym przykładzie słowo kluczowe `class` może być bez zmiany kontekstu zastąpione słowem kluczowym `struct`.
W języku C++ jedną z niewielu różnic pomiędzy strukturą a klasą jest domyślny specyfikator dla metod pól znajdujących się bezpośrednio pod nazwą klasy:

- `struct` - publiczny;
- `class` - prywatny;

Aby unikać niejednoznaczności w dobrym stylu jest użyć odpowiednich słów kluczowych niezależnie od domyślnego parametru.

Definiowanie typów za pomocą słowa kluczowego `struct` zwyczajowo zarezerwowane jest dla prostych typów, dla których nie definiuje się żadnych metod, o publicznym zakresie wszystkich pól.

Definiowanie własnego typu danych

nazwy typów

- Nazwa klasy powinna jednoznacznie wskazywać na zamiar z jakim została utworzona.
- Nazwa klasy powinna informować zarówno o swojej odpowiedzialności, jak rówież o tym jak daną odpowiedzialność realizuje.
- Zbyt szczegółowe nazwy (tj. często zbyt długie) utrudniają analizę kodu.
- Zbyt ogólne nazwy (np. niezrozumiałe skróty) również utrudniają analizę kodu oraz wymuszają przeglądanie implementacji.
- Wskazanie odpowiedniej nazwy dla typu nie jest zadaniem prostym.

Zakres dostępu do elementów i metod klasy

publiczny

- słowo kluczowe: `public`;
- możliwe jest wywoływanie metod oraz bezpośredni dostęp do pól klasy poza samą klasą;
- zbyt szeroka zakres publiczny przeczy idei hermetyzacji;
- publiczne powinny być jedynie te metody, które są niezbędne to realizacji odpowiedzialności obiektu;

prywatny

- słowo kluczowe: `private`;
- zabronione jest wywoływanie metod oraz dostęp do pól spoza klasy;
- jedynie inne metody tej klasy mogą wywoływać metody prywatne i modyfikować prywatne (niestałe) pola klasy;

Zakres dostępu do elementów i metod klasy

Wytyczne odnoście zakresu dostępu

- Należy starać się projektować własne typy tak, by większość metod była publicznych.
- Jeżeli klasa nie jest strukturą (rekordem danych) jej pola powinny być deklarowane jako prywatne.
- Metody prywatne często są czysto pomocnicze; nie zawierają więcej niż jednej lub dwóch linii kodu.
- Jeżeli metoda prywatna wykonuje więcej operacji, prawdopodobnie jej kod należy wydzielić do osobnej klasy i użyć w danym miejscu.

Metody klasy

```
TemperatureCelsius(float value);
TemperatureCelsius(const TemperatureKelvin&);
TemperatureCelsius(const TemperatureFahrenheit&);
float value() const;
static char symbol();
```

- Reprezentują czynności jakie mogą być wykonane na obiektach.
- Mogą być prywatne i publiczne.
- Jedna klasa może mieć kilka metod o tej samej nazwie. Muszą się one jednak różnić typem lub ilością przyjmowanych argumentów - polimorfizm statyczny.
- Klasa może mieć dowolną ilość metod.
- Zbyt duża ilość metod sugeruje, że klasa ma zbyt wiele odpowiedzialności!

Metody klasy

```
public:
    float value() const;
    static char symbol();
```

- Mają dostęp do wszystkich pól prywatnych i publicznych danego obiektu.
- Mogą wywoływać wszystkie metody prywatne i publiczne niebędące konstruktorami.
- Deklaracja znajduje się w pliku nagłówkowym - definicja klasy (.hpp)..
- Implementacja powinna być ukryta i znajdować się w pliku implementacji (.cc lub .cpp).

Metody klasy

```
public:
    TemperatureCelsius(float value);
    TemperatureCelsius(const TemperatureKelvin&);
    TemperatureCelsius(const TemperatureFahrenheit&);

    float value() const;
    static char symbol();

    void print(const File&) const;
    void print(const File&);
    void print(const Screen&) const;
```

Jedna klasa może mieć kilka metod o tej samej nazwie. Muszą się one jednak różnić:

- ilością argumentów lub
- typem argumentów lub
- kwantyfikatorem `const`.

Niemniej zbyt duża ilość metod sugeruje, że klasa ma zbyt wiele odpowiedzialności! (*Patrz przykład*)

Metody specjalne - konstruktory

```
public:
    TemperatureCelsius(float value);
    TemperatureCelsius(const TemperatureKelvin&);
    TemperatureCelsius(const TemperatureFahrenheit&);
```

Konstruktor

- Wywoływane są w momencie tworzenia obiektu.
- Służy do poprawnej inicjalizacji atrybutów danego obiektu.
- Nazwa identyczna z nazwą klasy.
- Brak zwracanego typu.
- Tak samo jak inne metody mogą być przeciążane.
- Mogą być metodami publicznymi i prywatnymi - do czego może służyć prywatny konstruktor?
- Można wyodrębnić kilka rodzajów konstruktorów: domyślny (bezparametrowy), zwykły, kopiujący oraz przenoszący.

Ogólne zasady dotyczące tworzenia metod (ew. funkcji)

- Każda metoda powinna wykonywać (być odpowiedzialną za) jedną i tylko jedną czynność.
- Nazwa metody powinna dobrze odzwierciedlać to co robi.
- Należy korzystać z nazw opisowych.
- Należy starać się tworzyć funkcje tak małe jak to możliwe.
- W danej metodzie powininie być tylko jeden poziom abstrakcji.
- Należy starać się ograniczać liczbę argumentów metody.
- Należy dobierać dobre (opisowe) nazwy argumentów i zmiennych lokalnych.
- Należy unikać argumentów wyjściowych (przekazywanych przez wskaźnik bądź referencję).
- Dobrze jest gdy metody niższego poziomu znajdują się w kodzie niżej niż funkcje wyższego poziomu.
- Nie należy powtarzać bloków kodu (ctrl+c, ctrl+v).

Metody specjalne - inicjalizacja pól w konstruktorze

temperature_celsius.cpp

```
TemperatureCelsius::TemperatureCelsius(float value)
{
// w tym momencie obiekt jest juz utworzony
// a pole value_ ma przypadkowa wartosc
    value_ = value_;
}
```

temperature_celsius.cpp

```
TemperatureCelsius::TemperatureCelsius(float value)
: value_(value)
{
// w tym momencie obiekt jest rowniez juz utworzony
// a pole value_ ma wartosc taka jak argument
}
```

W związku z powyższym, który sposób inicjalizowania pól klasy jest lepszy ?

Metody specjalne - tworzenie obiektów

temperature.cpp

```
int main()
{
    std::cout << "program - w3\n";

    TemperatureCelsius temperature_1(15);

    TemperatureCelsius temperature_2 =TemperatureCelsius (20);

    auto temperature_3 =TemperatureCelsius (25);

    TemperatureKelvin temperature_k(300);

    auto temperature_4 =TemperatureCelsius (temperature_k);

    return 0;
}
```

Metody specjalne - konstruktor domyślny

Konstruktor domyślny służy do tworzenia obiektów, których stan początkowy jest ustalony i (co ważne) poprawny. Konstruktor domyślny jest jednym ze specjalnych rodzajów konstruktora. Jeżeli w klasie nie zdefiniowano żadnego innego konstruktora, kompilator, o ile to możliwe, wygeneruje domyśny konstruktor i wypełni pola ich wartościami domyślnymi. W każdym innym przypadku taki konstruktor musi być napisany przez programistę.
Istnieją trzy sposoby nadania atrybutom klasy wartości domyślnych.

Nie wszystie typy mają sensowną wartość domyślną. Typy wbudowane mogą mieć nieokreślnoną wartość domyślną.

Konstruktor domyślny nie może być wygenerowany przez kompilator, jeżeli którakolwiek ze składowych nie posiada takowego konstruktora. Właściwość ta jest przechodnia.

Metody specjalne - konstruktor domyślny

Zdefiniowanie konstruktora domyślnego.

temperature_celcius.hpp

```
class TemeperatureCelcius
{
public:
    TemeperatureCelcius();
    /* ... */
private:
    float value_;
};
```

temperature_celcius.cpp

```
TemperatureCelsius::TemperatureCelsius()
: value_(0)
{
}
```

Metody specjalne - konstruktor domyślny

Przypisanie wartości domyślnej w ciele konstruktora. Jeżeli klasa ma inne konstruktry przy pomocy słowa kluczowego `default` możliwe jest wskazanie aby kompilator sam wygenerował konstruktor domyślny.

temperature_celcius.hpp

```
class TemeperatureCelcius
{
public:
    TemeperatureCelcius() = default;
    /* ... */
private:
    float value_ = 0;
};
```

temperature_celcius.cpp

Metody specjalne - konstruktor domyślny

Dodanie wartości domyślnej argumentu konstruktora parametrycznego.

temperature_celcius.hpp

```
class TemeperatureCelcius
{
public:
    TemeperatureCelcius(float value = 0);
    /* ... */
private:
    float value_;
};
```

temperature_celcius.cpp

```
TemperatureCelsius::TemperatureCelsius(float value)
: value_(value)
{}
```

Metody specjalne - tworzenie obiektów

temperature.cpp

```
int main()
{
    std::cout << "program - w3\n";

    TemperatureCelsius temperature_1;

    // TemperatureCelsius temperature_2(); UWAGA BLAD

    auto temperature_3 = TemperatureCelsius();

    return 0;
}
```

Metody specjalne - konstruktor kopiujący

Obiekt, może być skonstruowany na podstawie innej instancji danej klasy. W tym celu wykorzystywany jest konstruktor kopiujący. Tworzona kopia nie musi być (i często nie jest) idealną kopią obiektu przekazanego jako argument.

Kontruktor kopiujący wywoływany jest zawsze, gdy niezbędne jest wykonanie kopii danego obiektu:

- w przypadku inicjalizacja jednego obiektu innym;
- w przypadku przekazywanie obiektów do funkcji lub metody (nie poprzez referencję i nie poprzez wskaźnik);
- w przypadku zwracania obiektów z funkcji lub metody;

Powyższe przykłady są jedynie wskazówką gdzie mogą być wykonywane kopie obiektów. Kompilator może (a od c++17 musi) tam, gdzie nie ma to wpływu na działanie programu, może zignorować fakt wykonywania kopii.

Metody specjalne - konstruktor kopiujący

temperature_celcius.hpp

```
class TemeperatureCelcius
{
public:
    /* ... */
    TemeperatureCelcius(const TemeperatureCelcius&);
    /* ... */
};
```

temperature_celcius.cpp

```
 TemeperatureCelcius::TemeperatureCelcius(
    const TemeperatureCelcius& temp)
: value_(temp.value_)
{
}
```

Metody specjalne - konstruktor kopiujący

O ile jest to możliwe, kompilator sam wygeneruje domyślny konstruktor kopiujący.

W jakiej sytuacji kompilator nie będzie mógł wygenerować konstruktora kopiującego?

Podobnie jak w przypadku konstruktora domyślnego, możliwe jest również wskazanie kompilatorowi by samodzielnie wygenerował konstruktor kopiujący.

```
TemeperatureCelcius(const TemeperatureCelcius&) = default;
```

Metody specjalne - konstruktor kopiujący

Jawne tworzenie kopii instancji

```
TemeperatureCelcius t1(15);
TemeperatureCelcius t2(t1);
TemeperatureCelcius t3 = t1;
```

Przekazywanie argumentu do funkcji lub metody

```
TemeperatureCelcius temp;
screen.display(temp);
```

Zwracanie wartości przez funkcję lub metodę

```
auto temp = device.measureTemperature();
```

Metody specjalne - konstruktor przenoszący

Obiekt, może być również skonstruowany na podstawie instancji, która z jakiegoś powodu nie będzie wykorzystywana bądź za chwilę przestanie istnieć. W tym celu wykorzystywany jest konstruktor przenoszący.

Semantyka przenoszenia jest zaawansowanym mechanizmem języka C++ wprowadzonym wraz ze standardem c++11 w celu unikania niepotrzebnego kopiowania dużych obiektów.

Semantyka przenoszenia związana jest z tzw. prawą referencją. Kompilator niejako "rozpoznaje" obiekty, które za chwilę przestaną istnieć. Dzięki temu możliwe jest przejęcie (przeniesienie) własności danego obiektu i przekazanie jej innemu (bez zbędnej alokacji i dealokacji zasobów).

Semantyka przenoszenia zostanie szczegółowo omówiona na późniejszych wykładach.

Metody specjalne - konstruktor przenoszący

temperature_celcius.hpp

```
class TemeperatureCelcius
{
public:
    /* ... */
    TemeperatureCelcius(TemeperatureCelcius&&);
    /* ... */
};
```

temperature_celcius.cpp

```
 TemeperatureCelcius::TemeperatureCelcius(
    TemeperatureCelcius&& temp)
: value_(std::move(temp.value_))
{
}
```

Metody specjalne - destruktor

Podobnie jak konstruktory służą do poprawnego tworzenia obiektów, tak destruktor sluży do ich poprawnego niszczenia. Destrutkor służy do wykonania niezbędnych czynności przed zniszczeniem obiektu i zwolnieniem pamięci.

Kiedy niszczony jest obiekt. W jaki sposób można źle zniszczyć obiekt ?

- Każda klasa może mieć tylko jeden destruktor.
- Destruktor wywoływany jest automatycznie gdy niszczony jest obiekt.
- W przypadku tej metody nie podaje się typu zwracanego wyniku
- Zazwyczaj umieszcza się ją w części publicznej definicji klasy.
- Destruktor można "ręcznie" wywołać w z dowolnego miejsca w kodzie.
- Możliwe jest utworzenie implementacji destruktora przy pomocy słowa kluczegowe `default`.

Metody specjalne - destruktor

temperature_celcius.hpp

```
class TemeperatureCelcius
{
public:
    /* ... */
    ~TemeperatureCelcius();
    /* ... */
};
```

temperature_celcius.cpp

```
TemeperatureCelcius::~TemeperatureCelcius()
{
}
```

Metody specjalne - destruktor

temperature_celcius.hpp

```
class TemeperatureCelcius
{
public:
    /* ... */
    ~TemeperatureCelcius() = default;
    /* ... */
};
```

temperature_celcius.cpp

Uniemożliwianie konstruowania obiektów w określony sposób

Usuwanie metod

```
private:
    TemeperatureCelcius(const Time& t) {};
```

lub

```
public:
    TemeperatureCelcius(const Time& t) = delete;
```

Dlaczego programista miałby uniemożliwić kopiowania obiektów?

- Usunięte mogą być również inne konstruktory i metody.
- Programista przekazuje swoje intencje użytkownikowi.
- Czasami inne klasy mają dostęp do pól i metod prywatnych danej klasy.

Definiowanie własnego typu danych - atrybuty klasy

```
private:
    float value_ = 0;
```

- Reprezentują stan wewnętrzny obiektu.
- Jeśli tylko to możliwe powinny pozostawać prywatne.
- Mogą być to typy podstawowe lub inne typy złożone.
- Pola mogą być inicjalizowane wartością lub innym obiektem.
- Klasa może mieć dowolną ilość pól.
- Zbyt duża ilość pól sugeruje, że klasa ma zbyt wiele odpowiedzialności!
- Obiekty może też nie posiadać żadnych atrybutów.