Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 2

10 października 2019

Treść dzisiejszego wykładu

- przykład
- zasada pojedynczej odpowiedzialności na poziomie funkcjonalnym
- podstawowa obsługa wejścia i wyjścia
- tworzenie i używanie nowych typów danych
- zakresy dostępu do elementów klasy
- pola klasy
- metod klasy
- konstrukcja obiektów
- niszczenie obiektów
- przestrzenie nazw

ogólne zasady dotyczące tworzenia funkcji (metod)

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

podstawowa obsługa wejścia wyjścia

```
#include <iostream>
```

Obiekty globalny obsługujące strumienie

- std::cout - standardowe wyjście

```
std::cout << "standardowe wyjscie" << std::endl;
```

- std::cerr - standardowe wyjście dla błędów

```
std::cerr << "standardowe wyjscie bledow" << std::endl;
```

- std::clog - standardowe wyjście dla logów

```
std::clog << "standardowe wyjscie log" << std::endl;
```

- std::cin - standardowe wejście

```
int A;
std::cin >> A;
```

obiekty typu string

Język C++ udostępnia wygodny dla użytkownika typ danych reprezentujący ciągi znaków - std::string.

```
#include <string>

int main()
{
std::string s1 = "John␣Smith";
std::string s2 = "Ann␣Johnson";

std::cout << s1 << std::endl;
std::cout << s1 << "␣and␣" << s2 << std::endl;

return 0;
}
```

Więcej na :
http://www.cplusplus.com/reference/string/string/

Tworzenie i używanie nowego typu danych

W jaki sposób możemy przechowywać informację o czasie?

```
class Time
{
public:
    short h_;
    short m_;
    double s_;

    double timeDifference(const Time t);
};

double Time::timeDifference(const Time t)
{
    return ((h_ - t.h_) * 60 + m_ - t.m_) * 60 + s_ - t.s_;
}
```

Czy klasa potrzebuje dodatkowych metod? Czy są potrzebne inne pola klasy?

Tworzenie i używanie nowego typu danych

Jak wykorzystać przygotowaną klasę?

```
#include <iostream>
#include "Time.hpp"

int main(void){
Time t1 = { 14, 15, 0 };
Time t2 = { 16,  0, 0 };

std::cout << "Wyklad␣trwa␣";
std::cout << t2.timeDifference(t1);
std::cout << "␣sekund." << std::endl;
}
```

std::cout jest obiektem globalnym dołączanym wraz z biblioteką iostream. std::cout obsługuje standardowe wyjście. « oznacza specjalną metodę klasy std::ostream, której instancją jest obiekt std::cout. Metoda ta wypisuje na ekran informacje o obiektach podstawowych.

Tworzenie i używanie nowego typu danych

W jaki sposób możemy przechowywać informację od czasie?

```
class Time
{
public:
    Time(short h, short m, double s);
    Time(double s);
    Time(const Time& t);

    double timeDifference(const Time& t);
private:
    double s_;
};
```

W jaki sposób w C++ realizowane są abstrakcja, hermetyzacja oraz polimorfizm (statyczny)?

Zakres dostępu do elementów i metod klasy

publiczny

- słowo kluczowe: `public`
- możliwe jest wywoływanie metod oraz modyfikowanie pól klasy poza klasą
- zbyt szeroka zakres publiczny przeczy idei hermetyzacji
- publiczne powinny być jedynie te pola i metody które są niezbędne to realizacji odpowiedzialności obiektu

prywatny

- słowo kluczowe: `private`
- zabronione jest wywoływanie metod oraz modyfikowanie pól poza klasą
- jedynie inne metody tej klasy mogą wywoływać metody prywatne i modyfikować prywatne pola klasy

Pola klasy

```
private:
    double s_ = 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!

Metody klasy

```
public:
    Time(short h, short m, double s);
    Time(double s);
    Time(const Time& t);

    double timeDifference(Time t);
    void addSeconds(double s);
    void addMinutes(int m);
    void addHours(int h);
```

- reprezentują czynności jakie mogą być wykonane na obiektach
- mogą być prywatne i publiczne
- jedna klasa może mieć kilka metod o tej samej nazwie, jednak muszą się one 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:
    double timeDifference(Time t);
```

- 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
- implementacja powinna być ikryta i znajdować się w pliku implementacji

Metody klasy

```
public:
    Time(short h, short m, double s);
    Time(const Hours& h, const Minutes& m, Const Seconds& s);
    Time(double s);
    Time(const Seconds& s);
    Time(const Time& t);

    double timeDifference(const Time& t);
    void add(const Seconds& s);
    void add(const Minutes& m);
    void add(const Hours& h);
```

- jedna klasa może mieć kilka metod o tej samej nazwie, jednak muszą się one różnić typem lub ilością przyjmowanych argumentów - polimorfizm statyczny
- (niemniej) zbyt duża ilość metod sugeruje, że klasa ma zbyt wiele odpowiedzialności!

Metody specjalne - konstruktory

```
public:
    Time(short h, short m, double s);
    Time(double s);
    Time(const Time& t);
```

- tak samo jak inne metody mogą być przeciążane,
- wywoływane są w momencie tworzenia obiektu / służy do inicjalizowania obiektu
- brak zwracanego typu
- nazwa identyczna z nazwą klasy
- mogą być metodami publicznymi i prywatnymi - do czego może służyć prywatny konstruktor?
- można wyodrębnić kilka rodzajów konstruktorów: domyślny (bezparametrowy), zwykły, kopiujący oraz przenoszący

Metody specjalne - konstruktor zwykły

Który sposób inicjowania zmiennych jest lepszy?

```
public:
    Time(short h, short m, double s)
    {
        s_ = (h*60 + m)*60+s;
    }
```

```
public:
    Time(short h, short m, double s)
        :  s_((h*60 + m)*60+s)
    {
    }
```

Metody specjalne - konstruktor domyślny

```
public:
    Time(void) : s_(0) {}
```

```
class Time
{
    double s_ = 0;
public:
    Time(void) {}
};
```

- brak jakiegokolwiek konstruktora spowoduje, że kompilator wygeneruje konstruktor domyślny postaci:

  ```
  Time(void) {}
  ```

- w innym przypadku konstruktor taki musi napisać programista:

  ```
  Time(void) = default;
  ```

Metody specjalne - konstruktor kopiujący

```
public:
    Time(const Time&t) : s_(t.s_) {}
```

- służy do poprawnego kopiowania obiektów
  - inicjalizacja jednego obiektu innym

    ```
    Time t1;
    Time t2(t1);
    Time t3 = t1;
    ```

  - przekazywanie obiektów do funkcji/metody

    ```
    void setTime(Time t) { /*...*/}
    ```

  - zwracanie obiektów z funkcji/metody

    ```
    Time getCurrentTime(void) { Time t; /*...*/ return t; }
    ```

- może być wygenerowany przez kompilator
- czasami konieczne jest wskazanie kompilatorowi, że ma wygenerować domyślny konstruktor kopiujący

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

Metody specjalne - konstruktor przenoszący

Semantyka przenoszenia zostanie szczegółowo omówiona na późniejszych wykładach.

C++11

```
public:
    Time( Time&&t) : s_(std::move(t.s_)) {}
```

- służy do przeniesienia danych z innego obiektu, zostawiając go w stanie nieprawidłowym
  - inicjalizacja jednego obiektu innym

    ```
    Time t1;
    Time t2 = std::move(t1);
    ```

  - przekazywanie obiektów do funkcji/metody

    ```
    setTime(std::move(t2));
    ```

  - zwracanie obiektów z funkcji/metody

    ```
    return t;
    ```

Metody specjalne - destruktor

```
public:
    ~Time();
```

- każda klasa może mieć tylko jeden destruktor
- destruktor wywoływany jest automatycznie gdy niszczony jest obiekt
- służy do wykonania niezbędnych czynności przed likwidacją obiektu
- w przypadku tej metody nie podaje się typu zwracanego wyniku
- zazwyczaj umieszcza się ją w części publicznej definicji klasy
- destruktor można wywołać w dowolnym momencie

Uniemożliwianie konstruowania obiektów w określony sposób

Dlaczego programista miałby uniemożliwić kopiowania obiektów?

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

C++11

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

- 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

przestrzeń nazw

```
class First {};
namespace A {
    class First {};
    class Second {};
    void fun(void){}
}

int main(){
    First F1;
    A::First F2;
    A::Second S1;
    //Second S2; //niezdefiniony symbol
    A::fun();

    using namespace A;
    Second S3;
    //First F3; //niejednoznaczosc symboli
    fun();

    return 0;
}
```

przestrzeń nazw

- mechanizm wprowadzony w celu uniknięcia konfliktów nazw mogących wystąpić w przestrzeni globalnej
- mogą być zagnieżdżone
- mogą być definiowane rozłącznie, np. w wielu plikach
- mechanizm ułatwiający korzystanie z przestrzeni nazw:

```
using namespace nazwa_przestrzeni;
```

- biblioteka standardowa języka C++ posiada jedną przestrzeń nazw - **std**