Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 4

29 października 2020

treść dzisiejszego wykładu

- przeciążanie (przeładowywanie) nazw funkcji i metod - przypomnienie
- projektowanie obiektowe - zasada pojedynczej odpowiedzialności
- funkcje z parametrami domniemanymi
- przekazywanie obiektów do funkcji
- zwracanie obiektów z funkcji
- specjalny wskaźnik `this`
- modyfikatory `const` oraz `mutable`
- pola i metody statyczne - słowo kluczowe `static`
- zarządzanie pamięcią - operatory `new` oraz `delete`

przeładowywanie nazw funkcji

Przeciążanie (przeładowywanie, ang. *overloading*) pozwala na tworzeniu wielu funkcji lub metod klasy o tej samej nazwie różniących się między sobą liczbą lub typem argumentów wywołania.
Przeciążanie nazw funkcji można zaliczyć do elementów polimorfizmu.

Przeciążanie dotyczy jedynie argumentów funkcji lub metody. Nie można przeciążać ze względu na typ argumentu zwracanego.

przeładowywanie nazw funkcji

```
int max(int a, int b) { return a > b ? a : b; }
double max(double a, double b) { return a > b ? a : b; }
Complex max(Complex a, Complex b) {
    return compare(a,b) ? a : b;
}
```

Która z funkcji zostanie wywołana?

```
int main(void){
max(1,2);
max(1.0, 2.0)
max({1,2}, {3,4});
return 0;
}
```

Jak powinna wyglądać klasa `Complex`?

przeładowywanie nazw funkcji

```
struct Complex
{
    double x, y;
};

bool compare(const Complex& a, const Complex& b)
{
    return a.x * a.x + a.y * a.y > b.x * b.x + b.y * b.y;
}
```

Dlaczego użyto słowa kluczowego `struct` ?
Jakie są inne możliwe implementacje tej funkcji?

przeładowywanie nazw funkcji

Uwaga:

Przeładowywanie nazw funkcji nie działa wtedy gry różnice występują jedynie w typie zwracanym przez funkcję.

Napisanie poniższego kodu spowoduje błąd kompilacji.

Błąd

```
int max(int a, int b) { return a > b ? a : b; }
double max(int a, int b) { return a > b ? a : b; }
```

przeładowywanie nazw funkcji

Uwaga:

Przeładowywanie nazw funkcji nie działa również wtedy gdy jednej z funkcji obiekty przekazywane są przez wartość, drugiej zaś przez referencję.

Błąd

```
void f(F f1) {}
void f(F& f1) {}
```

Ok

```
void f(F f1) {}
void f(F* f1) {}
```

przeładowywanie nazw metod

Podobnie jak funkcje, metody klasy również mogą być przeładowywane.

file_screen_printer.hpp

```
#inculde "text_file.hpp"
#inculde "csv_file.hpp"

class RandomAccessTextFile;
class LightRandomAccessTextFile;

class TextFileScreenPrinter
{
public:
    void print(const RandomAccessTextFile&) const;
    void print(const LightRandomAccessTextFile&) const;
};
```

Jaką funkcję pełni obiekt typu `TextFileScreenPrinter`? Jakie może być jego zastosowanie?

przeładowywanie nazw metod

main.cc

```
int main()
{
    LightRandomAccessTextFile file_1("../test_file.txt");
    RandomAccessTextFile file_2("../test_file.txt");

    TextFileScreenPrinter screen_printer;

    screen_printer.print(file_1);
    screen_printer.print(file_2);

    return 0;
}
```

przeładowywanie nazw metod

main.cc

```
#include "FileFactory.hpp"

int main()
{
    auto file = FileFactory().create("../test_file.txt");

    TextFileScreenPrinter screen_printer;

    screen_printer.print(file);

    return 0;
}
```

zmiany, zmiany, zmiany...

Zmiany w oprogramowaniu są nieuniknione, nawet na późnym etapie jego powstawania.
Dobrze zaprojektowane oprogramowanie pozwala (względnie tanio) wprowadzać istotne zmiany zmiany nawet w przypadku potencjalnie rozbudowanych systemów.

Istnieje szereg technik pozwalających skutecznie przezwyciężać trudności we wprowadzaniu zmian do oprogramowania. Są to między innymi:

- Stosowanie technik projektowania oprogramowania uodparniające je na częste wprowadzanie zmian.
- Częste dostarczanie działającego oprogramowania.
- Testowanie oprogramowania na każdym etapie jego powstawania. Oprogramowanie sterowane (napędzane) testami, .ang *Test Driven Developmnet*.

zasada pojedynczej odpowiedzialności

- Powinien istnieć jeden powód do zmiany klasy (zmiany w klasie).
- Każda zmiana w projekcie powinna sutkowac zmianami w minimalnej ilości klas.
- Zachowywanie zasady pojedynczej odpowiedzialności sprzyja swobodnemu dokonywaniu zmian w projekcie.
- Przyjęcie przez klasę zbyt dużej ilości odpowiedzialności (> 1) zazwyczaj prowadzi do sprzężeń co utrudnia dokonywanie zmian.
- Zmiany związane z jedną odpowiedzialnością mogą wprowadzać (i zazwyczaj tak jest) niepożądane zmiany w innych odpowiedzialnościach.

ogólne zasady projektowania klas - powtórzenie

- Klasa powinna robić jedną i tylko jedną rzecz, i powinna to robić w sposób możliwie prosty.
- Klasa powinna mieć dobrze zdefiniowaną odpowiedzialność.
- Nazwa klasy powinna dobrze odzwierciedlać jej odpowiedzialność.
- Klasa powinna mieć mała liczbę zmiennych instancyjnych.
- Metody powinny manipulować (najlepiej wszystkimi) zmiennymi instancyjnymi.
- Klasy powinny być stosunkowo małe i proste aby łatwo można było dokonywać z nich zmian.
- Zmienne instancyjne oraz metody użytkowe powinny zostać zadeklarowane jako prywatne (hermetyzacja).
- Rzadko istnieje dobry powód do upubliczniania tych składowych.

przykład, c.d.

Klasa `TextFileScreenPrinter` z przykłądu nie ma dobrze zdefiniowanej odpowiedzialności. Istnieją dwa powody do zmiany jej implementacji. Jeżeli zmianie ulegnie implementacja, klasy `RandomAccessTextFile` lub `LightRandomAccessTextFile` konieczna może się okazać ingerencja w kod klasy `TextFileScreenPrinter`.

W jaki sposób przeciwdziałać tym zależnościom?

text_file_screen_printer.hpp

```
class TextFileScreenPrinter
{
public:
    void print(const RandomAccessTextFile&) const;
    void print(const LightRandomAccessTextFile&) const;
};
```

modyfikatory const oraz mutable

- deklaracja pól klas

```
struct F {
    F(void) : object_no_(0){ }
private:
    const unsigned int object_no_;
};
```

- definicja i deklaracja obiektów

```
const F f;
```

- deklaracja funkcji niestatycznych klas

```
struct F {
    F copy(void) const { return *this; }
};
```

- deklaracja parametrów funkcji i metod

```
struct F {
    F(const F& f) {}    };
```

modyfikatory `const` oraz `mutable`

Metody niestatyczne klas z kwalifikatorem `const` nie modyfikują obiektu, na rzecz którego zostały wywołane.

Tylko metody z kwalifikatorem `const` mogą być wywoływane na rzecz stałych obiektów (referencji).

Kwalifikator `const` może być wykorzystany do przeładowywania metod klasy.

modyfikatory const oraz mutable

Kwalifikator mutable pozwala na modyfikowanie danego pola przez metody z kwalifikatorem const oraz przez bezpośredni dostęp dla obiektów stałych (jeśli taki dostęp jest możliwy).

```
struct F {
    void fun(void) const { var++; }
private:
    mutable int var = 0;
};
int main(void){
    const F f;
    f.fun();
    return 0;
}
```

modyfikatory const oraz mutable

Co powinny zwracać metody stałe (z modyfikatorem const) ?

```
class TextFileReader
{
public:
    TextFileReader(std::string path);

    std::string getLine();
    std::string getLine() const;

    bool endOfFile() const ;

    void rewind();

private:
    std::ifstream file_;
};
```

Jak powinna zostać zaimpementowana klasa powyżej ?

modyfikatory const oraz mutable

Co powinny zwracać metody stałe (z modyfikatorem const) ?

```
class Array
{
    static const size_t MAX_SIZE = 1000;
public:
    double& at(size_t index) {return array_[index];}
    ??      at(size_t index) const {return array_[index];}

    size_t size() const { return MAX_SIZE;}

    void rewind();

private:
    double array_[MAX_SIZE];
};
```

funkcje z parametrami domniemanymi

Funkcje mogą mieć również parametry wywołania o wartościach domniemanych. Wartości domniemane definiuje się w miejscu deklaracji funkcji (prototypy).

```
typ0 funkcja(    typ1 zmienna1,
                 typ2 zmienna2,
                 typ3 zmienna3 = zmiennaDomniemana3,
                 typ4 zmienna4 = zmiennaDomniemana4);
```

- parametry obowiązkowe
- parametry opcjonalne - występują **zawsze** po parametrach obowiązkowych

Czy funkcje z parametrami domniemanymi zwiększają czytelność kodu? Kiedy stosowanie parametrów domniemanych jest dobrym pomysłem?

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**