Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 3

17 października 2019

treść dzisiejszego wykładu

- przykład
- projektowanie obiektowe - zasada pojedynczej odpowiedzialności
- przeładowywanie (przeciążanie) nazw funkcji i metod
- funkcje z parametrami domniemanymi
- przekazywanie obiektów do funkcji
- zwracanie obiektów z funkcji
- specjalny wskaźnik `this`
- zaprzyjaźnianie - `friend`
- modyfikatory `const` oraz `mutable`
- zarządzanie pamięcią - operatory `new` oraz `delete`

przykład

text_file.hpp

```
class TextFile
{
public:
    TextFile() = delete;
    TextFile(const std::string& file_name);

    std::string getLine();

    void printOnScreen() const;

    ~TextFile();
private:
    bool _checkFile();
    void _open();
    void _close();
    bool _isEndOfFile();
    /*...*/
};
```

przykład

text_file.hpp

```
class TextFile
{
public:
    TextFile() = delete;
    TextFile(const std::string& file_name);

    std::string getLine();
    bool isEndOfFile();

    ~TextFile();
private:
    void _open();
    void _close();
    /*...*/
};
```

przykład

text_file_screen_printer.hpp

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

text_file_screen_printer.cc

```
#include <iostream>

void TextFileScreenPrinter::print(const TextFile& file) const
{
    while(!file.isEndOfFile())
    {
        std::cout << file.getLine() << "\n";
    }
}
```

przykład

text_file.cc

```
class TextFileChecker
{
public:
    bool check(const std::string& file_name) const;
};

TextFile::TextFile(const std::string& file_name)
    /*: ...*/
{
    /*...*/
    if (!TextFileChecker().check(file_name))
    {
        /*...*/
    }
    /*...*/
}
```

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

- 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` ma dobrze zdefiniowaną odpowiedzialność i istnieje tylko jeden powód do zmiany tej klasy. Tym powodem jest sposób w jaki ma być wyświetlana na ekranie zawartość pliku. Jeśli chcielibyśmy aby przed zawartością kolejnych wierszy wyświetlany był ich numer wystarczy zmienić implementację metody.

text_file_screen_printer.cc

```
#include <iostream>

void TextFileScreenPrinter::print(const TextFile& file) const
{
    unsigned int line_number = 0;
    while(!file.isEndOfFile())
    {
        std::cout << line_number++ << file.getLine() << "\n";
    }
}
```

przeładowywanie nazw funkcji

Przeładowywanie (przeciążanie, ang. *overloading*) pozwala na tworzenie wielu funkcji o tej samej nazwie różniących się między sobą typem parametrów wywołania.

```
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 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 operator>(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 do funkcji metody klasy również mogą być przeładowywane.

file_screen_printer.hpp

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

class FileScreenPrinter
{
public:
    void print(const TextFile&) const
    void print(const CSVFile&) const;
    /*...*/
};
```

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

przeładowywanie nazw metod

file_screen_printer.cc

```
#include "file_screen_printer.hpp"
#include "text_file_screen_printer.hpp"
#include "csv_file_screen_printer.hpp"

FileScreenPrinter::print(const TextFile& file)
{
    TextFileScreenPrinter().print(file);
}

FileScreenPrinter::print(const CSVFile& file)
{
    CSVFileScreenPrinter().print(file);
}

/*...*/
```

przeładowywanie nazw metod

main.cc

```
#include <iostream>

#inculde "text_file.hpp"
#inculde "csv_file.hpp"
#include "screen_file_printer.hpp"

int main()
{
    FileScreenPrinter printer;

    TextFile text_file("file.txt");
    CSVFile csv_file("file.csv");

    std::cout << "wypisanie plikow na ekran monitora \n";
    printer.print(text_file);
    printer.print(csv_file);

    return 0;
}
```

funkcje z parametrami domniemanymi

Funkcje mogą mieć również parametry wywołania o wartościach domniemanych - zarówno ich deklaracje (prototypy) jaki i definicje.

```
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?

sposoby przekazywania obiektów do funkcji

przez wartość

```
void funkcja(Typ1 zmienna1, Typ2 zmienna2);
```

przez adres zmienne

```
void funkcja(Typ1* w_zmienna1, Typ2* w_zmienna2);
```

przez referencję do zmiennej

```
void funkcja(Typ1& zmienna1, Typ2& w_zmienna2);
```

sposoby przekazywania obiektów do funkcji

Który ze sposobów przekazywania zmiennej do funkcji będzie najlepszy dla poszczególnych typów?

typy proste

```
char, int, float, double
```

typ zdefiniowany przez użytkownika

```
struct F {
    double data[1000];
};
```

sposoby zwracania wyniku przez funkcję

przez wartość

```
Typ0 funkcja(Typ1 zmienna1, ...) {
    Typ0 zmienna;
    /**/
    return zmienna; }
```

sposoby zwracania wyniku przez funkcję

przez wskaźnik do dynamicznie alokowanego obiektu

```
Typ0* funkcja(Typ1 zmienna1, ...) {
    auto wsk = new Typ0;
    /**/
    return wsk;
}
```

Uwaga

Zwracanie adresu dynamicznie alokowanego obiektu przy zwiększonej uwadze może czasami być stosowane, jeżeli rzeczywiście jest niezbędne. Często natomiast zwracany jest adres zmiennej globalnej. Zwracanie adresu zmiennej automatycznej jest **zawsze** złym rozwiązaniem.

sposoby zwracania wyniku przez funkcję

Metoda klasy może zwracać wskaźnik należący do tej klasy, lub adres jakiegoś innego obiektu składowego. Należy uważać, żeby obiekt nie został zniszczony wcześniej niż wskazanie na jego składową.

```
struct F {
    double* data(void) { return data_; }
private:
    double data_[1000];
};
int main(void){
    double* wsk;
    {
        F f;
        wsk = f.data();
    }
    wsk[0]; //?
    return 0;
}
```

sposoby zwracania wyniku przez funkcję

przez przekazanie adresu obiektu do wypełnienia

```
void funkcja(Typ0* zmienna0, Typ1 zmienna1, ...) {
    zmienna0->wykonajOperacje1(zmienna1, ...);
}
```

przez przekazanie obiektu do wypełnienia

```
void funkcja(Typ0& zmienna0, Typ1 zmienna1, ...) {
    zmienna0.wykonajOperacje1(zmienna1, ...);
}
```

Uwaga

Ze względu na nieczytelności stosowanie argumentów wyjściowych nie jest zalecanym sposobem zwracania wyniku przez funkcję. Konieczność stosowania tego typu konstrukcji zazwyczaj wynika ze złego projektu klasy / modułu i należy spróbować go poprawić.

specjalny wskaźnik this

Wskaźnik this wskazuje na obiekt, na rzecz którego wywołana została metoda danej klasy. Dostępny jest jedynie w zasięgu lokalnym metod niestatycznych.

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

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

zaprzyjaźnianie - `friend`

- Funkcja, która jest przyjacielem klasy, ma dostęp do wszystkich jej prywatnych i chronionych składowych.
- To klasa określa, które funkcje są jej przyjaciółmi.
- Deklaracja przyjaźni może się pojawić w dowolnej sekcji i jest poprzedzona słowem kluczowym `friend`.
- Jedna funkcja może się przyjaźnić z kilkoma klasami. Funkcją zaprzyjaźnioną może być funkcja składowa z innej klasy.
- Możemy w klasie zadeklarować przyjaźń z inną klasą, co oznacza, że każda metoda tej innej klasy jest zaprzyjaźniona z klasą pierwotną.

zaprzyjaźnianie - friend

```
class Complex{
public:
    Complex(double x, double y) : x(x), y(y) {}
    friend std::ostream& operator<<(std::ostream out&,
                                    const Complex& a);
private:
    double x = 0, y = 0;
};

std::ostream& operator<<(std::ostream out&,
                         const Complex& a) {
    out << a.x << " + j" << a.y;
    return out;
}
```

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) ?

```
    double* data_ = nullptr;
    size_t size_ = 0;
public:
    SimpleVector(size_t s)  // uwaga na rozmiar s
        : size_(s), data_(new double[s]) {}

    ~SimpleVector(void) {
        delete[] data_; } // uwaga na nullptr

    double& operator[](unsigned int i) {
        return data_[i]; } // uwaga na zakres

    ??? operator[](unsigned int i) const {
        return data_[i]; } // uwaga na zakres

    double* data(void) { return data_; }

    ??? data(void) const { return data_; }
};
```

Czy to jedyne możliwe rozwiązanie ?

```
    double* data_ = nullptr;
    size_t size_ = 0;
public:
    SimpleVector(size_t s)  // uwaga na rozmiar s
        : size_(s), data_(new double[s]) {}

    ~SimpleVector(void) {
        delete[] data_; } // uwaga na nullptr

    double& operator[](unsigned int i) {
        return data_[i]; } // uwaga na zakres

    const double& operator[](unsigned int i) const {
        return data_[i]; } // uwaga na zakres

    double* data(void) { return data_; }

    const double* data(void) const { return data_; }
};
```

zarządzanie pamięcią - zmienne automatyczne

```
Typ zmienna;
```

- dotyczą kontekstu
- po opuszczeniu danego kontekstu są usuwane
- stosunkowo łatwo jest nimi zarządzać
- umieszczane są na stosie

```
struct F {
    F(void) { std::cout << "F␣c-tor" << std::endl; }
    ~F(void) { std::cout << "F␣d-tor" << std::endl; }
};

int main(void){
    F f1;
    return 0;
}
```

zarządzanie pamięcią - zmienne automatyczne

```
struct B {
    B(void) { std::cout << "B c-tor" << std::endl; }
    ~B(void) { std::cout << "B d-tor" << std::endl; }
};
```

```
int main(void){
    F f1;
    B b1;
    {
        F f2;
        {
            B b2;
        }
    }
    return 0;
}
```

Ile będzie wywołań konstruktorów i destruktorów? W jakiej kolejności będą się one wywoływane?

zarządzanie pamięcią - operatory new oraz delete

```
Typ* w_zmienna = new Typ;
```

```
auto w_zmienna = new Typ;
```

- zmienne dynamiczne tworzone są na stercie
- nie są związane z kontekstem
- po opuszczeniu danego kontekstu usuwane są **jedynie** wskaźniki odnoszące się do danej zmiennej
- stosunkowo łatwo jest doprowadzić do wycieku pamięci
- używamy wtedy gdy nie nie da się utworzyć zmiennej automatycznej

zarządzanie pamięcią - operatory new oraz delete

Język C++ dostarcza operatory przydzielania pamięci na stercie (new) oraz jej zwalniania (delete) w wersji skalarnej oraz tablicowej

```
auto w_zmienna = new Typ(parametr1, parametr2, ...);
delete w_zmienna;

auto w_zmienna_tablicowa = new Typ[wielkoscTablicy];
delete[] w_zmienna_tablicowa;
```

- w przypadku niepowodzenia alokacji zgłaszany jest wyjątek **bad_alloc** - o wyjątkach w dalszej części wykładu
- wielkość tworzonej tablicy nie musi być znana w czasie kompilacji
- bardzo ważne aby używać poprawnego operatora zwalniającego pamięć - adekwatnego do sposoby jej alokowania
- nie można utworzyć tablicy obiektów, które nie posiadają domyślnego konstruktora

zarządzanie pamięcią - operatory new oraz delete

```
int main(void){
    auto pf1 = new F;
    auto pb1 = new B;
    {
        auto pf2 = new F;
        {
            auto pb2 = new B;
        }
    }
    return 0;
}
```

Ile będzie wywołań konstruktorów i destruktorów? W jakiej kolejności będą się one wywoływane?

Poprawiona wersja kodu.

```
int main(void){
    auto pf1 = new F;
    auto pb1 = new B;
    {
        auto pf2 = new F;
        {
            auto pb2 = new B;
            \*...*\
            delete pb2;
        }
        delete pf2;
    }

    delete pb1;
    delete pf1;
    return 0;
}
```