Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 4

8 listopada 2018

Treść dzisiejszego wykładu

- przypomnienie
- organizacja kodu w plikach
- szablon funkcji
- szablon klasy
- szablon metody
- specjalizacja szablonu

przypomnienie

Istotę programowania obiektowego stanowią obiekty i ich wzajemne relacje w trakcie działania programu.
Klasa stanowi opis tego jak obiekty są tworzone i niszczone, jakie mogą mieć stany wewnętrzne oraz w jaki sposób współpracują z innymi obiektami obecnymi w programie.

Klasa

```
class Point {
    double x, y;
    /*...*/
public:
     Point (double x, double y);
     /*...*/
};
```

Obiekt

```
int main(void)
{
    Point p1(1, 1);
    Point p2 = p1 + p1;
    /*...*/
    return 0;
}
```

przypomnienie

W języku C++ istnieje podział na trzy główne typy obiektów:

- typ zwykły,
- typ referencyjny,
- typ wskaźnikowy.

Typ referencyjny może być tak zwaną l-referencją lub r-referencją.

```
int main(void)
{
    Point p(1, 1);          // obiekt
    Point* wp = &p;         // wskaznik
    Point& rp = p;          // l-referencja
    std::move(p);           // r-referencja
    return 0;
}
```

przypomnienie

W języku C++ istnieje możliwość definiowania wolnych funkcji, tj. funkcji niezwiązanych z żadnym obiektem bądź klasą.
Funkcje te mogą przyjmować postać zwykłą bądź postać operatora.
Funkcje mogą przyjmować dowolną ilość argumentów dowolnego typu oraz zwracać tylko jedną wartość. Mogą także mieć takie same nazwy (o ile różnią się typem argumentów) - polimorfizm statyczny.

```
void print_info(const char* str)
{
    std::cout << str << std::endl;
}
```

```
Point operator+(const Point& lhs, const Point& rhs)
{
    return Point(lhs.x + rhs.x, lhs.y + rhs.y);
}
```

przypomnienie

Wszystkie metody niestatyczne są to **funkcje**, wywoływane ma rzecz danego obiektu, w który obiekt ten jest niejawnym argumentem danym przez wskaźnik `this`.
Metody mogą mieć modyfikator `const`, który informuje kompilator (użytkownika), że dana metoda nie zmienia **stanu wewnętrznego** obiektu.
Metody statyczne mogą modyfikować jedynie statyczne pola klasy, nie są związane z żadnym konkretnym obiektem danego typu.

```
class TrainBuilder
{
    Locomotive buildLocomotive(/*...*/) const;
    TrainWagon buildTrainWagon(/*...*/) const;
public:
    TrainBuilder(/*...*/);
    Train build(const TrainSpec& spec) const;
};
```

przypomnienie

Wszystkie pola (inaczej atrybuty) klasy pomijając pola statyczne (`static`) oraz modyfikowalne (`mutable`) opisują stan wewnętrzny obiektu.
Podobnie jak metody, pola również mogą mieć atrybut stałości (`const`)

```
class MemoryChunk
{
private:
    struct MemorySize { size_t size; };

    const double * memory_;
    const MemorySize size_;

public:
    MemoryChunk(const double* mem, const size_t size);
    const double* data(void) const;
    const double& operator[](size_t index) const;
};
```

organizacja kodu w plikach

Zazwyczaj klasy implementowane są w dwóch rodzajach plików

- pliki nagłówkowe (ang. *header*) z rozszerzeniem `.hpp` lub `.h`
- pliki źródłowe (ang. *source*) z rozszerzeniem `.cpp`, `.c` lub `.cc`

Plik nagłówkowy stanowi interfejs danego modułu (moduł może zawieraj więcej niż jedną klasę), wciągany (dyrektywa `#include`) do innych modułów. Wszystko co zawiera plik nagłówkowy jest widoczne dla innych użytkowników i jest niejako publiczne (nawet jeśli zadeklarowane w obszarze prywatnym - `private`)
Plik źródłowy może być kompilowany do pliku obiektowego (zawierającego kod binarny) a następnie dołączany do innych jednostek translacji w trakcie konsolidacji w celu utworzenia pliku wykonywalnego. Pliki obiektowe, biblioteki statyczne i dynamiczne nie są czytelne dla ich użytkowników - zawierają kod maszynowy.

matrix.hpp

```
#ifndef MATRIX_HPP
#define MATRIX_HPP

class Matrix{
    class Vector    {
        /* ... */
    public:
        double& operator[](unsigned int n);
        const double& operator[](unsigned int n) const;
        Vector(unsigned int n, double * data);
    };

    struct Size { unsigned int n, m; };

    double * data_;
    Size size_;
public:
    Matrix(unsigned int n, unsigned int m);
    Matrix(Matrix&& rhs);
    Matrix(const Matrix& rhs);
    ~Matrix(void);

    friend Matrix operator*(const Matrix& lhs, const Matrix& rhs);
    Vector operator[](unsigned int m);
    const Vector operator[](unsigned int m) const;
};
#endif
```

matrix.cpp

```
#include "matrix.hpp"

Matrix::Vector::Vector(unsigned int n, double * data)
    : size_{ n }, data_(data) {}

Matrix::Matrix(unsigned int n, unsigned int m)
    : size_{ n,m }, data_(new double[n*m]) {}

Matrix::Matrix(const Matrix& rhs)
    : Matrix(rhs.size_.n, rhs.size_.m){ /* ... */ }

Matrix::Matrix(Matrix&& rhs)
    : size_(rhs.size_), data_(rhs.data_){ /* ... */ }

Matrix::~Matrix(void){ /* ... */ }

double& Matrix::Vector::operator[](unsigned int n) { /* ... */ }
const double& Matrix::Vector::operator[](unsigned int n) const { /* ... */ }

Matrix operator*(const Matrix& lhs, const Matrix& rhs) { /* ... */ }

Matrix::Vector Matrix::operator[](unsigned int m) { /* ... */ }

const Matrix::Vector Matrix::operator[](unsigned int m) const
{ /* ... */ }
```

programowanie uogólnione

- jeden z paradygmatów programowania
- możliwe jest tworzenie kodu (algorytmu) bez znajomości typów danych na jakich będzie operował - o ile algorytm na to pozwala.
- możliwe jest opracowanie wzorców typów danych, które będą się zachowywać w różny sposób w zależności od lokalnej potrzeby użytkownika.

Paradygmat programowania to inaczej sposób w jaki programista patrzy na program i w jak steruje jego przepływami sterowania. W programowaniu obiektowym program traktowany jest jako współpracujące ze sobą obiekty.
Jakie są przykłady algorytmów, które są poprawne niezależnie od typu danych?

problem obliczania potęgi o wykładniku naturalnym

Potęgowanie za pomocą mnożenia

$$x^k = x \cdot x^{k-1} = \underbrace{x \cdot x \cdot x \cdot \ldots \cdot x}_{k} \tag{1}$$

Algorytm szybkiego potęgowania

$$x^n = \begin{cases} x \cdot (x^2)^{\frac{n-1}{2}}, & \text{jeśli } n \text{ jest nieparzyste} \\ (x^2)^{\frac{n}{2}}, & \text{jeśli } n \text{ jest parzyste} \end{cases} \tag{2}$$

problem obliczania potęgi o wykładniku naturalnym

```
double power(double x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}

int main(void)
{
    for( unsigned int i = 0; i < 10; ++i)
        std::cout << power(2, i) << std::endl;
    return 0;
}
```

Co musimy zrobić gdy chcemy podnosić do potęgi tylko liczby całkowite, reprezentowane na przykład przez typ `long long` ? Dlaczego chcielibyśmy móc to robić?

problem obliczania potęgi o wykładniku naturalnym

```
long long power(long long x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}
```

problem obliczania potęgi o wykładniku naturalnym

```
long long power(long long x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}
```

```
template<typename T>
T power(T x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}
```

szablony w języku C++

Szablony (ang. *template*) w języku C++ oznaczane są przy pomocy słowa kluczowego `template`. Deklaracja lub definicja funkcji (lub obiektu, metody) szablonowej poprzedza konstrukcja:

```
template<typename T>
```

lub:

```
template<class T>
```

W trakcie kompilacji, kod odpowiedniej funkcji zostaje wygenerowany na podstawie przekazanych argumentów szablonu. Każda funkcja wygenerowana w ten sposób posiada osobny kod binarny w programie i osobny adres w pamięci.

szablony funkcji

- Argumenty szablonu mogą być podawane w sposób jawny za nazwą wołanej funkcji, argument ten określa typ dla którego konkretyzowany jest szablon:

```
auto r1 = power<int>(2, 4);
```

- Jeśli kompilator na podstawie wywołania jest wstanie wydedukować typy argumentów szablonu, wtedy argumenty te mogą być pominięte w wywołaniu; Automatyczna dedukcja typów argumentów szablonu wyłącza automatyczne konwersje:

```
auto r2 = power(2, 4);
auto r3 = power(2.1, 4);
```

- Podanie argumentów w sposób jawny, z kolei umożliwia automatyczną konwersję typów:

```
auto r2 = power<double>(2, 4);
auto r3 = power<int>(2.1, 4);
```

szablony funkcji

- Automatyczna dedukcja parametrów szablonu jest możliwa jedynie wtedy, gdy parametry wywołania funkcji zależą od typów parametrów szablonu.
- Jeżeli parametr szablonu określa na przykład typ zwracany przez funkcję musi być on podany jawnie wraz z wywołaniem funkcji.
- Liczba parametrów szablonu nie jest w żaden sposób ograniczona.
- Próba konkretyzacji poprawnego szablonu w nieprawidłowymi typami argumentów, może generować błąd kompilacji

```
//auto r4 = power("napis", 4); // blad kompilacji
```

Dlaczego powyższa instrukcja jest nieprawidłowa a poniższa już tak:

```
auto r4 = power('n', 4);
```

szablony funkcji

W celu wywołania odpowiedniej funkcji kompilator postępuje według następujących kroków:

1. Poszukiwana jest funkcja o argumentach dokładnie pasujących do tych w wywołaniu
2. Poszukiwany jest wzorzec funkcji pozwalający na generację funkcji spełniającej warunek pierwszy
3. Poszukiwana jest funkcja przeciążona o pasujących argumentach
4. Zgłaszany jest błąd.

Podobnie jak w przypadku funkcji istnieje możliwość definiowania wzorców klas.

```
template<typename T>
class MemoryT
{
public:
    struct Size { size_t size; };
    class SizeExceeded {};

    MemoryT(Size size) : data_(new T[size.size]), size_(size) {}

    T& operator[](size_t i) {
        if (i < size_.size) return data_[i]; throw SizeExceeded(); }
    const T& operator[](size_t i) const {
        if (i < size_.size) return data_[i]; throw SizeExceeded(); }

    ~MemoryT(void) { if (nullptr != data_) delete[] data_; }
private:
    T* data_{ nullptr };
    Size size_{ 0 };

    MemoryT(const MemoryT&) = delete;
    MemoryT(MemoryT&&) = delete;
};
```

szablony klas

W przypadku szablonu klasy nie istnieje możliwość automatycznej dedukcji argumentów szablonu. Należy podać je w sposób jawny.

```
int main(void)
{
    size_t N = 10;
    MemoryT<double> mem1({ N });
    MemoryT<char> mem2({ N });
    // MemoryT mem3({ N }); // blad
};
```

Istnieje możliwość podania argumentów domyślnych

```
template<typename T = double>
class MemoryT {/*...*/};

int main(void)
{
    size_t N = 10;
    MemoryT mem3({ N }); // OK
};
```

Parametrami szablonów mogą być również stałe pozatypowe (np. liczby całkowite, typ wyliczeniowy)

```
template<typename T, size_t N = 10>
class MemoryT2
{
public:
    class SizeExceeded {};

    T& operator[](size_t i) {
        if (i < N) return data_[i]; throw SizeExceeded(); }
    const T& operator[](size_t i) const {
        if (i < N) return data_[i]; throw SizeExceeded(); }

private:
    T data_[N];
};
```

Uwaga. Każda nowa specjalizacja szablonu powoduje powstanie nowego typu danych, który nic nie wiem o pozostałych specjalizacjach z danej rodziny. W szczególności nie istnieje możliwość kopiowania danych pomiędzy obiektami różnych typów.

```
MemoryT2<double, 10> mem4;
mem4[0] = 0; // ...
MemoryT2<double, 10> mem5 = mem4; // OK
// MemoryT2<double, 11> mem6 = mem4; // blad
// MemoryT2<float, 10> mem7 = mem4; // blad
```

definicja metod szablonu klas poza ciałem szablonu

```
template<typename T>
class MemoryT
{
public:
    /*...*/
    MemoryT(const MemoryT&);
    MemoryT(MemoryT&&);
    /*...*/
};

template<typename T>
MemoryT<T>::MemoryT<T>(const MemoryT<T>& rhs)
    : MemoryT<T>(rhs.size_)
{
    memcpy(data_, rhs.data_, rhs.size_.size * sizeof(T));
}

template<typename T>
MemoryT<T>::MemoryT<T>(MemoryT<T>&& rhs)
    : data_(rhs.data_), size_(std::move(rhs.size_))
{
    rhs.data_ = nullptr;
    rhs.size_.size = 0;
}
```

metoda szablonowa

W języku C++ istnieje również możliwość definiowania metod szablonowych zarówno w klasach jak i wzorcach klas.

```
template<typename T>
class MemoryT {
public:
    template<typename U>
    MemoryT(const MemoryT<U>& rhs);

    friend class MemoryT;
    /*...*/
};

template<typename T>
template<typename U>
MemoryT<T>::MemoryT<T>(const MemoryT<U>& rhs)
   : MemoryT<T>({rhs.size_.size})
{
    for (size_t i = 0; i < size_.size; ++i)
        data_[i] = static_cast<T>(rhs.data_[i]);
}
```

W poniższym przypadku zdefiniowany został konstruktor pozwalający na konwersję jednego typu w drugi. Taki konstruktor pomimo, że przypomina konstruktor kopiujący jest zwykłym konstruktorem parametrycznym. Dlaczego?

metoda szablonowa

```
int main(void)
{
    MemoryT<double> mem_d({ N });
    for (int i = 0; i < N; ++i)
        mem_d[i] = i+0.1;

    MemoryT<int> mem_i(mem_d);

    for (int i = 0; i < N; ++i)
        std::cout << mem_d[i] << " " << mem_i[i] << std::endl;

    // MemoryT<std::string> mem_s = mem_d; // blad


    return 0;
}
```

Konstruktor taki jak w przykładzie pozwala na automatyczną konwersję pomiędzy różnymi typami. Jego implementacja może być kontrowersyjna, gdyż kompilator ma możliwość zastosowania automatycznej konwersji. Pozostawienie takiego konstruktora pozostaje w decyzji programisty / projektanta systemu.

metoda szablonowa

```
template<typename T>
class MemoryT {
private:
    template<typename U>
    MemoryT(const MemoryT<U>& rhs);
public:
    template<typename U>
    MemoryT<U> to(void);

    friend class MemoryT;
    /*...*/
};

template<typename T>
template<typename U>
MemoryT<U> MemoryT<T>::to(void) { return MemoryT<U>(*this); }
```

```
int main(void)
{
    MemoryT<double> mem_d({ N });
    for (int i = 0; i < N; ++i) mem_d[i] = i+0.1;

    //MemoryT<int> mem_i(mem_d);
    auto mem_i = mem_d.to<int>();
    return 0;
}
```

organizacja plików szablonów

memory_t.hpp

```
#ifndef MEMORY_T_HPP
#define MEMORY_T_HPP
template<typename T>
class MemoryT {
    /* ... */
public:
    template<typename U>
    MemoryT<U> to(void);
    /* ... */
};

#include "memory_t.tpp"
#endif
```

memory_t.tpp

```
#ifdef MEMORY_T_HPP
/* ... */
template<typename T>
template<typename U>
MemoryT<T>::MemoryT<T>(const MemoryT<U>& rhs)
    : MemoryT<T>({ rhs.size_.size }) { /* ... */ }
/* ... */
#endif
```

Jak sprawić by konwersja na napis była możliwa?

```
int main(void)
{
    MemoryT<double> mem_d({ N });
    for (int i = 0; i < N; ++i)
        mem_d[i] = i+0.1;

    auto mem_s = mem_d.to<std::string>();

    return 0;
}
```

Jak sprawić by konwersja na napis była możliwa?

```
int main(void)
{
    MemoryT<double> mem_d({ N });
    for (int i = 0; i < N; ++i)
        mem_d[i] = i+0.1;

    auto mem_s = mem_d.to<std::string>();

    return 0;
}
```

Język C++ umożliwia specjalizację szablonów częściową bądź pełną:

```
template<typename T> template<typename U>
MemoryT<U> MemoryT<T>::to(void) { return MemoryT<U>(*this); }

template<> template<typename U>
MemoryT<std::string>::MemoryT<std::string>(const MemoryT<U>& rhs)
    : MemoryT<std::string>({ rhs.size_.size }) {
    for (size_t i = 0; i < size_.size; ++i)
        data_[i] = std::to_string(rhs.data_[i]);
}
```